Verification Corpus

Combined reference papers for Kubernetes autofix research

Contents

1 **Arsc: Automated Reconciliation for State Correctness of Cluster Management** **2**

2 **AutoNet: Automatic Reachability Policy Management in Public Cloud Networks** **13**

3 **Large-scale Cluster Management at Google with Borg** **23**

4 **Centralized Defense: Logging and Mitigation of Kubernetes Misconfigurations with Open Source Tools** **41**

5 **ChaosEater: Fully Automating Chaos Engineering with Large Language Models** **58**

6 **GenKubeSec: LLM-Based Kubernetes Misconfiguration Detection, Localization, Reasoning, and Remediation** **172**

7 **AI-Powered Anomaly Detection for Kubernetes Security: A Systematic Approach to Identifying Threats** **187**

8 **Enhancing Kubernetes Resilience through Anomaly Detection and Prediction** **194**

9 **A Generic Approach for Reactive Stateful Mitigation of Application Failures in Distributed Robotics Systems Deployed with Kubernetes** **221**

10 **KGSecConfig: A Knowledge Graph Based Approach for Secured Container Orchestrator Configuration** **228**

11 **LLMSecConfig: An LLM-Based Approach for Fixing Software Container Misconfigurations** **240**

12 **Magpie: Python at Speed and Scale using Cloud Backends** **253**

13 **PRISTINE: Priority-Aware Smart Resource Orchestration Engine for Cloud-Native Applications** **317**

2025 IEEE 45th International Conference on Distributed Computing Systems (ICDCS) | 979-8-3315-1723-6/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICDCS63083.2025.00015

Arsc: Automated Reconciliation for State Correctness of Cluster Management

Hongjing Yu†, Chaoqun Li†, Tao Han†, Pengfei Hu†*
† School of Computer Science and Technology, Shandong University, Qingdao, China
{yuhj, chaoqunli, hantaotao}@mail.sdu.edu.cn, phu@sdu.edu.cn

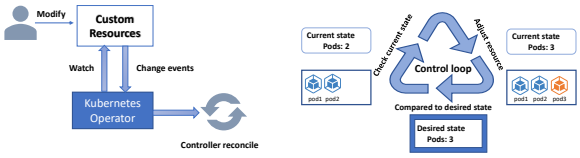
***Abstract*—Modern cluster management platforms, such as Kubernetes, implement declarative interfaces based on state-centric principles and extend cluster capabilities through operators. Once the desired system state is declared, operators automatically reconcile the actual state to align with the desired state. Ensuring state correctness after operations is critical for system safety and stability, as incorrect or unhealthy states can lead to serious consequences. However, there is currently no systematic process in place to guarantee state correctness. The vast state space presents significant challenges in detecting and correcting abnormal states. To tackle this issue, we propose Arsc, an automated technique for test generation, detection, and correction aimed at maintaining state correctness. Arsc establishes the definition of state correctness, offers a state detection method, and presents three strategies for state correction. It was evaluated on five popular open-source operators, generating 11,581 tests and detecting 208 incorrect states, 71 unhealthy states, and 28 crashes, all of which were effectively corrected.**

***Index Terms*—State correctness, Reconciliation, Kubernetes, Operator, System testing**

I. Introduction

With the rapid expansion of cloud computing, the number of nodes in a cluster has grown exponentially, making traditional manual management impractical. Modern cluster management platforms, such as Kubernetes [1], Docker Swarm [2], and Apache Mesos [3], offer high automation, reliability, and continuity. Operators in cluster management extend the cluster's capabilities by automating tasks such as container deployment, networking, and load balancing, significantly improving resource management efficiency and reducing operational costs. Nowadays, the operator pattern is becoming more and more thriving. Taking Kubernetes as an example, an increasing number of commercial and open-source projects have adopted operators to automate various tasks. According to data from OperatorHub, the official Kubernetes-recommended operator community, over 370 operators have been released [4], many of which have been widely deployed in the production environment [5]–[7].

In Kubernetes, operators follow a declarative design pattern with state reconciliation, relying on Custom Resource (CR) [8] and controllers [9]. Operators expose a declarative interface via CRs, which define modifiable properties for managing resources. State declarations specify the property values within CRs. Controllers monitor the resources, and when the state declaration changes, they reconcile the current state to ensure the resources remain in the desired state. Fig.1a briefly illustrates the workflow of an operator, and Fig.1b shows the control loop in the state reconciliation process.



(a) Operator workflow. (b) State reconciliation in controller.

Fig. 1: Operator's working pattern.

The widespread deployment of operators makes ensuring the correctness of their reconciliation results a pressing need. State correctness is crucial for system security and reliability; incorrect or unhealthy states can lead to severe consequences. For example, Fig.2 illustrates an incorrect state in a RabbitMQ cluster. The RabbitMQ operator expects the cluster to have 3 replicas and waits for all replicas to be ready. However, the replica count is overridden to 2 via *statefulSet.spec.replicas*, despite *spec.replicas* being set to 3. As a result, the actual number of replicas does not meet the operator's expectations, preventing progress. More critically, the *StatefulSet* will not be updated until the operator detects 3 replicas, resulting in a persistent incorrect state. Such errors can lead to performance degradation or even serious service outages.

To address state anomalies, users need a reliable method to ensure system state correctness. Most research focuses on improving controller reliability and detecting errors due to operator design flaws [10]–[12]. In contrast, our work aims to develop a technique for detecting and correcting state anomalies that occur during operator operation. These anomalies can arise from various sources, not just potential flaws in the operator itself. We present Arsc, an automated tool for ensuring state correctness in cluster management. Arsc generates state declarations to guide operators through the workflow and monitors the system state during operation. If anomalies are detected, Arsc takes corrective actions. State correctness is defined by three criteria: (1) conformity with the desired state, (2) a healthy state, and (3) if the current state is abnormal, returning at least to the previous correct state.

Arsc consists of three main functional modules: the generation module, the detection module, and the correction module.

* Pengfei Hu is the corresponding author.

2575-8411/25/$31.00 ©2025 IEEE
DOI 10.1109/ICDCS63083.2025.00015

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:45:53 UTC from IEEE Xplore. Restrictions apply.

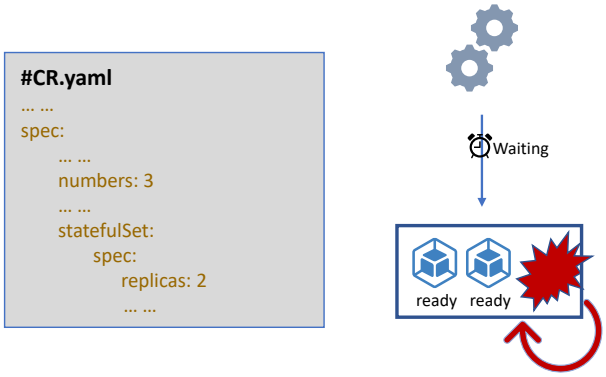


Fig. 2: An incorrect state in RabbitMQ cluster.

In the generation module, Arsc creates state declarations by parsing the Custom Resource Definition (CRD), which defines property constraints in the *OpenAPISchema* format, to explore state transitions in different scenarios. In the detection module, Arsc detects whether the system state is correct. The complex state space presents a challenge for us. We focus on system resources, particularly CRs, and establish the definition of state correctness along with a method for state detection. In the correction module, Arsc corrects detected anomalies in three ways. State recovery fixes issues caused by failed adjustments or temporary faults by modifying resources or configurations. State rollback handles anomalies from invalid declarations or irrecoverable failures by reverting to a previous state. Restart restores the system to its initial state after a crash. The operator continuously reconciles the system state based on a sequence of generated state declarations. Each state declaration drives the operator to transition the system to a new state, from which the subsequent operation begins. This process leverages the level-triggering principle [13].

We evaluated Arsc on five popular Kubernetes operators. Arsc generates 11581 state declarations and detects 208 incorrect states, 71 unhealthy states, and 28 crashes. Arsc was able to effectively detect system anomalies and perform reliable correction.

Contributions. The main contributions of this paper are summarized as follows:

- We present an automated technique that ensures state correctness for cloud system operators.
- We implement Arsc, an automated generation, detection, and correction tool based on state-centric principle.
- Arsc has already been evaluated on five Kubernetes operators. The results show it is reliable.

II. Related Work

In this section, we provide an overview of related work in the areas of controller correctness, system verification, cluster management verification, and state space exploration. These fields are crucial for understanding the current approaches and techniques used to ensure the reliability, correctness, and efficiency of distributed systems, particularly in the context of Kubernetes.

A. Controller Correctness

As part of a literature review on controller correctness and verification techniques, several notable works have been identified that contribute to the field. Sun et al. [10] employed fault injection techniques to verify the reliability of controller implementations, aiming to identify potential faults. Gu et al. [12] adopted an end-to-end testing approach to evaluate the correctness of controllers. Liu et al. [11] utilized model checking to verify controllers and their configurations. In a broader context, Sun et al. [14] proposed a general controller correctness specification framework to formalize the process of verifying controller implementations, providing guidelines for ensuring that controllers align with their intended functionality and meet the requirements of distributed systems. These works primarily focus on identifying potential flaws within the controller itself, while our approach examines the system to detect state anomalies. These anomalies may arise from various factors, including but not limited to potential flaws in controllers.

B. System Verification

Verification techniques have been successfully applied to various distributed systems [15]–[23] and network protocols [24]–[28]. For example, Yabandeh et al. [29] used model checking methods to prevent inconsistencies in distributed systems by analyzing possible state transitions. Hawblitzel et al. [30] focused on system liveness. While Kubernetes shares the characteristics of distributed systems, existing works cannot be directly applied. Kubernetes tends to focus on verifying resource management and scheduling, whereas other distributed systems may prioritize properties such as data consistency or fault tolerance. Moreover, the resource states in Kubernetes differ significantly, as they involve the dynamic management of large-scale systems with complex dependencies.

C. Cluster Management Verification

Formal verification methods have also been applied to cluster management systems to ensure the correctness of their operational behavior. Turin et al. [31] presented a formal model of Kubernetes based on simplified assumptions of controllers but did not test it for verification. Liu et al. [32] proposed a proof-of-concept method, focusing on modeling several failure scenarios. Ding et al. [33] applied verification techniques to serverless architectures, with a particular focus on idempotence properties from the application perspective. Our work focuses on state, particularly resource state, and aims to detect system state in real Kubernetes working environments.

D. State Space Exploration

Exploring the state space of complex systems is a challenging task due to its vastness and complexity. Godefroid [34] proposed a method that divides the state space into

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:45:53 UTC from IEEE Xplore. Restrictions apply.

local and global states, which reduces the complexity of state space search and makes it more feasible to explore large-scale systems. Yang et al. [35] applied state space reduction techniques to storage systems. Yabandeh et al. [29] used distributed snapshots to explore the state space to a certain depth from an intermediate state during runtime, checking whether the system state violates a given property. The state space in Kubernetes is complex, and Godefroid's work has provided us with valuable insights. Based on the operational characteristics of operators, after state reconciliation, we focus on CRs and the dependent resources related to them.

III. Motivation

The operator pattern flourishes. The scalability of Kubernetes has fostered a thriving ecosystem of thousands of domain-specific operators [36]–[39]. Thousands of operators developed by both commercial vendors and the open-source community. For instance, OperatorHub [4] offers over 370 operators, and ArtifactHub [40] offers over 1000 operators. While these technological advancements provide significant convenience and flexibility, they also introduce new challenges. As the Kubernetes ecosystem continues to grow, ensuring the state correctness has become a key issue that needs to be addressed.

Kubernetes clusters are state-centric. The functionality and stability of the system depend on the continuous management and synchronization of its current state. We analyzed 64 Common Vulnerabilities and Exposures (CVEs) listed in the official Kubernetes Security Response Committee and found that most directly affect the system state, with two specifically related to CRs [41]. Any deviation between the actual and desired states can lead to service disruptions or even cause the entire system to crash. This highlights the importance of effective state management and correction mechanisms as the cornerstone for maintaining a reliable and resilient Kubernetes environment.

The need for comprehensive method. Existing research primarily focuses on identifying vulnerabilities or implementation issues within the operator itself. In contrast, there is a lack of methods focused on detecting and ensuring the correctness of Kubernetes cluster state. By monitoring the dynamic changes of the cluster state, we can safeguard the system's stability and reliability at a broader level, rather than solely concentrating on the security of individual operators.

IV. Technique

Arsc is committed to maintaining the correctness of the system state by monitoring system resources. It detects anomalies when state correctness requirements are violated and takes action to reconcile the system. Such anomalies include: (1) not meeting the desired state, (2) being in an unhealthy state, and (3) a system crash may occur under special circumstances.

To use Arsc, users need to provide the configuration information of the operator, guiding Arsc to correctly deploy the operator. Arsc accepts CRD as input and ultimately provides detecting and correction results. The workflow of Arsc is described in section IV-A.

A. Workflow

As shown in Fig. 3, Arsc consists of three main modules: the generation module, the detection module, and the correction module. Its workflow can be summarized in seven steps: ① deploying the operator, ② parsing the CRD, ③ generating state declarations, ④ reconciling the state, ⑤ detecting whether the system state is correct, ⑥ correcting the state, and ⑦ collecting the results. The following describes the five main components of Arsc:

Parser: The parser takes the CRD file as input, parses it into the target format, and performs both syntactic and semantic analysis. It extracts key properties and analyzes their usage scenarios. As one of the core components for generation module, it provides a foundation for targeted state declaration generation through structural parsing and semantic analysis of the CRD.

Generator: The generator automatically creates CR files that declare the desired state based on the parser's results. We ensure that the generated state declarations are syntactically correct, but not necessarily semantically correct. In other words, in addition to valid operation scenarios, the generator also generates invalid declarations to test whether Arsc can detect and correct state anomalies in time. The process of generating state declarations is described in detail in section IV-B.

Monitor: The monitor is responsible for real-time monitoring of the system state. It interacts with the Kubernetes API to regularly collect system operation data, such as resource states, system events, etc. Once the system state is deemed stable, it captures a system state snapshot for subsequent state detection and correction. The key point is explained in section IV-C.

Checker: The checker compares and checks the differences between the current system state and the desired state. Its inputs include the real-time system state obtained from the monitor and the desired state provided by the generated state declaration. In section IV-D, we establish the definition of state correctness and describe how to detect resource state.

Mediator: The mediator is the key correction component of Arsc, responsible for attempting to restore the system to a correct state by automating resource adjustments when system anomalies are detected. The mediator provides three strategies, with the goal of achieving "self-healing", which are explained in section IV-E.

In short, the parser and generator are responsible for generating CR files which provide state declarations, while the monitor and checker handle system state detection. The checker and mediator are tasked with correcting state anomalies.

B. CR Files Generation

1) CRD Parsing: The core task of a Kubernetes operator is to automatically manage the cluster based on the state declarations of CRs. CRD defines the structure, fields, and validation

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:45:53 UTC from IEEE Xplore. Restrictions apply.

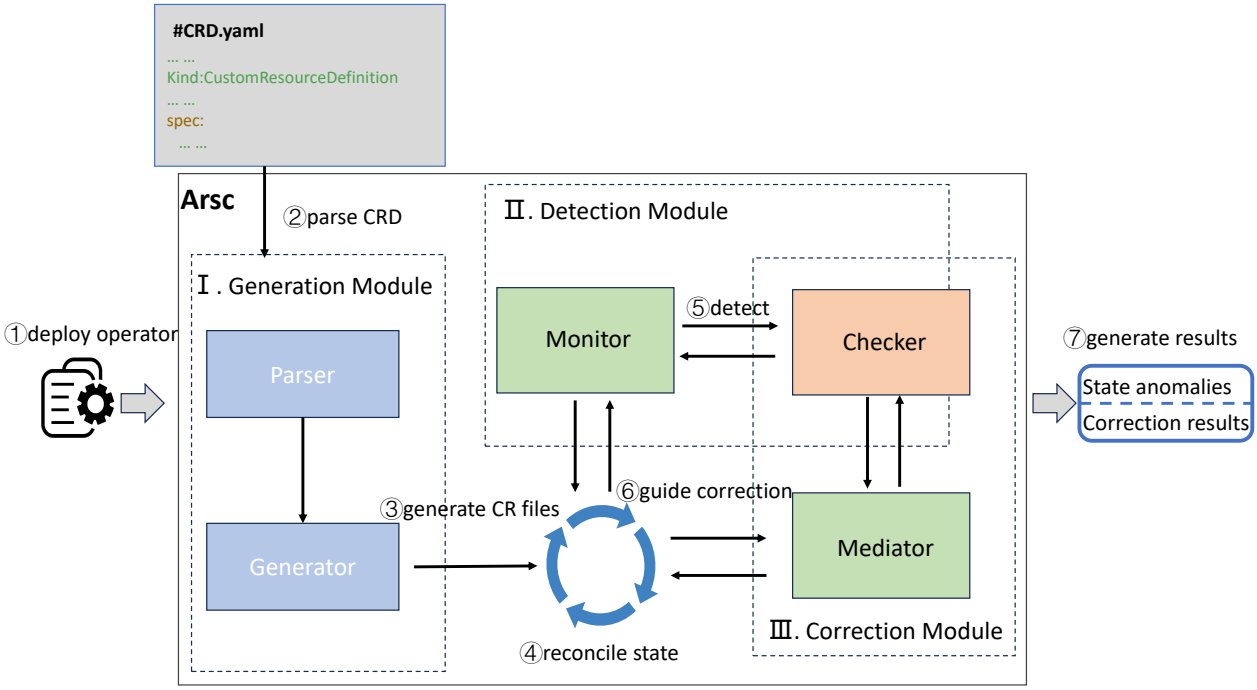


Fig. 3: The workflow of our proposed Arsc method.

rules of CRs. The parser analyzes the CRD files, extracting structural and property information, while the generator creates the CR files. The operator monitors changes in the CRs and performs the corresponding actions based on those changes.

Syntax Analysis: The CRD's *spec.openAPI3Schema* specification defines constraints for all supported properties. The parser extracts these constraints to ensure that all property values are syntactically valid. For simple properties and basic data types (e.g., *type: string*), the type constraints, value range limitations, and format requirements are extracted, and the dependencies between fields are recorded. For nested properties and complex data types (e.g., *type: object*), each possible combination of conditions is expanded, generating multiple independent branches. For special keywords (e.g., *anyOf*, *oneOf*, *allOf*), if complex cross-field constraints exist, corresponding constraint expressions are generated and marked as logical points. In practice, the structure of CRD files in real-world is often much more complex, resulting in a large number of branches during the search process. To minimize complexity, we define pruning strategies that terminate the search early for branches determined to be redundant (e.g., *description*). We also employ heuristic searches to prioritize fields that significantly impact operator behavior (e.g., *replicas*, *scale*, *timeout*, etc.).

Semantic Analysis: Syntax analysis provides only basic information, such as the data type and the constraints of properties. Obtaining the functionality and usage scenarios requires further semantic analysis. The most straightforward method for semantic inference is to deduce from the field name. For example, common naming patterns, such as *minXXX* or *maxXXX*, can provide hints during semantic inference. Additionally, we create a mapping table based on Kubernetes terminology, which helps us infer the potential semantics by matching them with Kubernetes terms.

The actual semantics of fields may be ambiguous due to differences in naming conventions, domain knowledge, and context. Moreover, CRD is static and does not directly reflect how an operator dynamically adjusts resources at runtime. By tracking the data flow of a property, we can determine how the property value is used. If a property value is passed to the Kubernetes API or assigned to Kubernetes resource objects, we use the mapping table to map the property to the corresponding Kubernetes object. System state changes often involve aspects such as replica counts, resource configuration, and lifecycle management. We focus on properties relevant to these scenarios.

2) Generating Strategy: Based on the results of syntax and semantic analysis, all properties are assigned values using different strategies depending on their *type*. The generator focuses on exploring various scenarios based on semantics, with each property value representing a specific scenario. For properties where semantic analysis fails, the generator generates property values according to syntactic rules. These values may be semantically valid or invalid.

C. State Transition

1) State Reconciliation: A series of CR files are generated and sequentially applied to the cluster. The operator

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:45:53 UTC from IEEE Xplore. Restrictions apply.

continuously monitors resource change events. When a state declaration is applied to the cluster, the operator receives an event notification. It then captures the current state of the cluster resources, analyzes the desired state described in the state declaration, and generates a reconciliation plan based on the differences. Finally, the operator invokes the corresponding resources through the Kubernetes API and performs the necessary operations to bring the current state into alignment with the desired state.

2) Capturing System State: To detect system state, the monitor needs to capture a system-wide snapshot in order to fully understand the state of the cluster.

Checkpoint: Ideally, the snapshot captured after the operator reconciliation should represent the up-to-date and stable state, rather than an intermediate or incomplete state during the process. Therefore, the timing of checkpoint selection is crucial. Capturing the system state too early may lead to false positives, as the system has not fully completed reconciliation. Capturing it too late may waste resources or miss critical points of fluctuation in the system state. To optimize this process, we drew on the work of Sun et al., Anvil [14], which introduced the concept of Eventually Stable Reconciliation (ESR). ESR focuses on two core concepts: progress and stability, asserting that the system should reconcile the cluster's state to meet the desired state and maintain that state after reconciliation is complete. We adapted this concept by using a dynamic adjustment mechanism to select checkpoints, employing delay injection to capture the stable state after reconciliation.

System Snapshot: When capturing a snapshot, the focus is on resources (e.g., *Deployments*, *Pods*, *Services*) and external dependencies (e.g., *ConfigMaps* and *Secrets*). Information is recorded at multiple levels, including resource metadata, runtime status, and interactions with external dependencies.

D. State Detection

1) State Correctness Definition: In Kubernetes, resources refer to the various objects used for management and scheduling within the cluster. Each resource has specific functions and properties, and Kubernetes uses these resources to achieve automated management and reconciliation of the cluster. The state of a resource reflects its current configuration and operational status. The system state provides an overall description of these resource configurations. Tab. I lists the common types of resources in Kubernetes, including native and custom resources. These resource types form the foundation of Kubernetes management and scheduling.

Kubernetes resources often depend on each other, where the proper functioning of one resource relies on the state of another. For example, a pod may depend on a configMap for configuration, or a service may rely on a deployment to manage pods. Understanding these dependencies is crucial for grasping how resource states impact each other in the cluster.

We can represent the system's resources as a set $R = \{R_1, R_2, \ldots, R_n\}$, where $R_i \in R$ $(1 \leq i \leq n)$. The dependency relationships between these resources are defined as E, such that if R_i depends on R_j, then $(R_i \rightarrow R_j) \in E$.

The system state is denoted as $S(R)$, where $s(R_i)$ represents the actual state of resource R_i, and the desired state is denoted as $D(R)$, where $d(R_i)$ represents the desired state of resource R_i.

Definition 1. *Given the system state $S(R)$ and the desired state $D(R)$, we say that S matches D, denoted as $S \sim D$, if the following conditions are satisfied:*

- *For each resource $R_i \in R$, the actual state $s(R_i)$ must match the desired state $d(R_i)$.*
- *For each dependency $(R_i \rightarrow R_j) \in E$, the actual state $s(R_j)$ of the dependent resource R_j must satisfy the requirements set by R_i.*

Formally, this consistency is defined as:

$$S \sim D \iff \forall R_i \in R,\ s(R_i) = d(R_i) \wedge$$
$$\forall (R_i \rightarrow R_j) \in E,\ s(R_j) \text{ satisfies the requirements of } R_i.$$

After establishing the consistency between the system state and the desired state ($S \sim D$), it is essential to address the health of the system's resources. While $S \sim D$ ensures that the actual state of the resources aligns with the desired state, it does not necessarily guarantee that the resources are functioning optimally. The health of the resources, defined by the "SAR" conditions, plays a critical role in ensuring the stability, availability, and responsiveness of the system.

Definition 2. *For any $R_i \in R$, the system state is considered healthy, denoted as $S \sim H$, if $s(R_i)$ satisfies the "SAR" conditions, where:*

- *Stable: The resource remains stable without frequent restarts or failures.*
- *Available: The resource is ready to accept requests and provide services.*
- *Responsive: The resource responds to traffic or requests in a timely manner, ensuring good service quality.*

After defining the consistency between the system state and the desired state ($S \sim D$), as well as the health condition of the resources ($S \sim H$), we can conclude that the system state is correct if both conditions are satisfied.

Definition 3. *If $S \sim D$ and $S \sim H$, then the system state is considered correct, denoted as $S \sim C$.*

2) Local Detection: Local detection focuses on detecting the state correctness by checking only the custom resources and their dependent resources, rather than examining the entire set of system resources. This approach reduces the scope of the detection process, making it more efficient and scalable.

The core idea behind local detection is that the system's behavior and overall state are primarily determined by the custom resources and the resources that directly interact with them, particularly in the context of the operator pattern. By narrowing the detection scope to these resources, we can capture inconsistencies between the actual and desired system

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:45:53 UTC from IEEE Xplore. Restrictions apply.

TABLE I: Some resources in cluster.

Kind	Resource	Description
Computing resource	Pod	Pod is the smallest unit of compute in Kubernetes and usually consists of one or more tightly coupled containers.
	Deployment	Deployment is a declarative resource used to manage the lifecycle of Pods.
	StatefulSet	StatefulSet is used for managing stateful applications like databases.
	DaemonSet	DaemonSet ensures that a copy of a Pod is running on each node in the cluster.
Storage resource	PersistentVolume	PersistentVolume abstracts physical storage devices and manages the lifecycle of storage volumes.
	PersistentVolumeClaim	PVC is a request for storage resources by a Pod.
	ConfigMap	ConfigMap is a resource used to store non-sensitive configuration data that can be shared across multiple Pods.
	Secret	Secret is a resource for storing sensitive data like passwords, API keys, etc.
Network resource	Service	Service is a Kubernetes network resource used to expose a set of Pods and provide load balancing.
	Ingress	Ingress manages external HTTP and HTTPS traffic to services in the cluster.
	NetworkPolicy	NetworkPolicy defines the rules for network communication between Pods, controlling which Pods can communicate and which cannot.
Scheduling and management	Namespace	Namespace is a mechanism for dividing resources into logical groups within a Kubernetes cluster.
	Node	Node is a worker machine in Kubernetes, responsible for running containerized workloads.
	ResourceQuota	ResourceQuota limits the total amount of resources that can be used within a namespace, preventing resource overuse.
Custom resource	-	Custom resource extends Kubernetes by defining my own resource types.

states, while reducing the overhead of checking irrelevant resources.

The process can be broken down as follows:

- Identify the custom resources: The primary focus is on the custom resources, which represent user-defined objects and configurations in the Kubernetes system.
- Determine dependencies: Identify the native resources that the custom resource depends on, such as *Pod*, *Service*, *ConfigMap*, or other Kubernetes objects.
- Verify state consistency: For the custom resources and its dependencies, check if the actual state matches the desired state, as defined by the state specifications. Also, verify the health status of the resources.
- Detect anomalies locally: Any inconsistencies or unhealthiness found in the custom resources or its dependent resources are flagged for further action.

3) Global Detection: Global detection focuses on detecting the health of system resources that are independent of custom resources. It can be roughly assumed that the state of resources not mentioned in the state declaration remains stable and only needs to be checked for compliance with Definition 2.

This assumption is based on the premise that the system starts from a healthy state. The cluster configuration is typically rigorously validated by the vendor during the deployment phase, which minimizes the risk of errors or inconsistencies in the initial state. Since the system is deployed with a healthy and validated configuration, the primary focus of detection shifts to ensuring the ongoing health of native resources within the cluster. Therefore, global detection emphasizes detecting whether resources remain stable, available, and responsive over time.

4) Detection Method: In practice, both detection methods are applied simultaneously, complementing each other to provide a comprehensive solution for system state detecting. Local detection focuses on detecting custom resources and their direct dependencies, offering a more targeted approach to ensuring state consistency within the operator's scope. Global detection, on the other hand, shifts its focus to maintaining the ongoing health and stability of native resources, enabling monitoring of the system's overall state.

E. State Correction

1) State Recovery: In the case of simple system anomalies, the first step is to attempt reconciliation in the current state. Upon detecting an anomaly, the mediator triggers event filtering, which serves as the system's immediate response mechanism. This step helps prevent misoperations and limits the propagation of errors. Event filtering is implemented by inserting hooks into the event processing flow, which are supplemented by timer-based delay logic.

Event filtering provides the system with sufficient time to recover and return to the correct state. After event filtering, the mediator initiates corrective actions to restore the system to the desired state. These corrective actions heavily rely on the controller's self-healing capabilities, particularly its built-in control loop. Such actions include resource adjustment, rescheduling, and fixing configuration errors, and more. The process is illustrated in Fig. 4.

2) State Rollback: If the state recovery mechanism fails to reconcile the system to a correct state, the rollback mechanism is triggered to restore the system to the last known correct state. This rollback mechanism ensures a "safe starting point" for subsequent detection and intervention. The process follows these steps: (1) When the system state is found to be abnormal and cannot be corrected through simpler recovery operations, the rollback mechanism identifies the most recent valid state snapshot. (2) It then restores critical components to this known state, clearing any operations or data that may have caused the current inconsistencies. (3) Once the rollback is complete, the

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:45:53 UTC from IEEE Xplore. Restrictions apply.

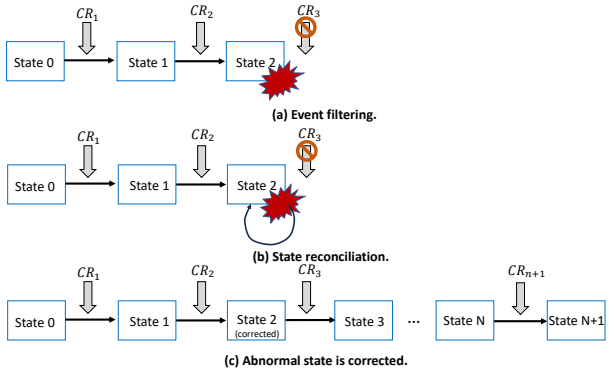


Fig. 4: State recovery process.

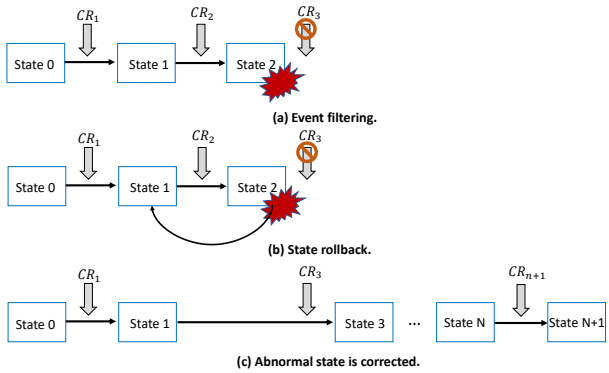


Fig. 5: State rollback process.

system resumes operation from the restored state and proceeds with further operations. The rollback process is shown in Fig. 5.

3) Restart: In cases where the system encounters an unrecoverable crash due to a critical error, a full restart from the initial state is required. This ensures that the system starts fresh, eliminating the possibility of residual inconsistencies. The restart process is depicted in Fig. 6.

4) Strategy Priority: State recovery is suitable for state anomalies that can be resolved through resource adjustments and similar measures. State rollback is used for irrecoverable state anomalies. Restart is employed in the event of a system

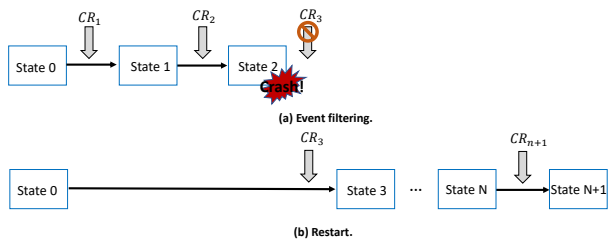


Fig. 6: Restart process.

crash. The anomalies detected by the detection module can be categorized into three types: incorrect state, unhealthy state, and system crash. After detecting a state anomaly, selecting the appropriate correction strategy is crucial to ensuring system stability. We prioritize three correction strategies in the following order: state recovery, state rollback, and restart. For both incorrect and unhealthy states, the state recovery strategy is applied first. If state recovery is insufficient, the state rollback strategy is used. In the case of system crashes, the restart strategy is typically employed.

5) State Recheck: To ensure the correctness and effectiveness of the correction process, the system undergoes comprehensive revalidation after any corrective action. This revalidation includes checking the state of individual components, verifying the consistency of inter-component dependencies, and confirming overall system health. The recheck process not only ensures that the system has returned to the desired healthy state but also verifies that business logic is functioning as expected and that previous abnormalities have been successfully resolved.

F. Implementation

Arsc is designed to ensure state correctness in cluster management. It operates on a virtualized Kubernetes cluster. Its primary focus is the analysis of system resources, offering a degree of generality. Arsc executes tasks sequentially: when the Kubernetes API receives a new state declaration, the operator compares the current state with the desired state and makes the necessary adjustments. Once Arsc confirms that the system state is correct, the Kubernetes API can receive the next state declaration, meaning the current state serves as the starting point for the subsequent state.

V. Evaluation

Our experiments were conducted on a Cambricon server equipped with two Intel(R) Xeon(R) Platinum 8358 CPUs (32 cores) and 503GB of memory, running Ubuntu 22.04.3 LTS with kernel version 5.15.0-119-generic. We used Kind [42], a tool designed to run Kubernetes clusters within Docker containers, to quickly deploy the Kubernetes cluster. Considering system load, we configured only one worker node for the experiment.

To evaluate Arsc, we tested it with five popular open-source Kubernetes operators, all of which were developed either by the official teams of the managed systems or by companies offering services built around these systems (Tab. II). We focus on three key questions: (1) Can Arsc generate efficient state declarations? (2) Can Arsc detect system anomalies? (3) After detecting anomalies, can Arsc successfully correct the system? The main evaluation results are summarized below:

- Arsc generates 11,581 state declarations, and after analysis, 11387 are valid and 194 are invalid.
- Arsc detects 208 incorrect states, 71 unhealthy states, and 28 crashes.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:45:53 UTC from IEEE Xplore. Restrictions apply.

- Arsc applies different correction strategies based on the types of anomalies and ensures effective correction for all detected state anomalies.

TABLE II: The operators we used.

Operator	Provider	Categories
RocketMQ	Apache Software Foundation	Streaming & Messaging
Cockroach	Cockroach Labs	Database
CloudnativePG	CloudNativePG Contributors	Database
Zookeeper	StreamNative	Storage
RabbitMQ	VMware Tanzu	Streaming & Messaging

A. Generation Results

1) Validity of Declaration: Valid state declarations clearly and accurately describe the desired state, with parameters and configurations strictly adhering to the defined specifications, enabling the system to correctly parse and execute them. Specifically, the replica counts, resource requests, and other parameters in the generated CR files conform to the CRD requirements, allowing the system to allocate resources and achieve the desired state based on these valid declarations. In contrast, invalid state declarations fail to meet expectations or rules, possibly due to configuration errors, out-of-range parameter values, or violations of constraints, rendering them unrecognizable or unprocessable by the system. The generator generated 11,581 state declarations, of which 11387 were valid and 194 were invalid. We present the generation results in Tab. III.

Invalid declarations can lead to incorrect or unhealthy states and may even cause system malfunctions or crashes. In the correction process, we chose to skip these invalid declarations to maintain system correctness.

TABLE III: Generation results.

	Type	Num	Percentage
Valid	-	11387	98.32%
Invalid	Invalid field	43	0.37%
	Invalid value	65	0.56%
	Invalid scale	37	0.32%
	Invalid label	35	0.30%
	Others	14	0.12%
	Total	11581	-

2) Property Coverage: Arsc achieves 100% property coverage for each operator, with the generator ensuring at least one operation for every property. The automated generation of state declarations not only covers a wide range of properties but also generates more property values and implements a greater number of state transitions.

B. Detection Results

Arsc detects 208 incorrect states, 71 unhealthy states, and 28 crashes. Detailed results are provided in Tab. IV.

1) Incorrect State: Arsc identified 208 instances of incorrect states that violated state declaration. An incorrect state refers to a discrepancy between the system's actual state and its desired state. This does not imply that the system is completely crashed or unhealthy, but rather indicates issues such as configuration errors, resource allocation problems, or state declaration exceptions, which prevent resources or system behavior from meeting the desired objectives.

TABLE IV: Detection results.

Operator	Test	Incorrect	Unhealthy	Crash
RocketMQ	1584	27	0	0
Cockroach	737	29	9	12
CloudnativePG	2093	32	40	6
Zookeeper	2880	79	16	5
RabbitMQ	4287	41	6	5
Total	11581	208	71	28

Case 1: An issue was observed in the RabbitMQ cluster, which has been mentioned in fig. 2. The number of replicas is overridden to 2 via *statefulSet.spec.replicas*, even though *spec.replicas* is set to 3. After the system scales to two replicas, the operator stops reconciling, causing the number of replicas to no longer match the desired state.

Case 2: This is an issue found in the RocketMQ cluster where incorrect states were caused by the inclusion of invalid fields. The *spec.resources* field is used to define resource requests and limits for containers, typically including standard resource types such as CPU, memory, and storage. These fields have strict format requirements to ensure accurate and consistent resource allocation. However, adding invalid fields in the resources configuration prevents Kubernetes' internal resource management system from recognizing them, leading to inaccurate resource allocation.

2) Unhealthy State: Arsc identified 71 instances of unhealthy states. An unhealthy state refers to a condition where the system or resource is running normally, but its health status does not meet expectations, potentially indicating underlying faults or instability.

Case 1: This issue was discovered in the RocketMQ cluster, caused by the use of invalid values in the configuration fields. The *spec.env* field is used to define environment variables for containers, and the values can be set using either the *value* or *valueFrom* field. However, *value* and *valueFrom* are mutually exclusive fields. If both are set for the same environment variable, it results in a configuration conflict. The Kubernetes API server returns a validation error, indicating the field conflict and refusing to apply the configuration.

Case 2: A pod startup failure was observed in the RabbitMQ cluster. The cluster added two pods and their corresponding persistent volumes through the state declaration. However, due to false binding of dependencies, both newly added pods remained in the Pending state.

3) Crash: Arsc identified 28 instances of crashes. A crash refers to a complete failure of the system or service, rendering it unable to operate normally. System crashes typically occur when a process or node fails, causing part or all of the system to be unable to provide services.

Case 1: This crash scenario was observed in a Cloudnative cluster. When the *ephemeralVolumesSizeLimit* is not specified, it indicates that there is no explicit limit on the ephemeral storage size for PostgreSQL cluster instances. However, when

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:45:53 UTC from IEEE Xplore. Restrictions apply.

the shared memory size is explicitly configured to 4Gi using *ephemeralVolumesSizeLimit.shm*, it sets a 4Gi limit on shared memory for each PostgreSQL cluster instance. Under this constraint, attempting to allocate memory beyond the shared memory limit results in a system crash.

Case 2: This crash scenario occurred in a Zookeeper cluster. The *pod.securityContext* field configures the security context of a pod, including several key settings: *fsGroup* defines the filesystem group inside the container, *runAsGroup* sets the process group, *runAsUser* defines the user ID for the process, and *supplementalGroups* specifies additional group IDs. The crash was caused by an incompatibility between the *pod.securityContext* configuration and the filesystem permissions, storage volume access, and cluster security policies.

4) False Positive: A total of 194 state anomalies were caused by invalid state declarations, while 113 resulted from other factors. Additionally, we identified one false positive in RabbitMQ, caused by the unclear impact of the *skipPostDeploySteps* property on state. Therefore, the overall false positive rate of Arsc is 0.32%.

C. Correction Results

Once a state anomaly is detected, the correction module selects the state recovery, state rollback, or restart solution in order of priority. The correction results for the three types of state anomalies are shown in Tab. V.

TABLE V: The correction rate of abnormal state.

Strategy	Incorrect	Unhealthy	Crash
State recovery	11.06%	87.32%	-
State rollback	88.94%	12.68%	-
Restart	-	-	100%

1) Correction for Incorrect State: For incorrect states, the state recovery strategy is prioritized, achieving a correction rate of 11.06%. For states that cannot be corrected by the state recovery strategy, the state rollback strategy is applied.

The incorrect states identified in our experiments fall into two main types: The first type is adjustment failure, where the system fails to adjust the resource state to match the desired state, as illustrated in Case 1 in section V-B1. For this type of incorrect state, we apply the state recovery strategy, which restores the system to the desired state by re-adjusting the resources. The second type is invalid state declarations, where users may incorrectly declare resource configurations in custom resources, leading the system to believe the resources are correctly configured when, in fact, they do not meet expectations. Our experiments show that this type of error constitutes the majority of incorrect states, approximately 88.94%. In this case, we correct the anomaly by rolling back to the previous correct state and skipping the invalid declaration. This approach not only proved effective in experiments but also offers practical insights for real-world applications. In practice, invalid state declarations are almost inevitable, and therefore, employing effective strategies to skip these declarations and avoid potential errors is essential.

2) Correction for Unhealthy State: The causes of unhealthy states are primarily due to insufficient system resources, configuration errors, and external dependency failures. To address these issues, the state recovery strategy adjusts resource limits, optimizes resource allocation, and waits for external services when necessary. For irrecoverable unhealthy states, such as Case 1 in the section V-B2, the state rollback strategy is applied. The correction strategy for unhealthy states is similar to that for incorrect states.The state recovery strategy can correct 87.32% of unhealthy states, while some states, due to invalid state declarations or other special reasons, may require the state rollback strategy.

3) Correction for System Crash: Restarting resolved all identified crash issues. When a system crashes, it is typically due to a severe error or exceptional condition that prevents recovery or continued operation. Compared to attempting more complex recovery methods, restarting is the simplest and most effective solution. It immediately restarts the process, ensuring the service begins from its initial state, reloads all configurations, and resumes normal operation. This method is universally efficient, but we do not advocate for its use as a preferred solution.

4) Reflection: The current correction strategies ensure that the system maintains the correct state, but in some cases, they may lead to the loss of certain progress, particularly in the case of meaningless state declarations. This is unavoidable, and we believe that skipping such operations is acceptable.

D. Execution Time

We deployed five open-source operators and conducted evaluation experiments on their respective clusters. As shown in Tab. VI, the total execution time ranges from 22.53 to 133.66 hours across the operators. The majority of Arsc's runtime is spent on detection and correction. The experimental duration varies significantly due to the differing operational complexity of each operator.

TABLE VI: Arsc execution time for per operator (hours).

Operator	Generation	Detection & Correction	Total
RocketMQ	0.01	22.52	22.53
Cockroach	0.01	97.28	97.29
CloudnativePG	0.02	95.34	95.36
Zookeeper	0.02	136.64	136.66
RabbitMQ	0.03	75.04	75.07

VI. Limitations and Future Work

Limitation in scenario coverage. Modern cloud cluster systems consist of a wide range of components and resources, leading to a vast and highly dynamic state space. Due to the complex interactions among these components, system anomalies can manifest in numerous patterns. As a result, abnormal events are not only difficult to predict but also challenging to reproduce in a controlled environment. When generating test cases, the complexity and variability of the state space make it difficult to ensure comprehensive coverage of all potential failure scenarios.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:45:53 UTC from IEEE Xplore. Restrictions apply.

Limitation in integrality. Intermediate anomalies in the system are often difficult to capture, particularly during complex dynamic adjustments. Even if the final state of the system meets the expectations, it is challenging to determine whether any subtle errors or inconsistencies occurred during the adjustment process. Due to their subtlety and occasional nature, such anomalies are often difficult to detect. For these undetectable anomalies, we cannot predict whether they will have a minor or even latent long-term effect on the system.

Limitation in interaction and future work. Currently, we focus only on detecting and correcting the state of a single operator, without considering the complex interactions between multiple operators. In cloud-native environments, multiple operators manage resources, and their interactions can cause state conflicts, resource contention, or dependency issues. These anomalies often exceed the scope of single-operator mechanisms, making integrated detection and correction across operators a critical challenge. In future work, we plan to address interactions between multiple Kubernetes operators, aiming to enhance the robustness and reliability of cloud-native systems. Our goal is to provide a more comprehensive solution for Kubernetes operators in large-scale production environments.

VII. Conclusion

This paper introduces Arsc, a technique for generating, detecting, and correcting the state of Kubernetes clusters. The system state anomalies we detected can be categorized into three types: incorrect state, unhealthy state, and system crash. The causes of these anomalies include internal reconciliation issues and external disruptive errors. We propose three correction strategies to address these anomalies, depending on their underlying causes. The results show that Arsc demonstrates strong performance in maintaining state correctness in operator-based Kubernetes systems. Our future research goal is to address the state correctness issues in systems involving interactions between multiple operators, aiming to enhance the reliability and consistency of the system in more complex environments.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (Grant No. 62422208, 62202276, 62232010), in part by Shandong Science Fund for Excellent Young Scholars (No. 2022HWYQ-038), in part by Shandong Science Fund (No. 2023TSGC0105, 2024TSGC0228), in part by Ministry of Industry and Information Technology of China: Cloud-Data Integrated Machine for Cloud-Edge Collaboration.

References

[1] "Kubernetes." https://kubernetes.io/.
[2] "Swarm." https://docs.docker.com/engine/swarm/.
[3] "Apache mesos." https://mesos.apache.org/.
[4] "Operatorhub.io." https://operatorhub.io/.
[5] S. Engineering, "Fleet management at spotify (part 2): The path to declarative infrastructure." https://engineering.atspotify.com/2023/05/fleet-management-at-spotify-part-2-the-path-to-declarative-infrastructure/, 05 2023.
[6] Databricks, "Scaling apache spark on kubernetes at lyftli gao lyft,rohit menon lyft." https://www.youtube.com/watch?v=PPtrY_XxYBE.
[7] "Demystifying kubernetes as a service - how alibaba cloud manages 10,000s of kubernetes clusters." https://www.cncf.io/blog/2019/12/12/demystifying-kubernetes-as-a-service-how-does-alibaba-cloud-manage-10000s-of-kubernetes-clusters/, 12 2019.
[8] "Custom resources." https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/.
[9] "Controller." https://kubernetes.io/docs/concepts/architecture/controller/.
[10] X. Sun, W. Luo, J. T. Gu, A. Ganesan, R. Alagappan, M. Gasch, L. Suresh, and T. Xu, "Automatic reliability testing for cluster management controllers," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 143–159, 2022.
[11] B. Liu, G. Lim, R. Beckett, and P. B. Godfrey, "Kivi: Verification for cluster management," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pp. 509–527, 2024.
[12] J. T. Gu, X. Sun, W. Zhang, Y. Jiang, C. Wang, M. Vaziri, O. Legunsen, and T. Xu, "Acto: Automatic end-to-end testing for operation correctness of cloud system management," in *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 96–112, 2023.
[13] T. Hockin, "Edge vs. level triggered logic." https://speakerdeck.com/thockin/edge-vs-level-triggered-logic, 2017.
[14] X. Sun, W. Ma, J. T. Gu, Z. Ma, T. Chajed, J. Howell, A. Lattuada, O. Padon, L. Suresh, A. Szekeres, *et al.*, "Anvil: Verifying liveness of cluster management controllers," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 649–666, 2024.
[15] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich, "Verifying concurrent, crash-safe systems with perennial," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 243–258, 2019.
[16] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran, *et al.*, "Using lightweight formal methods to validate a key-value storage node in amazon s3," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pp. 836–850, 2021.
[17] T. Hance, A. Lattuada, C. Hawblitzel, J. Howell, R. Johnson, and B. Parno, "Storage systems are distributed systems (so verify them that way!)," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 99–115, 2020.
[18] N. Yaseen, B. Arzani, R. Beckett, S. Ciraci, and V. Liu, "Aragog: Scalable runtime verification of shardable networked systems," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 701–718, 2020.
[19] M. Zou, H. Ding, D. Du, M. Fu, R. Gu, and H. Chen, "Using concurrent relational logic with helpers for verifying the atomfs file system," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 259–274, 2019.
[20] H. Zhang, W. Honoré, N. Koh, Y. Li, Y. Li, L.-Y. Xia, L. Beringer, W. Mansky, B. Pierce, and S. Zdancewic, "Verifying an http key-value server with interaction trees and vst," in *The 12th Conference on Interactive Theorem Proving*, p. 32, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.
[21] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson, "Verdi: a framework for implementing and formally verifying distributed systems," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 357–368, 2015.
[22] R. Tao, J. Yao, X. Li, S.-W. Li, J. Nieh, and R. Gu, "Formal verification of a multiprocessor hypervisor on arm relaxed memory hardware," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pp. 866–881, 2021.
[23] H. Sigurbjarnarson, L. Nelson, B. Castro-Karney, J. Bornholt, E. Torlak, and X. Wang, "Nickel: A framework for design and verification of information flow control systems," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 287–305, 2018.
[24] V. Arun, M. T. Arashloo, A. Saeed, M. Alizadeh, and H. Balakrishnan, "Toward formally verifying congestion control behavior," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pp. 1–16, 2021.
[25] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pp. 155–168, 2017.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:45:53 UTC from IEEE Xplore. Restrictions apply.

[26] Y. Yuan, S.-J. Moon, S. Uppal, L. Jia, and V. Sekar, "{NetSMC}: A custom symbolic model checker for stateful network verification," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pp. 181–200, 2020.
[27] F. Yousefi, A. Abhashkumar, K. Subramanian, K. Hans, S. Ghorbani, and A. Akella, "Liveness verification of stateful network functions," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pp. 257–272, 2020.
[28] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark, "Kinetic: Verifiable dynamic network control," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 59–72, 2015.
[29] M. Yabandeh, N. Knežević, D. Kostić, and V. Kuncak, "Predicting and preventing inconsistencies in deployed distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 28, no. 1, pp. 1–49, 2010.
[30] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill, "Ironfleet: proving practical distributed systems correct," in *Proceedings of the 25th Symposium on Operating Systems Principles*, pp. 1–17, 2015.
[31] G. Turin, A. Borgarelli, S. Donetti, E. B. Johnsen, S. L. Tapia Tarifa, and F. Damiani, "A formal model of the kubernetes container framework," in *International Symposium on Leveraging Applications of Formal Methods*, pp. 558–577, Springer, 2020.
[32] B. Liu, A. Kheradmand, M. Caesar, and P. B. Godfrey, "Towards verified self-driving infrastructure," in *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pp. 96–102, 2020.
[33] H. Ding, Z. Wang, Z. Shen, R. Chen, and H. Chen, "Automated verification of idempotence for stateful serverless applications," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pp. 887–910, 2023.
[34] P. Godefroid, "Model checking for programming languages using verisoft," in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 174–186, 1997.
[35] J. Yang, C. Sar, and D. Engler, "Explode: a lightweight, general system for finding serious storage system errors," in *Proceedings of the 7th symposium on Operating systems design and implementation*, pp. 131–146, 2006.
[36] "How an operator becomes the hero of the edge." https://www.kubermatic.com/resources/how-an-operator-becomes-the-hero-of-the-edge/, 2021.
[37] C. Hall, "Aws, google, microsoft, red hat's new registry to act as clearing house for kubernetes operators." https://www.datacenterknowledge.com/open-source/aws-google-microsoft-red-hats-new-registry-act-clearing-house-kubernetes-operators, 03 2019.
[38] "Why operators are essential for kubernetes." https://www.redhat.com/en/blog/why-operators-are-essential-kubernetes, 2021.
[39] "Introducing the aws controllers for kubernetes (ack) — amazon web services." https://aws.amazon.com/cn/blogs/containers/aws-controllers-for-kubernetes-ack/, 08 2020.
[40] "Artifacthub." https://artifacthub.io/.
[41] "Official cve feed." https://kubernetes.io/docs/reference/issues-security/official-cve-feed/.
[42] "kind." https://kind.sigs.k8s.io/.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:45:53 UTC from IEEE Xplore. Restrictions apply.

AutoNet: Automatic Reachability Policy Management in Public Cloud Networks

German Sviridov Zheng Tao Shen Jorge Cardoso
Huawei Technologies

arXiv:2404.19372v1 [cs.NI] 30 Apr 2024

***Abstract*—Virtual Private Cloud (VPC) is the main network abstraction technology used in public cloud systems. VPCs are composed of a set of network services that permit the definition of complex network reachability properties among internal and external cloud entities such as tenants' VMs or some generic internet nodes. Although hiding the underlying complexity through a comprehensible abstraction layer, manually enforcing particular reachability intents in VPC networks is still notably error-prone and complex.**

In this paper, we propose AutoNet, a new model for assisting cloud tenants in managing reachability-based policies in VPC networks. AutoNet is capable of safely generating incremental VPC configurations while satisfying some metric-based high-level intent defined by the tenants. To achieve this goal, we leverage a MaxSAT-based encoding of the network configuration combined with several optimizations to scale to topologies with thousands of nodes. Our results show that the developed system is capable of achieving a sub-second response time for production VPC deployments while still providing fine-grained control over the generated configurations.

I. Introduction

Public cloud adoption has been observing exponential growth in recent years [1]. Thousands of organizations are either migrating their existing on-premise IT deployments to the cloud or are building their new solutions to be directly cloud-native [2]. As more diverse businesses started to move to the cloud, cloud providers started to provide more complex managed services in an attempt to satisfy the particular requirements of their ever-growing customer base. While many major cloud players started with a handful of general-purpose computing and storage services, today these services are in the hundreds and tailored for any possible use case a tenant may have. At the same time, most of services related to the management and operation of the fundamental cloud functions remained largely unchanged. Such is the case with cloud network connectivity management. Indeed, cloud networking still employs legacy network architectures composed of subnets, firewalls, routers, and other as-a-service counterparts of traditional network devices.

With the introduction of additional cloud-specific network services, operating cloud networks has reached a complexity high enough to require specialized roles within the enterprises just for such a task. Nevertheless, no matter the amount of available tools and expertise, most of the cloud network configuration changes still remain highly error-prone due to the tangled and complex nature of the cloud network services. Under some conditions, these configuration errors may potentially lead to catastrophic results for the enterprises. Indeed accidentally blocking some services or exposing sensitive information to the public network may result in multi-million losses in revenues [3] or in hefty fines in the case of severe data breaches [4], [5].

Major cloud providers have been trying to address issues related to accidental network misconfigurations by offering network reachability debugging tools [6], [7], [8] to their customers. While these tools are capable of assisting network engineers in finding potential network bugs they do so only after they have been introduced, as opposed to *proactively preventing and mitigating them* through controllable network configuration management. Thus, not providing tenants a way of reliably and controllably deploying network configuration updates within their cloud deployments.

In this paper, we tackle the requirement of proactive cloud network configuration management by proposing AutoNet. AutoNet is designed to assist cloud network engineers by providing a single network-wide abstraction for reachability policy management at the cloud tenants' level. The main goal of AutoNet is that of assisting tenants with incremental network updates by introducing the minimum set of configuration changes with respect to the existing tenant-defined configurations. To achieve this goal, AutoNet addresses four main requirements that are necessary to guarantee both the system's correctness and its practical usefulness in the public cloud:

- A simple abstraction model to define and manage reachability policies
- A second-level response time capable of supporting multiple configuration updates per minute
- Formal *network-wide* correctness guarantees of the network configuration changes
- Possibility of seamlessly tailoring configuration changes to meet specific high-level objectives such as deployment cost or service performance

To achieve these goals, AutoNet employs SAT-based [9] encoding of the entire cloud network. Additionally, to scale to large topologies, AutoNet makes use of different network compression techniques based on Binary Decision Diagram (BDD) [10]. Finally, AutoNet uses a weighted MaxSAT [11] formulation to allow the system to generate different network configurations based on user-specified functional intents and additional objectives such as configuration complexity, monetary cost, or service performance.

The rest of the paper is organized as follows: in Sec. II we describe key concepts behind public cloud networking infras-

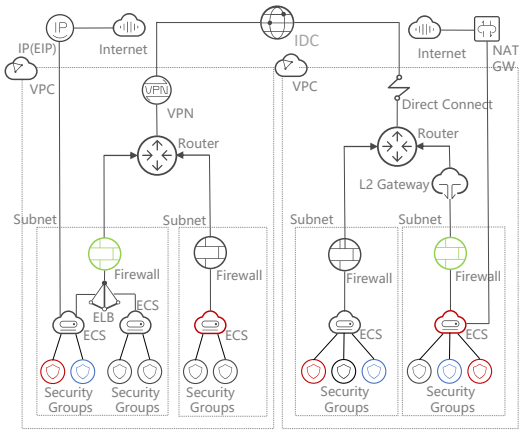


Fig. 1: Example of the most commong VPC services and their interconnect

tructure and motivate the importance of automatic reachability policy management. We then present the overall architecture of AutoNet ands describe its core components in Sec. III. We evaluate our system in Sec. IV and describe how it compares to existing work in Sec. V. Finally, we discuss limitations of our proposal and possible future work in Sec. VI, and draw our conclusions in Sec. VII.

II. Public Cloud Networking

Modern public cloud deployments are composed of a broad set of managed services such as storage, computing, and networking services. Nowadays there exist multiple different flavors of storage and elastic computing services (ECSs) that keep evolving based on the customers' requirements. Despite the vast service offering, management simplicity remains among the main objectives cloud providers have when developing new managed services for public customers. Indeed, users are typically exposed to a simplified interface that permits them to easily request new resources and seamlessly interconnect them. Such a level of abstraction hides most of the underlying complexity away from the users while providing only basic building blocks for building complex and intricate network interconnects for their applications.

When it comes to cloud networking, the majority of intra-cloud connectivity services are offered within the Virtual Private Cloud (VPC) macro-service. VPCs include the virtualized counterparts of the traditional network devices such as routers or firewalls, as depicted in Fig. 1. Their main role is that of providing connectivity between different ECSs, as well as, providing access control, load balancing, and other traffic and security-related features.

A. Virtual Private Cloud architecture

The main difference that separates VPCs from normal networks is that each service can be located only at designated locations within the network topology. To do so, VPCs are built in a structured and hierarchical way as depicted in Fig. 1. ECSs are aggregated into logical groups within a VPC using virtual subnets and are connected through one or more virtual NIC. Finally, multiple subnets are grouped into logical VPCs with an N:1 mapping among them. In addition to traditional services, VPCs may include direct connect network gateways which may provide load balancing through elastic load balancers (ELBs), connectivity to external networks, to other VPCs within the cloud infrastructure, or to the internet network through Elastic IPs (EIPs) or NAT gateways (NAT-GWs).

To provide traffic control over the traffic flows, VPC services are typically associated with one or more routing or filtering service. These services are configured with a corresponding configuration that defines, e.g., a routing table for routers, a remote destination for VPC Peerings, a public IP address for EIP, or ACL rules for network ACLs (NACLs) and security groups (SGs), etc. The entirety of the configurations of all of the VPC network services defines the overall VPC *network reachability policy*. A reachability policy is a mapping between all pairs of VPC services representing the traffic source and the traffic destination and a corresponding header space representing the allowed traffic between the two VPC services. Although limited, the exposed set of configurations within the VPC macro-service can still lead to an explosion in complexity when customers' topologies grow larger, in the presence of non-out-of-the-box deployments (e.g., hybrid or multi-cloud deployments), or simply when the existing network services are not properly managed. Indeed, differently from the mapping of VPC services, individual configurations can be assigned to multiple service instances. This relation implies the possibility of the following scenarios manifesting:

- **Replicated configurations:** a single configuration can be associated with multiple distinct network services. This implies that, e.g., the same set of ACL rules can reside within two different NACLs. Thus changing the configuration of one network service may affect the configuration of another network service. Fig. 1 shows a possible instance of this behavior with the first and last subnets sharing a green firewall.
- **Tangled configurations:** multiple network services can have more than one associated configuration which are treated as a single configuration according to some configuration merging mechanism. E.g., a single router can be associated with a list of different routing tables, or a single subnet can be associated with multiple NACLs that are applied to the transient traffic according to their priority. Fig. 1 depicts such possibility with first and last ECSs sharing a red and blue SGs.
- **Replicated network addresses:** two or more VPC network services can share the same network address space. In Fig. 1 a possible example of this scenario could be the two red ECSs, with one being located on an on-premise deployment and the other being deployed inside the cloud data center.

The aforementioned three properties imply that *a change to a potentially localized configuration can have a broader*

effect on the entire network if the network configuration is not managed properly. This, in turn, makes it even more difficult to verify or enforce desired reachability intents without introducing misconfigurations and potentially compromising the correctness of the entire network.

B. Reachability policy management in VPC networks

Most cloud customers nowadays spend considerable amount of resources towards continuous assurance of reachability policy intents within networks. Such reachability policies may include some intents related to the security of the services, such as access to some sensitive storage from the public networks, or to some functional intents such as guaranteeing connectivity between the front-end services and the back-end databases. In Tbl. I we summarize some of the most common reachability policy intents that customers typically want to implement within their deployments.

1) The difficulty of verifying reachability properties:: Most of the reachability policy intent assurance systems still employ traditional network probing [12] capable of only providing weak guarantees about the network behavior. While such approaches can lead to meaningful coverage in small deployments, due to the ever-growing complexity and scale of modern VPCs, continuously and provably assuring full test coverage over the entire network keeps becoming more and more challenging [13]. To tackle this problem, major cloud providers have been offering VPC reachability verification tools [6], [7], [14], [8] capable of verifying whether a given reachability intent is satisfied or not within the tenants' deployment. Yet, while addressing the original problem, whenever a misconfiguration or a policy violation is found, customers are still required to manually act to fix the problem. This may lead to an iterative and tedious trial and error approach which may require up to hours of manual labor for large topologies [15] with complex intents [1]. When considering that cloud network configurations can be operated on up to several times per hour, even a small degree of misconfiguration probability may lead to the emergence of severe bugs [16] that are notoriously difficult to detect and may persist over a prolonged amount of time.

2) The necessity of satisfying high-level objectives:: While a network configuration can be functionally correct by itself and satisfy all functional requirements in terms of desired reachability properties, it may still not satisfy some high-level cost or performance metric. Due to the presence of multiple different network services, in most of the existing public cloud providers, there may be multiple alternatives to satisfy the same functional objective. Although being functionally equivalent, each alternative can lead to different properties in terms cost, security, or performance. Indeed, connecting the front-end server to the in-house database through a direct connect would lead to the highest possible service performance and security while incurring a substantial cost for the business.

[1]Estimation based on the measured average response time of commercial network verification services

TABLE I: Example reachability policy intents to be enforced in a commercial cloud deployment

Intent	Intent description
I1 - Simple Reachability	Some traffic type is allowed or denied between two ECSs
I2 - Internet Isolation	Some traffic type is allowed or denied between an ECS and some internet node
I3 - Subnet Isolation	Some traffic type is allowed or denied between two subnets
I4 - Subnet Internet Isolation	Some traffic type is allowed or denied between a subnet and some internet node
I5 - Waypointing	Force some 5-tuple traffic between two ECSs to traverse a third ECS

On the other hand, letting the traffic transit through the public internet instead will lead to the opposite effect. The choice among these alternatives, alongside to functional requirements, is usually left to the cloud network engineer to satisfy. In such a complex setting guaranteeing that a particular change is both correct and optimal in terms of some metric becomes virtually impossible without specialized tools. To the best of our knowledge, no cloud provider has been able to provide an effective system capable of jointly enforcing and verifying a given functional requirement while satisfying additional high-level metrics in a fully automated and transparent way.

III. AutoNet: Automatic reachability policy management for VPC networks

AutoNet aims at providing a scalable cloud network management tool capable of transparently enforcing new reachability policies at a second-level scale for large VPC deployments. At the same time, AutoNet aims at preserving the properties of the existing reachability policies while also satisfying high-level configuration objectives defined by the operators.

The properties AutoNet tries to achieve have been shown to be an EXPSPACE-complete and NP-complete problem in the general case [17], [18]. To tackle this complexity and to achieve our functional and performance goals we employ a set of suitably tailored techniques that permit to reduce the dimensionality of the problem, and at the same time safely and efficiently generate updated reachability policies based on user-defined intents. Most notably, AutoNet is based on the following core components that are later discussed in detail:

- **Definition of reachability intent (Sec. III-A)**: Provides a way for the user to define a reachability policy intent among the one present in Tbl. I as well as configuration weights used to steer AutoNet towards more desirable solutions
- **Network topology abstraction (Sec. III-B)**: Provides a common abstraction for all of the complex network services present in the cloud deployments and permits to

reduce the entire VPC topology to a simple to manage tree-based topology

- **Network pre-processing (Sec. III-C)**: Allows to reduce the dimensionality of the problem by pruning part of the services present in the cloud deployment and/or by compressing their configurations
- **SAT-based actionable network reachability encoding (Sec. III-D)**: Allows to formally encode reachability properties among different services in the cloud deployment by reasoning about the topological properties of the VPC deployment as opposed to path-based based techniques
- **MaxSAT-based configuration update synthesis (Sec. III-E)**: Allows to compute the minimum set of configurations to be changed to enforce some particular reachability policy while satisfying users' intents in terms of desired solution space

A. Definition of reachability intent

Similarly to the existing tools [6], [7], AutoNet requires users to specify the desired reachability intent. In AutoNet we define an intent as a particular desired reachability policy composed of i) a desired traffic action such as allow or deny, ii) an intent header space (IHS) which represents the header space for which the intent must be satisfied, iii) a set of source and destination nodes between which the intent must hold, and, optionally, iv) a set of configuration weights that permit the user to steer AutoNet towards the generation of some specific configurations (e.g., preferring the changes to the configuration of existing network services as opposed to the generation of new ones).

B. Network topology abstraction

Cloud deployments may include a multitude of different complex services that provide networking capabilities. Nevertheless, it can be noticed that almost all of these services can be reduced to a small set of abstract services. Indeed, these services can be broadly classified into three main abstract network services:

- **Filtering service**: defined as network services capable of allowing or denying (i.e., filtering) at least a part of the traffic that traverses them (e.g., NACLs, SGs, firewalls). The associated configuration is composed of a list of rules that map some header space to an allow/deny action.
- **Routing service**: defined as network services capable of altering the path of at least a part of the traffic that traverses them (e.g., routers, ELBs, transit gateways, etc.). The associated configuration is composed of a list of rules which map some header space to a next-hop network service.
- **End devices**: defined as network services that act as one of the endpoints for a traffic flow. Such services can be either a source or a destination of a particular traffic flow (e.g., ECSs, BMSs). The associated configuration is typically composed of at least an associated unique ID, but in most cases includes also an associated IP address.

We noticed that this set of basic primitive services is sufficient to model most of the network services present in a cloud deployment and we use this abstraction in AutoNet to simplify the modeling and analysis of the multitude of network services present in commercial cloud providers by representing each of them as a combination of the three primitive classes.

Additionally, by observing Fig. 1, it can be seen that, apart from a handful of network services, most of the services within a VPC are connected in a tree topology. To achieve a full tree topology move the out-of-place network services to an ad-hoc created VPC while also installing an equivalent set of immutable rules on all of the intermediate nodes to provide reachability equivalence to the original topology.

C. Network pre-processing

While the network topology abstraction permits us to make the problem more approachable, it does not help in reducing the algorithmic complexity. To tackle this problem we employ two network pre-processing techniques, namely *topology pruning*, and *configuration quantization*. The main goal of these techniques is to filter out and remove some of the abstract services from the network topology or to at least simplify their configuration.

1) Topology pruning: aims at reducing the total amount of network services present in the final model of the network. For this purpose, all of the endpoints which do not lead to source or destination network address space overlap with the specified IHS are removed from the topology. The rationale behind this processing step is that, even if the IHS traffic between the filtered-out end devices is affected by the configuration change, the overall reachability policy will remain invariant due to the default traffic filtering at end devices. Exceptions to the aforementioned scheme (e.g., EIPs that provide connectivity to the entirety of public IPs) exist and are managed accordingly by not applying any pruning.

2) Configuration quantization: aims at quantifying the behavior of a network service in response to the IHS. For the purpose of quantization, service configurations are translated into their corresponding BDD representation. Similarly to [14], for both filtering and routing services, the quantization is performed by iteratively applying a sequence of boolean operations on the BDD representation of individual configuration lines. For network filters, quantization implies querying the associated configuration for the response in case of an incoming/outgoing IHS flow. Such a response can either be an "allow", a "deny" or a partial result. The latter implies that only a part of the IHS is allowed to flow through the network filter. In such cases, the result is substituted with the opposite of the desired reachability property (e.g., if the desired reachability property is "allow" the partial quantified result is substituted with "deny"). The rationale behind this is that we want the entirety of the IHS defined in the intent to satisfy the desired reachability property. After the quantization has been performed each configuration file associated with network filters is substituted with the quantified filtering result (i.e., "allow" or "deny"). For network routers configuration

quantization involves extracting the set of possible reachable devices for the IHS defined within the associated configuration. The possible reachable devices are extracted by considering the next hop field within the associated configuration. After the quantization has been performed each configuration file associated with network routers is substituted with the extracted set.

D. Network encoding

The network encoding step involves the translation of the quantified and sampled network into an SAT-based representation of the entire network reachability policy. The resulting encoding will be used later by our MaxSAT to enforce reachability intents. In the following we provide a simplified formulation of the aforementioned constraints using the developed model and discuss the scalability of the proposed approach. For the purpose of the formulation, we assume to have an unquantified and un-quantized 3-layer tree topology. We also assume that all nodes have a single configuration as opposed to having a distinct set of associated ingress and egress configurations.

All configurations $f \in F$ present in the abstracted network are associated with a set of header spaces that summarize their behavior. Such header spaces are defined in terms of explicitly allowed, denied, and matched header space, namely $c_f^{(A)}$, $c_f^{(D)}$, $c_f^{(M)}$. Since a single node can be associated with multiple configurations we need to define a suitable node configuration encoding function $c(f_1, .., f_n)$ which translates all of the configurations associated with a node to their synthetic representation. Such a function is defined recursively as:

$$c(f_1, .., f_n) = c_{f_1}^{(M)} \rightarrow (c_{f_1}^{(A)} \wedge \neg c_{f_1}^{(D)}) \wedge (\neg c_{f_1}^{(M)} \rightarrow c(f_2, .., f_n)) \quad (1)$$

The equation Eq. (1) encodes the multi-level matching of different configurations assuming a decreasing matching priority (i.e., f_1 has the highest priority). Following this definition, we can define the total admissible header space over a generic node $n \in N$ with of all of the associated configurations $F_n \in F$ as $R_n = c(F_n)$.

Using $P : N \rightarrow N$ as a function that maps each node to the corresponding parent node, the total admissible traffic between two nodes $s, d \in N$ can be expressed as follows:

$$R(s,d) = (R_s \wedge R_d) \wedge (\neg(P(s) = P(d)) \rightarrow (R_{P(s)} \wedge R_{P(d)})) \wedge (\neg(P(P(s)) = P(P(d))) \rightarrow (R_{P(P(s))} \wedge R_{P(P(d))})) \quad , \forall s \in N, \forall d \in N \quad (2)$$

The key peculiarity of Eq. (2) is that the entire encoding can be performed in $O(N)$. Indeed, Eq. (2) can be easily rewritten as a series of hierarchical clauses, thus not requiring, neither enumerating all of the possible paths a flow can take nor building $O(N^2)$ equations for all of the possible combinations of source and destination pairs.

Our goal is to devise an algorithm capable of altering the above formulation in order to achieve the desired target reachability policy for a particular set of $s, d \in N$. To do so, we need a way of altering the previously formulated expressions to include a set of decision variables. These decision variables allow us to bypass the normal expression for the admissible traffic on any given configuration or network device. For network configurations, this is done by encoding the condition *"if the decision variable is set to some header space then return the set header space, otherwise return the normal admissible header space"*. If we want to support both allow and deny queries we have to introduce two types of decision variables for each network service and each configuration. The new expression becomes a ternary expression of the form *"if the allow decision variable is set to some header space then return the header space, else if it the deny decision variable is set to some header space, return the negated header space, otherwise return the normal admissible header space"*. For this reason, we introduce an *actionable equivalent* expression of Eq. (1) $a(f_1, .., f_n)$ defined as:

$$a(f_1, .., f_n) = ((\phi_{f_1}^{(A)} \vee \phi_{f_1}^{(D)}) \rightarrow (\phi_{f_1}^{(A)} \wedge \neg\phi_{f_1}^{(D)})) \vee c_{f_1}^{(M)} \rightarrow (c_{f_1}^{(A)} \wedge \neg c_{f_1}^{(D)}) \wedge (\neg c_{f_1}^{(M)} \rightarrow a(f_2, .., f_n)) \quad (3)$$

$\phi_{f_i}^{(D)}$ and $\phi_{f_i}^{(A)}$ represent respectively the decision variables that, once assigned, permit to respectively add an explicit deny or allow for the IHS within a set of configurations f_i associated with a single node[2].

Nevertheless, in some cases, there may exist scenarios that would make the problem unsolvable without adding additional configurations to the network (e.g., missing EIP for internet access). For this reason, we introduce an additional set of node ternary decision variables which encode the possibility of generating new configurations (e.g., NACLs, EIPs, routing tables, etc) from scratch as:

$$A_n = ((\psi_n^{(A)} \vee \psi_n^{(D)}) \rightarrow (\psi_n^{(A)} \wedge \neg\psi_n^{(D)})) \vee a(F_n) \quad (4)$$

Eq. (4) encodes the aforementioned requirement for each node n by permitting the assignment of some values to the decisional variables $\psi_n^{(\cdot)}$ in a similar way to Eq. (3). Given the aforementioned expression, we can define the *actionable configuration expression* $A(s, d, \Phi^{(A)}, \Phi^{(D)}, \Psi^{(A)}, \Psi^{(D)})$ in an equivalent way to Eq. (2).

Finally, given a desired reachability intent expressed in the form of a set of source and destination nodes $N^{(\texttt{IHS,SRC})}, N^{(\texttt{IHS,DST})}$, and a particular IHS $h^{\texttt{IHS}}$, we can formalize the reachability intent satisfaction and the reachability policy conservation constraints respectively as:

[2]For the sake of simplicity we omit the decision variables for the creation of new routing entries

$$\exists \Phi^{(A)},\ \Phi^{(D)},\ \Psi^{(A)},\ \Psi^{(D)} | \\ A(s,d,\Phi^{(A)},\Phi^{(D)},\Psi^{(A)},\Psi^{(D)}) = R(s,d) \vee h^{\texttt{IHS}} \\ ,\forall s \in N^{(\texttt{IHS,SRC})},\ \forall d \in N^{(\texttt{IHS,DST})} \quad (5)$$

$$\exists \Phi^{(A)},\Phi^{(D)},\Psi^{(A)},Psi^{(D)} | \\ A(s,d,\Phi^{(A)},\Phi^{(D)},\Psi^{(A)},\Psi^{(D)}) = R(s,d) \\ ,\forall s \neq d \in N \setminus (N^{(\texttt{IHS,SRC})} \cup N^{(\texttt{IHS,DST})}) \quad (6)$$

Eq. (5) guarantees that the desired reachability intent will be enforced while Eq. (6) guarantees that at the same time, no accidental changes to the reachability properties of unrelated network elements will be introduced. Satisfying these constraints requires finding a suitable assignment for all services in $\Phi^{(A)},\Phi^{(D)},\Psi^{(A)},\Psi^{(D)}$. Such an assignment will guarantee that only the reachability policy between $N^{(\texttt{IHS,SRC})}$ and $N^{(\texttt{IHS,DST})}$ will be affected while all of the other end-to-end reachability policies will remain unaffected.

Throughout the formulation, we use only two decision variables per configuration element which is enough to cover all reachability scenarios involving intra-cloud communication. Indeed, a single decision variable can model a single header space and since intra-cloud communication does not involve packet header overwriting each configuration will require at most two rules to be added/changed to enforce a given reachability property (represented by $\phi_{f_i}^{(A)},\ \phi_{f_i}^{(D)}$). For inter-cloud communication we jointly solve multiple instances of the same problem with different network quantization applied to the network based on the actual header space transformation performed by network services.

E. Configuration generation

To find a suitable solution to the encoded problem we can formalize the problem as a minimization problem. We want to minimize the number of decision variables set to a value different from the default value (i.e., $\emptyset$). We can formulate this requirement as follows:

$$\min \left(\sum \omega_i \left(\phi_i \neq \emptyset \right) \right) \forall \phi_i \in (\Phi^{(A)} \cup \Phi^{(D)} \cup\ \Psi^{(A)} \cup\ \Psi^{(D)}) \quad (7)$$

Solving this objective function will lead to the minimum assignment for all of the decision variables. To solve the constrained objective function, we use a MaxSAT formulation and solve it using a weighted implementation of the PmRes algorithm from [19]. The formulation treats AutoNet's decision variables as weighted soft constraints and uses a core-guided approach to find the minimum weighted set of unsatisfiable soft constraints that will satisfy the hard constraints in Eq. (5) and Eq. (6).

1) Objective-aware configuration updates: Weight factors ω_i permit us to steer the solution algorithm towards desired configurations. This permits to prioritize configurations that, e.g., introduce less explicit deny rules (larger weights for $\phi \in \Phi^{(D)}$) or avoid blackhole routing rules (larger weights for $\phi_f \in \left(\Phi^{(A)} \cup \Phi^{(D)}\right),\ \ \forall f \in P(s),\ \forall s \in N_e$). At the same time, a suitable assignment to the weight factors can encode high level-objectives such as monetary cost. As an example, one can require to enforce reachability between all hosts within a subnet and an external IP address. By using ω_i proportional to the hourly cost of an EIP and NAT-GW services AutoNet will be able to automatically find which service is more convenient to deploy under the current scenario. Similarly, one can assign ω_i based on the average amount of bandwidth multiplied by the price per GB of transferred data flowing between on-premise deployment and the public cloud. AutoNet will be capable of selecting the best solution among, e.g., VPN, EIP, or Direct Connect, to be used to satisfy the requirements while minimizing the price of the intent. Finally, under the same scenario, ω_i can express the security requirements of a reachability intent. Indeed, high-security requirements may limit the aforementioned choice to VPN and Direct Connect only as they guarantee that the traffic will either be encrypted or will flow exclusively between two secured endpoints.

2) Translation to VPC configurations: The outcome of the MaxSAT algorithm is a suitable assignment to the decision variables. Yet, these decision variables need to be synthesized into actual configurations to be applied to the network services. For this reason, we use a specific configuration translation engine capable of mapping each assignment into a corresponding change in a configuration of a particular network service. The actual deployment of the configuration changes depends on the specific network service. For simple filtering or routing services, an additional allow/deny rule or a routing table entry for the IHS is added in the top position of the affected configurations and eventual anomalies are removed [20]. This technique is guaranteed to lead to a correct final configuration as it mimics exactly the behavior formalized in the actionable node configuration encoding function equation in Eq. (3). In the case of the generation of new network services, such as in the case of a creation of a new EIP or ELB, the procedure will follow a the normal flow for the assignment of the configuration parameters of each service while requiring minimum input from the tenant (e.g., assigning metadata to new services). Finally, the generated configuration change can then be directly applied to the network following standard approaches such as cloud SDK or through the cloud management web console.

IV. Experimental evaluation

In this Section, we describe the implementation details and the evaluation of AutoNet with both synthetic and real data. The take-home messages of our evaluation can be briefly summarized in the following:

- AutoNet can perform reachability policy enforcements in under 10 seconds for 2K involved ECSs on top of a commodity machine.
- Total running time of AutoNet scales linearly with the number of ECSs involved in the reachability policy enforcement.
- The time required to perform SAT encoding of the cloud deployment increases linearly with the number of involved subnets and VPCs.
- AutoNet can enforce all of the reachability intents from Tbl. I in under 250ms for medium-to-large size cloud deployments

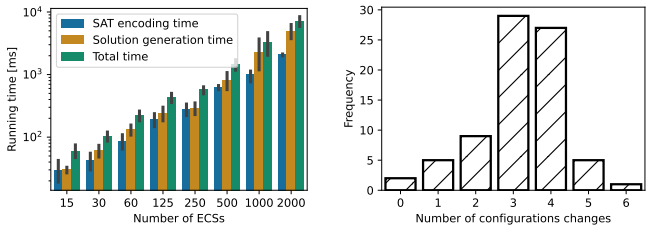
A. Implementation

We implemented AutoNet on top of Batfish [14] as a separate module with 6K lines of Java code. To be able to parse cloud configuration files and translate them to a vendor-independent representation we also extend the Batfish parser with extra logic. Finally, we extend the JavaBDD library used by Batfish to support multithreading and accelerate the network-preprocessing step of AutoNet. To perform the network encoding and generate final solutions we use Z3 [21] SMT solver version 4.8.12.

We evaluated AutoNet by running the system on top of a cloud compute instance equipped with 4 virtual CPUs and 8GB of RAM. The entirety of AutoNet runs as a single-threaded application. For our evaluation, we consider synthetically generated VPC configurations as well as some snapshots of real configurations of internal customers. The synthetic topology generator accepts as inputs the number of ECSs, total number of subnets, total number of VPCs, total number of unique SGs, NACLs, and routers, and inter-VPC peering probability. Additionally, to simulate tangled configurations the generator accepts the average number of SGs per ECS. The generator will randomly generate VPC topologies by assigning each ECS to a random subnet, and each subnet to a random VPC. A similar procedure is done for filtering and routing services.

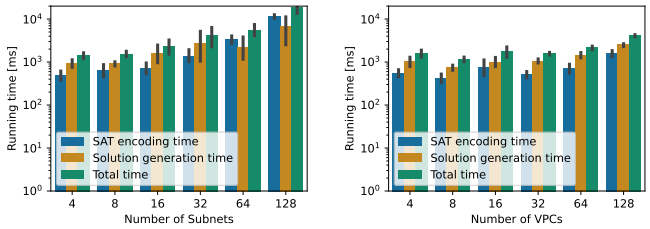
B. Scalability analysis

We evaluate the scalability of our system by considering random topologies of a varying number of ECSs. We fix the number of VPCs equal to 4, the number of subnets, NACLs and routers equal to 10, and the average number of SGs per ECS equal to 10. Unless, explicitly stated we use unitary weights for all of the possible decision variables a part from the generation of new network services which are set to a weight of 10. We consider two random ECSs e_s, e_d within the generated topologies and define the reachability policy to be enforced as *"allow all traffic originating from e_s and destined to any other host to also reach e_d"* (i.e., all of the destination fields of the reachability header space are set to wildcard). Under such reachability intent, the topology pruning will not be able to filter out any node from the topology due to the presence of a wildcard destination header space, thus forcing the SAT encoding to encode the entire network. We run all of the experiments 10 times and analyze the breakdown of the running time by considering the time required to generate the SAT encoding of the network and the time to generate the solution in terms of actual configurations to change. Fig. 2a shows that even for topologies with 2000 ECSs the total running time stays below 10 seconds. The majority of the time is spent in trying to find a satisfiable solution with the minimum cost, while the time required to perform SAT encoding of the network stays in the order of magnitude of 1 second. For the same experiments Fig. 2b depicts the distribution of the number of changes required to enforce the desired reachability policy. Most of the instances require between 3 to 5 unique configuration changes with some of the instances requiring up to 7 unique configuration changes. In all cases, the majority of these configuration changes relate to a simple addition of new ACL or routing line within existing SG/NACLs and routing tables.



(a) Breakdown of AutoNet running time (b) Distribution of the amount of configuration changes

Fig. 2: Scalability of AutoNet for reachability policy enforcement in a random topology for varying number of ECSs



(a) Variable number of subnets (b) Variable number of VPCs

Fig. 3: Scalability of AutoNet for reachability policy enforcement in a random topology for varying number of subnets and VPCs

Using the same setting from the previous experiments we now fix the total number of ECSs to 500 and assess the scalability of AutoNet in function of the number of subnets and VPCs. Fig. 3a depicts the running time breakdown for the scenario with a varying number of subnets. As the amount of subnets grows larger AutoNet takes more time to perform the SAT encoding of the network. Notably, for 128 unique subnets the encoding time approaches the 10 seconds mark. At the same time, the total time required to find the solution to the policy enforcement problem does not increase with the number

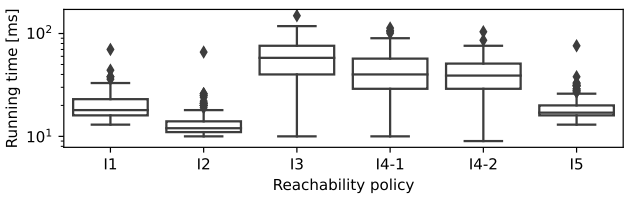


Fig. 4: Running times for intents in Tbl. I for synthetic topologies. I4-1 and I4-2 represent the exclusive use of, respectively, EIPs and NAT-GW.

of subnets. This happens due to the fact that each subnet does not contribute individually to the increase in the number of decision variables. Indeed, an increased number of subnets only contributes to the number of expressions of degree 3 generated in Eq. (2). We next performed results to assess the scalability of AutoNet for a variable number of VPCs. For this purpose we set the number of ECSs to 500 and the number of subnets equal to the number of VPCs. The breakdown of the running times under such a scenario is depicted in Fig. 3b. Similarly to the scenario with a variable number of subnets the total running time increases. Yet, under such a scenario, the network encoding contributes less to the total running time.

For both the experiments with variable VPCs and subnet sizes results similar to the variable ECSs case have been obtained for the distribution of the number of configuration changes and are omitted.

C. Intent performance analysis

We now consider a more realistic analysis in which we use a more strict reachability enforcement intent and we consider more different reachability intents. Notably, we evaluate AutoNet for the intents from Tbl. I and report our findings in Fig. 4. We consider both allow and deny counterparts of various intents. Yet, due to the way random topologies are generated most of the flows are denied by default, thus we report only the results for the allow case. Fig. 4 shows that for simple allow between two ECSs (I1), the running time stays just above 20ms. Similarly, enforcing reachability between a single ECS and a generic internet node (I2) the total required time stays around 14ms. This behavior is mainly due to the fact that fewer portions of the network need to be analyzed under such a scenario as most of the time this operation only requires a change to a single security group and the creation of a new EIP. The running times increase when it comes to more broader intents such as subnet-to-subnet reachability enforcement (I3). Under such scenario, the average number of actions required is 6.15, as opposed to the I1 and I2 which require 3.69 and 2.11 actions respectively. Additionally, we evaluate the effect of the weights in Eq. (7) by considering subnet-to-internet reachability enforcement intents (I4-1, I4-2). For this scenario, we consider two sets of distinct weights. For I4-1 we set the cost of adding a new NAT-GW to infinity, thus forcing the solver to assign individual EIPs to each ECS belonging to a subnet. On the contrary, in I4-2 we perform the opposite and force the solver to only use NAT-GW. The two solutions present a similar performance although the number of required actions varies greatly (8.10, and 5.85 respectively). Finally, we evaluate the performance of AutoNet for the waypoint enforcement intent (I5). Under this scenario the solver behaves similarly to I1, as the overall complexity of the solution is similar, leading to an average running time of 19.71ms.

D. Evaluation with a real deployment

In addition to the synthetic VPC topologies, we were able to evaluate AutoNet on top of a medium-size real cloud deployment that is used by an internal customer. The deployment includes applications related to big data processing, rendering and front/back-end web services running on top of 300-500 ECSs, 30-50 VPCs 10+ NAT-GW services, 10+ ELBs, and 50+ EIPs. Most of the access control is managed by SGs with 80+ distinct SG configurations, with the presence of replicated network addresses, as well as, a medium degree of replicated and tangled configurations.

We perform the same experiments performed in Sec. IV-C while additionally plotting also the time required to perform the network pre-processing (i.e., quantization and filtering). We consider a random sampling of source and destination nodes for all intents and repeat each run 1000 times. Fig. 5 shows the results in terms of running time for all of the considered reachability policies. The main difference with respect to the synthetic case is that I3-I5 exhibit a considerably higher running time. This increase in running time can be attributed to the overall increased complexity of the considered realistic configurations. Indeed, the main contributor to this increase is the time spent to find the solution for the MaxSAT problem. Although, due to data confidentiality, we cannot provide details regarding the distribution of the number of required configuration changes, we can say that in some cases AutoNet was forced to apply 10+ different configuration changes to satisfy some reachability intents. Such scenarios reflect in Fig. 5 with an increased variance for some of the results.

These results, combined with results from our scalability analysis, show that AutoNet is a highly promising approach for scalable and fast reachability policy enforcement for medium-to-large scale cloud deployments.

V. Related Work

Numerous works in the field of network verification [22], [23], [24], [25], [26], [27], [28], [14], [29] tackled the problem of scalability and correctness of data plane verification. For this reason, multiple techniques such as BDD [14], set algebra [22], and SAT-based encodings [29], have been employed. These techniques target the reduction of the complexity of the problem by efficiently summarizing complex header spaces and their interactions with the network services. Nevertheless, few attempts have been made in the direction of *incremental data plane configuration synthesis* as opposed to simple configuration verification.

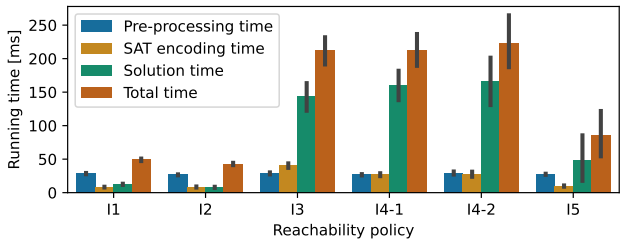


Fig. 5: Running times for intents in Tbl. I for a real VPC deployment.

Perhaps the most relevant works to AutoNet are [30], and [31]. Both systems propose an SMT-based framework for debugging reachability property violations within a VPC network. The system in [30] is capable of extracting the minimum configuration set that is required to be changed in order to enforce some desired connectivity property. Yet, while it can provide suggestions on how to fix a potential misconfiguration, it neglects any effect of the fix on the rest of the network. Thus, leading to the possibility of introducing security policy violations and/or service interruptions if applied blindly. [31] follows a similar approach, yet it targets SDN data planes, thus not taking full advantage of the topological properties of VPC networks, while at the same time not supporting replicated and tangled configuration, and any complex data plane service beyond routers and firewalls.

The authors of Jinjing [32] make use of SAT-based network encoding to reason about different ACL-related properties within a network. Notably, Jinjing is capable of synthesizing ACL configurations following a particular user specification. Nevertheless, the system is only capable of reasoning about ACLs and does not consider more complex network services such as routers, load balancers, NAT-GW, etc. Additionally, Jinjing can only change existing configurations as opposed to AutoNet that, when needed, can generate entirely new network services within the cloud deployment.

NetComplete [33], Propane/AT [34], NetEgg [35], AED [36] are capable of synthesizing network configurations from scratch, yet they targets mainly control plane protocols.

Finally, in addition to individual limitations, none of the aforementioned techniques can take into account the user-specified cost function used by AutoNet over which to perform the optimization process.

VI. Discussion and Future Work

While we have evaluated AutoNet on production data showing its versatility and scalability there are still challenges that remain unaddressed.

Inter-vpc hopping: Perhaps the biggest limitation of AutoNet is the absence of support for inter-VPC hopping. Indeed, AutoNet is not capable of reasoning about traffic scenarios in which a flow would hop between multiple VPCs before landing on the destination VPC. This limitation will not only introduce false positives as in the previous two cases but may also affect the safety of the system. Nevertheless, this behavior does not affect the correctness of the reachability policy enforcement and only requires a slight change to the SAT formulation to restore safety. This point, alongside the previously discussed points related to the SAT formulation, is an active part of our ongoing work.

Service coverage: Additionally, AutoNet is not complete in terms of the supported network services. Notably, we did not include hybrid cloud network services such as Enterprise Switch and Enterprise Routerin AutoNet. This choice was made due to their novelty at the time of the development of AutoNet and not due to the technological limitations adopted by AutoNet. Indeed, both services can be easily reduced down to a common abstraction used by AutoNet and encoded within the formulation. We plan to support them in the next releases of AutoNet in the near future.

Comprehensive objective definition engine: Throughout the paper, we discussed the user-defined objective weights to be used within the MaxSAT solution. While being sound from the theoretical point of view, from the practical point of view a customer-facing technology that requires the definition of such many parameters may be burdensome and confusing. In the future, we plan to provide an automatic intent-based weight assignment engine capable of synthesizing a suitable set of weights from high-level intents (e.g., minimizing the overall cost of the deployment, minimizing the configuration complexity, etc.). A suitable abstraction level for this subsystem is still under discussion and is planned for future releases.

VII. Conclusion

In this paper, we introduce AutoNet, a system for automatic network reachability management in cloud networks. AutoNet provides a comprehensive abstraction that hides all of the complexity of managed cloud network services behind an simple abstraction while achieving sub-second response time. At its core, AutoNet uses network pre-filtering and quantization to reduce the complexity of the reachability enforcement problem. At the same time, it uses SAT-based encoding of the network combined with topological properties of the network to achieve soundness. Our experiments show that, even for large VPC deployments, AutoNet is able to enforce the desired reachability policies in under 10 seconds, while at the same time, for more realistic intents the running time of AutoNet stays below 250ms. We argue that AutoNet can be seen as a full-fledged alternative to traditional cloud network abstractions and represents a step further towards fully autonomous cloud networks operation.

References

[1] "Gartner forecasts worldwide public cloud end-user spending to reach nearly $600 billion in 2023." [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2022-10-31-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-nearly-600-billion-in-2023

[2] N. Kratzke and P.-C. Quint, "Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study," *Journal of Systems and Software*, vol. 126, pp. 1–16, 2017.

[3] "The impact of exploitable misconfigurations on network security." [Online]. Available: https://www.titania.com/about-us/news-media/new-report-reveals-exploitable-network-misconfigurations-cost-organizations-9-percent-of-total-annual-revenue/

[4] C. J. Hoofnagle, B. Van Der Sloot, and F. Z. Borgesius, "The european union general data protection regulation: what it is and what it means," *Information & Communications Technology Law*, vol. 28, no. 1, pp. 65–98, 2019.

[5] D. J. Solove and P. M. Schwartz, *Information privacy law*. Aspen Publishing, 2020.

[6] J. Backes, S. Bayless, B. Cook, C. Dodge, A. Gacek, A. J. Hu, T. Kahsai, B. Kocik, E. Kotelnikov, J. Kukovec *et al.*, "Reachability analysis for aws-based networks," in *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II 31*. Springer, 2019, pp. 231–241.

[7] "Azure's network watcher." [Online]. Available: https://azure.microsoft.com/en-us/products/network-watcher/

[8] "Aws vpc reachability analyzer." [Online]. Available: https://docs.aws.amazon.com/vpc/latest/reachability

[9] M. Ganai and A. Gupta, *SAT-based scalable formal verification solutions*. Springer, 2007.

[10] S. B. Akers, "Binary decision diagrams," *IEEE Transactions on computers*, vol. 27, no. 06, pp. 509–516, 1978.

[11] C. Ansótegui, M. L. Bonet, and J. Levy, "Sat-based maxsat algorithms," *Artificial Intelligence*, vol. 196, pp. 77–105, 2013.

[12] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen *et al.*, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 139–152.

[13] X. Xu, R. Beckett, K. Jayaraman, R. Mahajan, and D. Walker, "Test coverage metrics for the network," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 775–787.

[14] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein, "A general approach to network configuration analysis," in *12th {USENIX} symposium on networked systems design and implementation ({NSDI} 15)*, 2015, pp. 469–483.

[15] R. Potharaju and N. Jain, "When the network crumbles: An empirical study of cloud network failures and their impact on services," in *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013, pp. 1–17.

[16] R. Kumar and R. Goyal, "On cloud security requirements, threats, vulnerabilities and countermeasures: A survey," *Computer Science Review*, vol. 33, pp. 1–48, 2019.

[17] T. Schneider, R. Schmid, and L. Vanbever, "On the complexity of network-wide configuration synthesis," in *2022 IEEE 30th International Conference on Network Protocols (ICNP)*. IEEE, 2022, pp. 1–11.

[18] Y. Velner, K. Alpernas, A. Panda, A. Rabinovich, M. Sagiv, S. Shenker, and S. Shoham, "Some complexity results for stateful network verification," in *Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings 22*. Springer, 2016, pp. 811–830.

[19] N. Narodytska and F. Bacchus, "Maximum satisfiability using core-guided maxsat resolution," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 28, no. 1, 2014.

[20] B. Khorchani, S. Hallé, and R. Villemaire, "Firewall anomaly detection with a model checker for visibility logic," in *2012 IEEE Network Operations and Management Symposium*. IEEE, 2012, pp. 466–469.

[21] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*. Springer, 2008, pp. 337–340.

[22] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, 2012, pp. 113–126.

[23] P. Zhang, X. Liu, H. Yang, N. Kang, Z. Gu, and H. Li, "Apkeep: Realtime verification for real networks." in *NSDI*, 2020, pp. 241–255.

[24] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," *IEEE/ACM Transactions on Networking*, vol. 24, no. 2, pp. 887–900, 2015.

[25] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar, "Plankton: Scalable network configuration verification through model checking," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 953–967.

[26] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 290–301, 2011.

[27] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis." in *NSDI*, 2013, pp. 99–111.

[28] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *12th {USENIX} symposium on networked systems design and implementation ({NSDI} 15)*, 2015, pp. 499–512.

[29] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 155–168.

[30] S. Bayless, J. Backes, D. DaCosta, B. F. Jones, N. Launchbury, P. Trentin, K. Jewell, S. Joshi, M. Q. Zeng, and N. Mathews, in *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II 33*. Springer, 2021, pp. 851–862.

[31] S. Saha, S. Prabhu, and P. Madhusudan, "Netgen: Synthesizing data-plane configurations for network policies," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, 2015, pp. 1–6.

[32] B. Tian, X. Zhang, E. Zhai, H. H. Liu, Q. Ye, C. Wang, X. Wu, Z. Ji, Y. Sang, M. Zhang *et al.*, "Safely and automatically updating in-network acl configurations with intent language," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 214–226.

[33] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev, "Netcomplete: Practical network-wide configuration synthesis with autocompletion," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 579–594.

[34] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker, "Network configuration synthesis with abstract topologies," in *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*, 2017, pp. 437–451.

[35] Y. Yuan, R. Alur, and B. T. Loo, "Netegg: Programming network policies by examples," in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, 2014, pp. 1–7.

[36] A. Abhashkumar, A. Gember-Jacobson, and A. Akella, "Aed: Incrementally synthesizing policy-compliant and manageable configurations," in *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, 2020, pp. 482–495.

Large-scale cluster management at Google with Borg

Abhishek Verma† Luis Pedrosa‡ Madhukar Korupolu
David Oppenheimer Eric Tune John Wilkes

Google Inc.

Abstract

Google's Borg system is a cluster manager that runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up to tens of thousands of machines.

It achieves high utilization by combining admission control, efficient task-packing, over-commitment, and machine sharing with process-level performance isolation. It supports high-availability applications with runtime features that minimize fault-recovery time, and scheduling policies that reduce the probability of correlated failures. Borg simplifies life for its users by offering a declarative job specification language, name service integration, real-time job monitoring, and tools to analyze and simulate system behavior.

We present a summary of the Borg system architecture and features, important design decisions, a quantitative analysis of some of its policy decisions, and a qualitative examination of lessons learned from a decade of operational experience with it.

1. Introduction

The cluster management system we internally call Borg admits, schedules, starts, restarts, and monitors the full range of applications that Google runs. This paper explains how.

Borg provides three main benefits: it (1) hides the details of resource management and failure handling so its users can focus on application development instead; (2) operates with very high reliability and availability, and supports applications that do the same; and (3) lets us run workloads across tens of thousands of machines effectively. Borg is not the first system to address these issues, but it's one of the few operating at this scale, with this degree of resiliency and completeness. This paper is organized around these topics, concluding with a set of qualitative observations we have made from operating Borg in production for more than a decade.

† Work done while author was at Google.

‡ Currently at University of Southern California.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

EuroSys'15, April 21–24, 2015, Bordeaux, France.
Copyright is held by the owner/author(s).
ACM 978-1-4503-3238-5/15/04.
http://dx.doi.org/10.1145/2741948.2741964

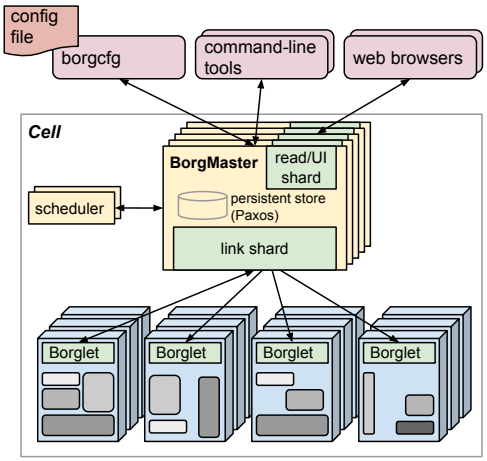


Figure 1: The high-level architecture of Borg. *Only a tiny fraction of the thousands of worker nodes are shown.*

2. The user perspective

Borg's users are Google developers and system administrators (site reliability engineers or SREs) that run Google's applications and services. Users submit their work to Borg in the form of *jobs*, each of which consists of one or more *tasks* that all run the same program (binary). Each job runs in one Borg *cell*, a set of machines that are managed as a unit. The remainder of this section describes the main features exposed in the user view of Borg.

2.1 The workload

Borg cells run a heterogenous workload with two main parts. The first is long-running services that should "never" go down, and handle short-lived latency-sensitive requests (a few μs to a few hundred ms). Such services are used for end-user-facing products such as Gmail, Google Docs, and web search, and for internal infrastructure services (e.g., BigTable). The second is batch jobs that take from a few seconds to a few days to complete; these are much less sensitive to short-term performance fluctuations. The workload mix varies across cells, which run different mixes of applications depending on their major tenants (e.g., some cells are quite batch-intensive), and also varies over time: batch jobs

come and go, and many end-user-facing service jobs see a diurnal usage pattern. Borg is required to handle all these cases equally well.

A representative Borg workload can be found in a publicly-available month-long trace from May 2011 [80], which has been extensively analyzed (e.g., [68] and [1, 26, 27, 57]).

Many application frameworks have been built on top of Borg over the last few years, including our internal MapReduce system [23], FlumeJava [18], Millwheel [3], and Pregel [59]. Most of these have a controller that submits a master job and one or more worker jobs; the first two play a similar role to YARN's application manager [76]. Our distributed storage systems such as GFS [34] and its successor CFS, Bigtable [19], and Megastore [8] all run on Borg.

For this paper, we classify higher-priority Borg jobs as "production" (prod) ones, and the rest as "non-production" (non-prod). Most long-running server jobs are prod; most batch jobs are non-prod. In a representative cell, prod jobs are allocated about 70% of the total CPU resources and represent about 60% of the total CPU usage; they are allocated about 55% of the total memory and represent about 85% of the total memory usage. The discrepancies between allocation and usage will prove important in §5.5.

2.2 Clusters and cells

The machines in a cell belong to a single *cluster*, defined by the high-performance datacenter-scale network fabric that connects them. A cluster lives inside a single datacenter building, and a collection of buildings makes up a site.[1] A cluster usually hosts one large cell and may have a few smaller-scale test or special-purpose cells. We assiduously avoid any single point of failure.

Our median cell size is about 10 k machines after excluding test cells; some are much larger. The machines in a cell are heterogeneous in many dimensions: sizes (CPU, RAM, disk, network), processor type, performance, and capabilities such as an external IP address or flash storage. Borg isolates users from most of these differences by determining where in a cell to run tasks, allocating their resources, installing their programs and other dependencies, monitoring their health, and restarting them if they fail.

2.3 Jobs and tasks

A Borg job's properties include its name, owner, and the number of tasks it has. Jobs can have *constraints* to force its tasks to run on machines with particular *attributes* such as processor architecture, OS version, or an external IP address. Constraints can be hard or soft; the latter act like preferences rather than requirements. The start of a job can be deferred until a prior one finishes. A job runs in just one cell.

Each task maps to a set of Linux processes running in a container on a machine [62]. The vast majority of the Borg workload does not run inside virtual machines (VMs), because we don't want to pay the cost of virtualization. Also, the system was designed at a time when we had a considerable investment in processors with no virtualization support in hardware.

A task has properties too, such as its resource requirements and the task's index within the job. Most task properties are the same across all tasks in a job, but can be overridden – e.g., to provide task-specific command-line flags. Each resource dimension (CPU cores, RAM, disk space, disk access rate, TCP ports,[2] etc.) is specified independently at fine granularity; we don't impose fixed-sized buckets or slots (§5.4). Borg programs are statically linked to reduce dependencies on their runtime environment, and structured as *packages* of binaries and data files, whose installation is orchestrated by Borg.

Users operate on jobs by issuing remote procedure calls (RPCs) to Borg, most commonly from a command-line tool, other Borg jobs, or our monitoring systems (§2.6). Most job descriptions are written in the declarative configuration language BCL. This is a variant of GCL [12], which generates protobuf files [67], extended with some Borg-specific keywords. GCL provides lambda functions to allow calculations, and these are used by applications to adjust their configurations to their environment; tens of thousands of BCL files are over 1 k lines long, and we have accumulated tens of millions of lines of BCL. Borg job configurations have similarities to Aurora configuration files [6].

Figure 2 illustrates the states that jobs and tasks go through during their lifetime.

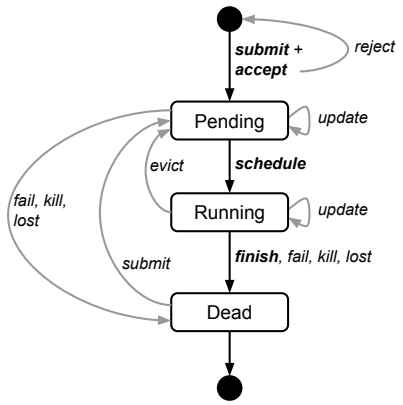


Figure 2: The state diagram for both jobs and tasks. *Users can trigger* submit, *kill, and* update *transitions.*

A user can change the properties of some or all of the tasks in a running job by pushing a new job configuration to Borg, and then instructing Borg to *update* the tasks to the new specification. This acts as a lightweight, non-atomic transaction that can easily be undone until it is closed (committed). Updates are generally done in a rolling fashion, and a limit can be imposed on the number of task disruptions

[1] There are a few exceptions for each of these relationships.

[2] Borg manages the available ports on a machine and allocates them to tasks.

(reschedules or preemptions) an update causes; any changes that would cause more disruptions are skipped.

Some task updates (e.g., pushing a new binary) will always require the task to be restarted; some (e.g., increasing resource requirements or changing constraints) might make the task no longer fit on the machine, and cause it to be stopped and rescheduled; and some (e.g., changing priority) can always be done without restarting or moving the task.

Tasks can ask to be notified via a Unix SIGTERM signal before they are preempted by a SIGKILL, so they have time to clean up, save state, finish any currently-executing requests, and decline new ones. The actual notice may be less if the preemptor sets a delay bound. In practice, a notice is delivered about 80% of the time.

2.4 Allocs

A Borg *alloc* (short for allocation) is a reserved set of resources on a machine in which one or more tasks can be run; the resources remain assigned whether or not they are used. Allocs can be used to set resources aside for future tasks, to retain resources between stopping a task and starting it again, and to gather tasks from different jobs onto the same machine – e.g., a web server instance and an associated logsaver task that copies the server's URL logs from the local disk to a distributed file system. The resources of an alloc are treated in a similar way to the resources of a machine; multiple tasks running inside one share its resources. If an alloc must be relocated to another machine, its tasks are rescheduled with it.

An *alloc set* is like a job: it is a group of allocs that reserve resources on multiple machines. Once an alloc set has been created, one or more jobs can be submitted to run in it. For brevity, we will generally use "task" to refer to an alloc or a top-level task (one outside an alloc) and "job" to refer to a job or alloc set.

2.5 Priority, quota, and admission control

What happens when more work shows up than can be accommodated? Our solutions for this are priority and quota.

Every job has a *priority*, a small positive integer. A high-priority task can obtain resources at the expense of a lower-priority one, even if that involves preempting (killing) the latter. Borg defines non-overlapping *priority bands* for different uses, including (in decreasing-priority order): monitoring, production, batch, and best effort (also known as testing or free). For this paper, prod jobs are the ones in the monitoring and production bands.

Although a preempted task will often be rescheduled elsewhere in the cell, preemption cascades could occur if a high-priority task bumped out a slightly lower-priority one, which bumped out another slightly-lower priority task, and so on. To eliminate most of this, we disallow tasks in the production priority band to preempt one another. Fine-grained priorities are still useful in other circumstances – e.g., MapReduce master tasks run at a slightly higher priority than the workers they control, to improve their reliability.

Priority expresses relative importance for jobs that are running or waiting to run in a cell. *Quota* is used to decide which jobs to *admit* for scheduling. Quota is expressed as a vector of resource quantities (CPU, RAM, disk, etc.) at a given priority, for a period of time (typically months). The quantities specify the maximum amount of resources that a user's job requests can ask for at a time (e.g., "20 TiB of RAM at prod priority from now until the end of July in cell xx"). Quota-checking is part of admission control, not scheduling: jobs with insufficient quota are immediately rejected upon submission.

Higher-priority quota costs more than quota at lower-priority. Production-priority quota is limited to the actual resources available in the cell, so that a user who submits a production-priority job that fits in their quota can expect it to run, modulo fragmentation and constraints. Even though we encourage users to purchase no more quota than they need, many users overbuy because it insulates them against future shortages when their application's user base grows. We respond to this by over-selling quota at lower-priority levels: every user has infinite quota at priority zero, although this is frequently hard to exercise because resources are oversubscribed. A low-priority job may be admitted but remain pending (unscheduled) due to insufficient resources.

Quota allocation is handled outside of Borg, and is intimately tied to our physical capacity planning, whose results are reflected in the price and availability of quota in different datacenters. User jobs are admitted only if they have sufficient quota at the required priority. The use of quota reduces the need for policies like Dominant Resource Fairness (DRF) [29, 35, 36, 66].

Borg has a capability system that gives special privileges to some users; for example, allowing administrators to delete or modify any job in the cell, or allowing a user to access restricted kernel features or Borg behaviors such as disabling resource estimation (§5.5) on their jobs.

2.6 Naming and monitoring

It's not enough to create and place tasks: a service's clients and other systems need to be able to find them, even after they are relocated to a new machine. To enable this, Borg creates a stable "Borg name service" (BNS) name for each task that includes the cell name, job name, and task number. Borg writes the task's hostname and port into a consistent, highly-available file in Chubby [14] with this name, which is used by our RPC system to find the task endpoint. The BNS name also forms the basis of the task's DNS name, so the fiftieth task in job jfoo owned by user ubar in cell cc would be reachable via 50.jfoo.ubar.cc.borg.google.com. Borg also writes job size and task health information into Chubby whenever it changes, so load balancers can see where to route requests to.

Almost every task run under Borg contains a built-in HTTP server that publishes information about the health of the task and thousands of performance metrics (e.g., RPC latencies). Borg monitors the health-check URL and restarts tasks that do not respond promptly or return an HTTP error code. Other data is tracked by monitoring tools for dashboards and alerts on service level objective (SLO) violations.

A service called Sigma provides a web-based user interface (UI) through which a user can examine the state of all their jobs, a particular cell, or drill down to individual jobs and tasks to examine their resource behavior, detailed logs, execution history, and eventual fate. Our applications generate voluminous logs; these are automatically rotated to avoid running out of disk space, and preserved for a while after the task's exit to assist with debugging. If a job is not running Borg provides a "why pending?" annotation, together with guidance on how to modify the job's resource requests to better fit the cell. We publish guidelines for "conforming" resource shapes that are likely to schedule easily.

Borg records all job submissions and task events, as well as detailed per-task resource usage information in Infrastore, a scalable read-only data store with an interactive SQL-like interface via Dremel [61]. This data is used for usage-based charging, debugging job and system failures, and long-term capacity planning. It also provided the data for the Google cluster workload trace [80].

All of these features help users to understand and debug the behavior of Borg and their jobs, and help our SREs manage a few tens of thousands of machines per person.

3. Borg architecture

A Borg cell consists of a set of machines, a logically centralized controller called the Borgmaster, and an agent process called the Borglet that runs on each machine in a cell (see Figure 1). All components of Borg are written in C++.

3.1 Borgmaster

Each cell's Borgmaster consists of two processes: the main Borgmaster process and a separate scheduler (§3.2). The main Borgmaster process handles client RPCs that either mutate state (e.g., `create job`) or provide read-only access to data (e.g., `lookup job`). It also manages state machines for all of the objects in the system (machines, tasks, allocs, etc.), communicates with the Borglets, and offers a web UI as a backup to Sigma.

The Borgmaster is logically a single process but is actually replicated five times. Each replica maintains an in-memory copy of most of the state of the cell, and this state is also recorded in a highly-available, distributed, Paxos-based store [55] on the replicas' local disks. A single *elected master* per cell serves both as the Paxos leader and the state mutator, handling all operations that change the cell's state, such as submitting a job or terminating a task on a machine. A master is elected (using Paxos) when the cell is brought up and whenever the elected master fails; it acquires a Chubby lock so other systems can find it. Electing a master and failing-over to the new one typically takes about 10 s, but can take up to a minute in a big cell because some in-memory state has to be reconstructed. When a replica recovers from an outage, it dynamically re-synchronizes its state from other Paxos replicas that are up-to-date.

The Borgmaster's state at a point in time is called a *checkpoint*, and takes the form of a periodic snapshot plus a change log kept in the Paxos store. Checkpoints have many uses, including restoring a Borgmaster's state to an arbitrary point in the past (e.g., just before accepting a request that triggered a software defect in Borg so it can be debugged); fixing it by hand *in extremis*; building a persistent log of events for future queries; and offline simulations.

A high-fidelity Borgmaster simulator called Fauxmaster can be used to read checkpoint files, and contains a complete copy of the production Borgmaster code, with stubbed-out interfaces to the Borglets. It accepts RPCs to make state machine changes and perform operations, such as "schedule all pending tasks", and we use it to debug failures, by interacting with it as if it were a live Borgmaster, with simulated Borglets replaying real interactions from the checkpoint file. A user can step through and observe the changes to the system state that actually occurred in the past. Fauxmaster is also useful for capacity planning ("how many new jobs of this type would fit?"), as well as sanity checks before making a change to a cell's configuration ("will this change evict any important jobs?").

3.2 Scheduling

When a job is submitted, the Borgmaster records it persistently in the Paxos store and adds the job's tasks to the *pending* queue. This is scanned asynchronously by the *scheduler*, which assigns tasks to machines if there are sufficient available resources that meet the job's constraints. (The scheduler primarily operates on tasks, not jobs.) The scan proceeds from high to low priority, modulated by a round-robin scheme within a priority to ensure fairness across users and avoid head-of-line blocking behind a large job. The scheduling algorithm has two parts: *feasibility checking*, to find machines on which the task could run, and *scoring*, which picks one of the feasible machines.

In feasibility checking, the scheduler finds a set of machines that meet the task's constraints and also have enough "available" resources – which includes resources assigned to lower-priority tasks that can be evicted. In scoring, the scheduler determines the "goodness" of each feasible machine. The score takes into account user-specified preferences, but is mostly driven by built-in criteria such as minimizing the number and priority of preempted tasks, picking machines that already have a copy of the task's packages, spreading tasks across power and failure domains, and packing quality including putting a mix of high and low priority

tasks onto a single machine to allow the high-priority ones to expand in a load spike.

Borg originally used a variant of E-PVM [4] for scoring, which generates a single cost value across heterogeneous resources and minimizes the change in cost when placing a task. In practice, E-PVM ends up spreading load across all the machines, leaving headroom for load spikes – but at the expense of increased fragmentation, especially for large tasks that need most of the machine; we sometimes call this "worst fit".

The opposite end of the spectrum is "best fit", which tries to fill machines as tightly as possible. This leaves some machines empty of user jobs (they still run storage servers), so placing large tasks is straightforward, but the tight packing penalizes any mis-estimations in resource requirements by users or Borg. This hurts applications with bursty loads, and is particularly bad for batch jobs which specify low CPU needs so they can schedule easily and try to run opportunistically in unused resources: 20% of non-prod tasks request less than 0.1 CPU cores.

Our current scoring model is a hybrid one that tries to reduce the amount of stranded resources – ones that cannot be used because another resource on the machine is fully allocated. It provides about 3–5% better packing efficiency (defined in [78]) than best fit for our workloads.

If the machine selected by the scoring phase doesn't have enough available resources to fit the new task, Borg *preempts* (kills) lower-priority tasks, from lowest to highest priority, until it does. We add the preempted tasks to the scheduler's pending queue, rather than migrate or hibernate them.[3]

Task startup latency (the time from job submission to a task running) is an area that has received and continues to receive significant attention. It is highly variable, with the median typically about 25 s. Package installation takes about 80% of the total: one of the known bottlenecks is contention for the local disk where packages are written to. To reduce task startup time, the scheduler prefers to assign tasks to machines that already have the necessary packages (programs and data) installed: most packages are immutable and so can be shared and cached. (This is the only form of data locality supported by the Borg scheduler.) In addition, Borg distributes packages to machines in parallel using tree- and torrent-like protocols.

Additionally, the scheduler uses several techniques to let it scale up to cells with tens of thousands of machines (§3.4).

3.3 Borglet

The Borglet is a local Borg agent that is present on every machine in a cell. It starts and stops tasks; restarts them if they fail; manages local resources by manipulating OS kernel settings; rolls over debug logs; and reports the state of the machine to the Borgmaster and other monitoring systems.

[3] Exception: tasks that provide virtual machines for Google Compute Engine users are migrated.

The Borgmaster polls each Borglet every few seconds to retrieve the machine's current state and send it any outstanding requests. This gives Borgmaster control over the rate of communication, avoids the need for an explicit flow control mechanism, and prevents recovery storms [9].

The elected master is responsible for preparing messages to send to the Borglets and for updating the cell's state with their responses. For performance scalability, each Borgmaster replica runs a stateless *link shard* to handle the communication with some of the Borglets; the partitioning is recalculated whenever a Borgmaster election occurs. For resiliency, the Borglet always reports its full state, but the link shards aggregate and compress this information by reporting only differences to the state machines, to reduce the update load at the elected master.

If a Borglet does not respond to several poll messages its machine is marked as down and any tasks it was running are rescheduled on other machines. If communication is restored the Borgmaster tells the Borglet to kill those tasks that have been rescheduled, to avoid duplicates. A Borglet continues normal operation even if it loses contact with the Borgmaster, so currently-running tasks and services stay up even if all Borgmaster replicas fail.

3.4 Scalability

We are not sure where the ultimate scalability limit to Borg's centralized architecture will come from; so far, every time we have approached a limit, we've managed to eliminate it. A single Borgmaster can manage many thousands of machines in a cell, and several cells have arrival rates above 10 000 tasks per minute. A busy Borgmaster uses 10–14 CPU cores and up to 50 GiB RAM. We use several techniques to achieve this scale.

Early versions of Borgmaster had a simple, synchronous loop that accepted requests, scheduled tasks, and communicated with Borglets. To handle larger cells, we split the scheduler into a separate process so it could operate in parallel with the other Borgmaster functions that are replicated for failure tolerance. A scheduler replica operates on a cached copy of the cell state. It repeatedly: retrieves state changes from the elected master (including both assigned and pending work); updates its local copy; does a scheduling pass to assign tasks; and informs the elected master of those assignments. The master will accept and apply these assignments unless they are inappropriate (e.g., based on out of date state), which will cause them to be reconsidered in the scheduler's next pass. This is quite similar in spirit to the optimistic concurrency control used in Omega [69], and indeed we recently added the ability for Borg to use different schedulers for different workload types.

To improve response times, we added separate threads to talk to the Borglets and respond to read-only RPCs. For greater performance, we sharded (partitioned) these functions across the five Borgmaster replicas §3.3. Together,

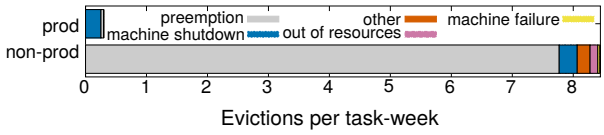


Figure 3: Task-eviction rates and causes for production and non-production workloads. *Data from August 1st 2013.*

these keep the 99%ile response time of the UI below 1 s and the 95%ile of the Borglet polling interval below 10 s.

Several things make the Borg scheduler more scalable:

Score caching: Evaluating feasibility and scoring a machine is expensive, so Borg caches the scores until the properties of the machine or task change – e.g., a task on the machine terminates, an attribute is altered, or a task's requirements change. Ignoring small changes in resource quantities reduces cache invalidations.

Equivalence classes: Tasks in a Borg job usually have identical requirements and constraints, so rather than determining feasibility for every pending task on every machine, and scoring all the feasible machines, Borg only does feasibility and scoring for one task per *equivalence class* – a group of tasks with identical requirements.

Relaxed randomization: It is wasteful to calculate feasibility and scores for all the machines in a large cell, so the scheduler examines machines in a random order until it has found "enough" feasible machines to score, and then selects the best within that set. This reduces the amount of scoring and cache invalidations needed when tasks enter and leave the system, and speeds up assignment of tasks to machines. Relaxed randomization is somewhat akin to the batch sampling of Sparrow [65] while also handling priorities, preemptions, heterogeneity and the costs of package installation.

In our experiments (§5), scheduling a cell's entire workload from scratch typically took a few hundred seconds, but did not finish after more than 3 days when the above techniques were disabled. Normally, though, an online scheduling pass over the pending queue completes in less than half a second.

4. Availability

Failures are the norm in large scale systems [10, 11, 22]. Figure 3 provides a breakdown of task eviction causes in 15 sample cells. Applications that run on Borg are expected to handle such events, using techniques such as replication, storing persistent state in a distributed file system, and (if appropriate) taking occasional checkpoints. Even so, we try to mitigate the impact of these events. For example, Borg:

- automatically reschedules evicted tasks, on a new machine if necessary;
- reduces correlated failures by spreading tasks of a job across failure domains such as machines, racks, and power domains;
- limits the allowed rate of task disruptions and the number of tasks from a job that can be simultaneously down during maintenance activities such as OS or machine upgrades;
- uses declarative desired-state representations and idempotent mutating operations, so that a failed client can harmlessly resubmit any forgotten requests;
- rate-limits finding new places for tasks from machines that become unreachable, because it cannot distinguish between large-scale machine failure and a network partition;
- avoids repeating task::machine pairings that cause task or machine crashes; and
- recovers critical intermediate data written to local disk by repeatedly re-running a logsaver task (§2.4), even if the alloc it was attached to is terminated or moved to another machine. Users can set how long the system keeps trying; a few days is common.

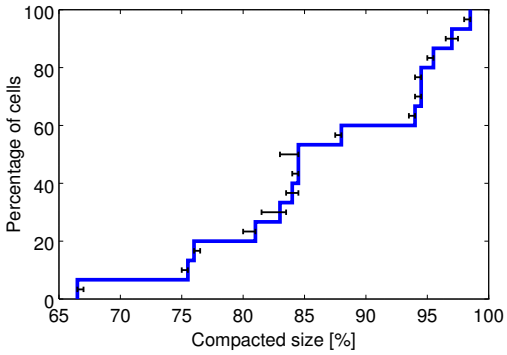


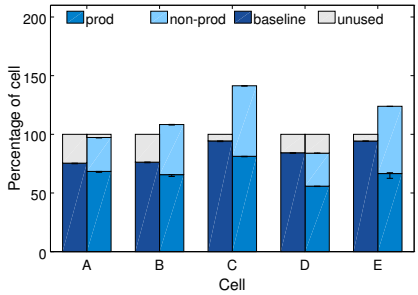
Figure 4: The effects of compaction. *A CDF of the percentage of original cell size achieved after compaction, across 15 cells.*

A key design feature in Borg is that already-running tasks continue to run even if the Borgmaster or a task's Borglet goes down. But keeping the master up is still important because when it is down new jobs cannot be submitted or existing ones updated, and tasks from failed machines cannot be rescheduled.

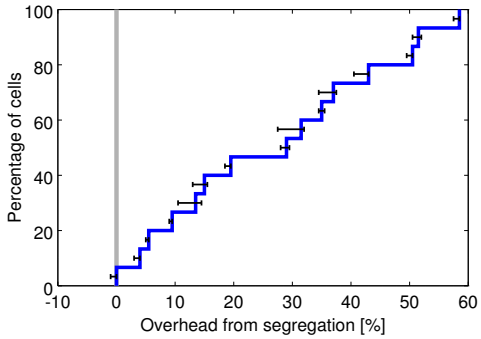
Borgmaster uses a combination of techniques that enable it to achieve 99.99% availability in practice: replication for machine failures; admission control to avoid overload; and deploying instances using simple, low-level tools to minimize external dependencies. Each cell is independent of the others to minimize the chance of correlated operator errors and failure propagation. These goals, not scalability limitations, are the primary argument against larger cells.

5. Utilization

One of Borg's primary goals is to make efficient use of Google's fleet of machines, which represents a significant financial investment: increasing utilization by a few percentage points can save millions of dollars. This section discusses and evaluates some of the policies and techniques that Borg uses to do so.



(a) *The left column for each cell shows the original size and the combined workload; the right one shows the segregated case.*



(b) *CDF of additional machines that would be needed if we segregated the workload of 15 representative cells.*

Figure 5: Segregating prod and non-prod work into different cells would need more machines. *Both graphs show how many extra machines would be needed if the* prod *and* non-prod *workloads were sent to separate cells, expressed as a percentage of the minimum number of machines required to run the workload in a single cell. In this, and subsequent CDF plots, the value shown for each cell is derived from the 90%ile of the different cell sizes our experiment trials produced; the error bars show the complete range of values from the trials.*

5.1 Evaluation methodology

Our jobs have placement constraints and need to handle rare workload spikes, our machines are heterogenous, and we run batch jobs in resources reclaimed from service jobs. So, to evaluate our policy choices we needed a more sophisticated metric than "average utilization". After much experimentation we picked *cell compaction*: given a workload, we found out how small a cell it could be fitted into by removing machines until the workload no longer fitted, repeatedly re-packing the workload from scratch to ensure that we didn't get hung up on an unlucky configuration. This provided clean termination conditions and facilitated automated comparisons without the pitfalls of synthetic workload generation and modeling [31]. A quantitative comparison of evaluation techniques can be found in [78]: the details are surprisingly subtle.

It wasn't possible to perform experiments on live production cells, but we used Fauxmaster to obtain high-fidelity simulation results, using data from real production cells and workloads, including all their constraints, actual limits, reservations, and usage data (§5.5). This data came from Borg checkpoints taken on Wednesday 2014-10-01 14:00 PDT. (Other checkpoints produced similar results.) We picked 15 Borg cells to report on by first eliminating special-purpose, test, and small (< 5000 machines) cells, and then sampled the remaining population to achieve a roughly even spread across the range of sizes.

To maintain machine heterogeneity in the compacted cell we randomly selected machines to remove. To maintain workload heterogeneity, we kept it all, except for server and storage tasks tied to a particular machine (e.g., the Borglets). We changed hard constraints to soft ones for jobs larger than half the original cell size, and allowed up to 0.2% tasks to go pending if they were very "picky" and could only be placed on a handful of machines; extensive experiments showed that this produced repeatable results with low variance. If

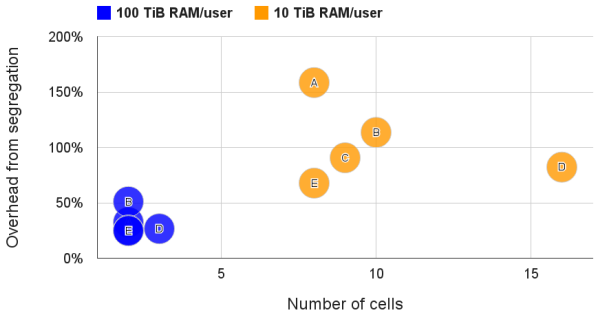
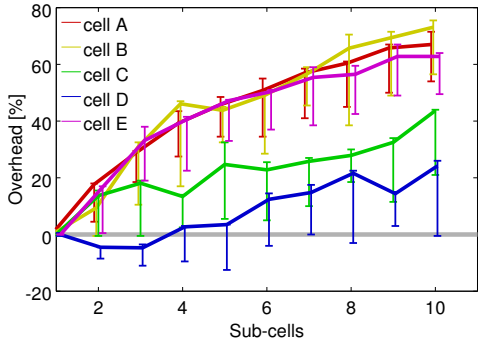


Figure 6: Segregating users would need more machines. *The total number of cells and the additional machines that would be needed if users larger than the threshold shown were given their own private cells, for 5 different cells.*

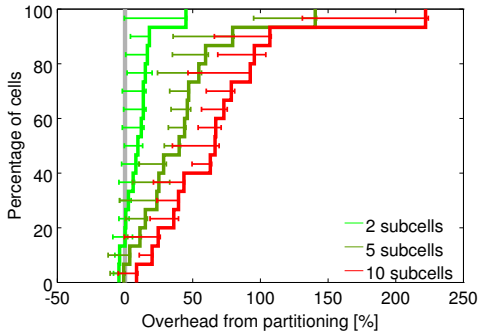
we needed a larger cell than the original we cloned the original cell a few times before compaction; if we needed more cells, we just cloned the original.

Each experiment was repeated 11 times for each cell with different random-number seeds. In the graphs, we use an error bar to display the min and max of the number of machines needed, and select the 90%ile value as the "result" – the mean or median would not reflect what a system administrator would do if they wanted to be reasonably sure that the workload would fit. We believe cell compaction provides a fair, consistent way to compare scheduling policies, and it translates directly into a cost/benefit result: better policies require fewer machines to run the same workload.

Our experiments focused on scheduling (packing) a workload from a point in time, rather than replaying a long-term workload trace. This was partly to avoid the difficulties of coping with open and closed queueing models [71, 79], partly because traditional time-to-completion metrics don't apply to our environment with its long-running services, partly to provide clean signals for making comparisons,



(a) *Additional machines that would be needed as a function of the number of smaller cells for five different original cells.*



(b) *A CDF of additional machines that would be needed to divide each of 15 different cells into 2, 5 or 10 cells.*

Figure 7: Subdividing cells into smaller ones would require more machines. *The additional machines (as a percentage of the single-cell case) that would be needed if we divided these particular cells into a varying number of smaller cells.*

partly because we don't believe the results would be significantly different, and partly a practical matter: we found ourselves consuming 200 000 Borg CPU cores for our experiments at one point—even at Google's scale, this is a non-trivial investment.

In production, we deliberately leave significant headroom for workload growth, occasional "black swan" events, load spikes, machine failures, hardware upgrades, and large-scale partial failures (e.g., a power supply bus duct). Figure 4 shows how much smaller our real-world cells would be if we were to apply cell compaction to them. The baselines in the graphs that follow use these compacted sizes.

5.2 Cell sharing

Nearly all of our machines run both prod and non-prod tasks at the same time: 98% of the machines in shared Borg cells, 83% across the entire set of machines managed by Borg. (We have a few dedicated cells for special uses.)

Since many other organizations run user-facing and batch jobs in separate clusters, we examined what would happen if we did the same. Figure 5 shows that segregating prod and non-prod work would need 20–30% more machines in the median cell to run our workload. That's because prod jobs usually reserve resources to handle rare workload spikes, but don't use these resources most of the time. Borg reclaims the unused resources (§5.5) to run much of the non-prod work, so we need fewer machines overall.

Most Borg cells are shared by thousands of users. Figure 6 shows why. For this test, we split off a user's workload into a new cell if they consumed at least 10 TiB of memory (or 100 TiB). Our existing policy looks good: even with the larger threshold, we would need 2–16× as many cells, and 20–150% additional machines. Once again, pooling resources significantly reduces costs.

But perhaps packing unrelated users and job types onto the same machines results in CPU interference, and so we would need more machines to compensate? To assess this, we looked at how the CPI (cycles per instruction) changed for tasks in different environments running on the same machine type with the same clock speed. Under these conditions, CPI values are comparable and can be used as a proxy for performance interference, since a doubling of CPI doubles the runtime of a CPU-bound program. The data was gathered from ∼ 12 000 randomly selected prod tasks over a week, counting cycles and instructions over a 5 minute interval using the hardware profiling infrastructure described in [83], and weighting samples so that every second of CPU time is counted equally. The results were not clear-cut.

(1) We found that CPI was positively correlated with two measurements over the same time interval: the overall CPU usage on the machine, and (largely independently) the number of tasks on the machine; adding a task to a machine increases the CPI of other tasks by 0.3% (using a linear model fitted to the data); increasing machine CPU usage by 10% increases CPI by less than 2%. But even though the correlations are statistically significant, they only explain 5% of the variance we saw in CPI measurements; other factors dominate, such as inherent differences in applications and specific interference patterns [24, 83].

(2) Comparing the CPIs we sampled from shared cells to ones from a few dedicated cells with less diverse applications, we saw a mean CPI of 1.58 ($\sigma = 0.35$) in shared cells and a mean of 1.53 ($\sigma = 0.32$) in dedicated cells – i.e., CPU performance is about 3% worse in shared cells.

(3) To address the concern that applications in different cells might have different workloads, or even suffer selection bias (maybe programs that are more sensitive to interference had been moved to dedicated cells), we looked at the CPI of the Borglet, which runs on all the machines in both types of cell. We found it had a CPI of 1.20 ($\sigma = 0.29$) in dedicated cells and 1.43 ($\sigma = 0.45$) in shared ones, suggesting that it runs 1.19× as fast in a dedicated cell as in a shared one, although this over-weights the effect of lightly loaded machines, slightly biasing the result in favor of dedicated cells.

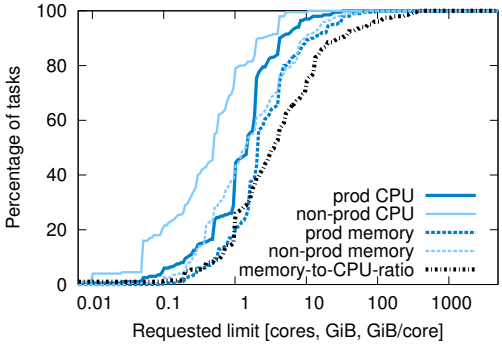


Figure 8: No bucket sizes fit most of the tasks well. *CDF of requested CPU and memory requests across our sample cells. No one value stands out, although a few integer CPU core sizes are somewhat more popular.*

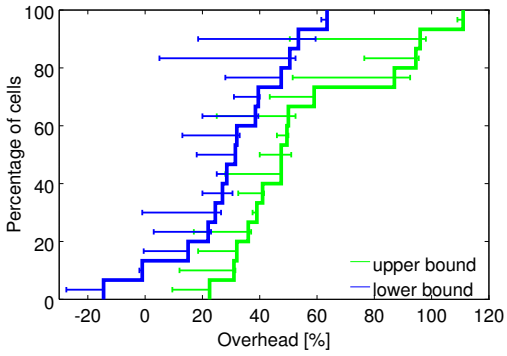


Figure 9: "Bucketing" resource requirements would need more machines. *A CDF of the additional overheads that would result from rounding up CPU and memory requests to the next nearest powers of 2 across 15 cells. The lower and upper bounds straddle the actual values (see the text).*

These experiments confirm that performance comparisons at warehouse-scale are tricky, reinforcing the observations in [51], and also suggest that sharing doesn't drastically increase the cost of running programs.

But even assuming the least-favorable of our results, sharing is still a win: the CPU slowdown is outweighed by the decrease in machines required over several different partitioning schemes, and the sharing advantages apply to all resources including memory and disk, not just CPU.

5.3 Large cells

Google builds large cells, both to allow large computations to be run, and to decrease resource fragmentation. We tested the effects of the latter by partitioning the workload for a cell across multiple smaller cells – by first randomly permuting the jobs and then assigning them in a round-robin manner among the partitions. Figure 7 confirms that using smaller cells would require significantly more machines.

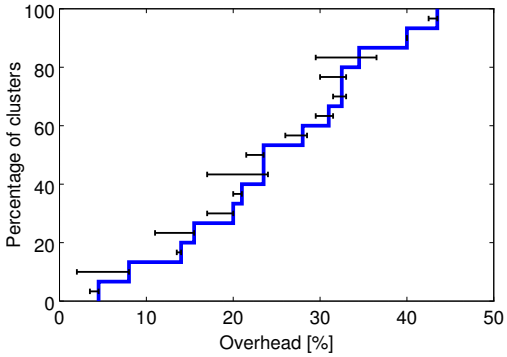


Figure 10: Resource reclamation is quite effective. *A CDF of the additional machines that would be needed if we disabled it for 15 representative cells.*

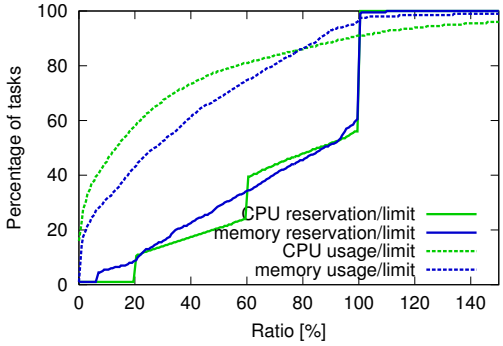


Figure 11: Resource estimation is successful at identifying unused resources. *The dotted lines shows CDFs of the ratio of CPU and memory usage to the request (limit) for tasks across 15 cells. Most tasks use much less than their limit, although a few use more CPU than requested. The solid lines show the CDFs of the ratio of CPU and memory reservations to the limits; these are closer to 100%. The straight lines are artifacts of the resource-estimation process.*

5.4 Fine-grained resource requests

Borg users request CPU in units of milli-cores, and memory and disk space in bytes. (A *core* is a processor hyperthread, normalized for performance across machine types.) Figure 8 shows that they take advantage of this granularity: there are few obvious "sweet spots" in the amount of memory or CPU cores requested, and few obvious correlations between these resources. These distributions are quite similar to the ones presented in [68], except that we see slightly larger memory requests at the 90%ile and above.

Offering a set of fixed-size containers or virtual machines, although common among IaaS (infrastructure-as-a-service) providers [7, 33], would not be a good match to our needs. To show this, we "bucketed" CPU core and memory resource limits for prod jobs and allocs (§2.4) by rounding them up to the next nearest power of two in each resource dimension, starting at 0.5 cores for CPU and 1 GiB for RAM. Figure 9 shows that doing so would require 30–50% more resources in the median case. The upper bound comes from allocating an entire machine to large tasks that didn't fit after quadru-

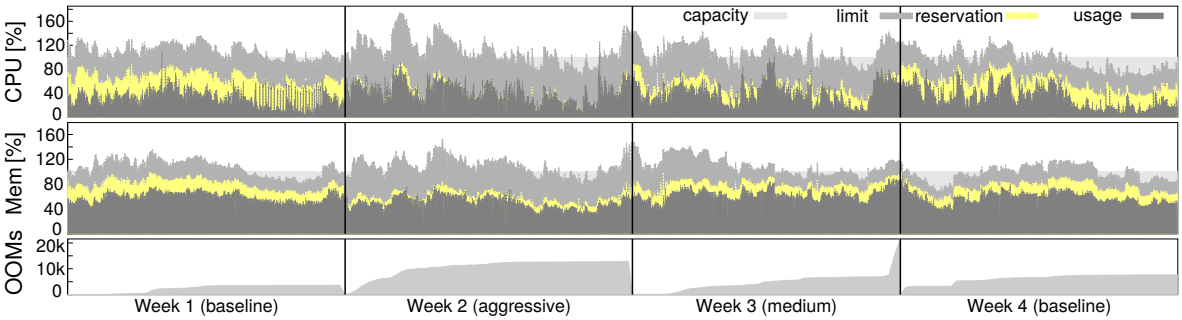


Figure 12: More aggressive resource estimation can reclaim more resources, with little effect on out-of-memory events (OOMs). *A timeline (starting on 2013-11-11) for one production cell of usage, reservation and limit averaged over 5-minute windows and cumulative out-of-memory events; the slope of the latter is the aggregate rate of OOMs. Vertical bars separate weeks with different resource estimation settings.*

pling the original cell before compaction began; the lower bound from allowing these tasks to go pending. (This is less than the roughly 100% overhead reported in [37] because we supported more than 4 buckets and permitted CPU and RAM capacity to scale independently.)

5.5 Resource reclamation

A job can specify a resource *limit* – an upper bound on the resources that each task should be granted. The limit is used by Borg to determine if the user has enough quota to admit the job, and to determine if a particular machine has enough free resources to schedule the task. Just as there are users who buy more quota than they need, there are users who request more resources than their tasks will use, because Borg will normally kill a task that tries to use more RAM or disk space than it requested, or throttle CPU to what it asked for. In addition, some tasks occasionally need to use all their resources (e.g., at peak times of day or while coping with a denial-of-service attack), but most of the time do not.

Rather than waste allocated resources that are not currently being consumed, we estimate how many resources a task will use and reclaim the rest for work that can tolerate lower-quality resources, such as batch jobs. This whole process is called *resource reclamation*. The estimate is called the task's *reservation*, and is computed by the Borgmaster every few seconds, using fine-grained *usage* (resource-consumption) information captured by the Borglet. The initial reservation is set equal to the resource request (the limit); after 300 s, to allow for startup transients, it decays slowly towards the actual usage plus a safety margin. The reservation is rapidly increased if the usage exceeds it.

The Borg scheduler uses limits to calculate feasibility (§3.2) for prod tasks,[4] so they never rely on reclaimed resources and aren't exposed to resource oversubscription; for non-prod tasks, it uses the reservations of existing tasks so the new tasks can be scheduled into reclaimed resources.

A machine may run out of resources at runtime if the reservations (predictions) are wrong – even if all tasks use less than their limits. If this happens, we kill or throttle non-prod tasks, never prod ones.

[4] To be precise, high-priority latency-sensitive ones – see §6.2.

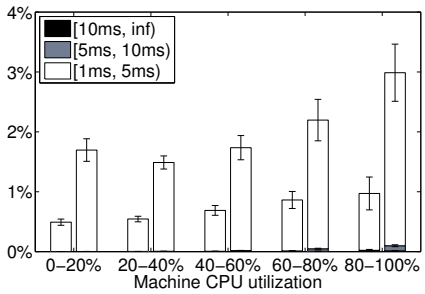


Figure 13: Scheduling delays as a function of load. *A plot of how often a runnable thread had to wait longer than 1ms to get access to a CPU, as a function of how busy the machine was. In each pair of bars, latency-sensitive tasks are on the left, batch ones on the right. In only a few percent of the time did a thread have to wait longer than 5 ms to access a CPU (the white bars); they almost never had to wait longer (the darker bars). Data from a representative cell for the month of December 2013; error bars show day-to-day variance.*

Figure 10 shows that many more machines would be required without resource reclamation. About 20% of the workload (§6.2) runs in reclaimed resources in a median cell.

We can see more details in Figure 11, which shows the ratio of reservations and usage to limits. A task that exceeds its memory limit will be the first to be preempted if resources are needed, regardless of its priority, so it is rare for tasks to exceed their memory limit. On the other hand, CPU can readily be throttled, so short-term spikes can push usage above reservation fairly harmlessly.

Figure 11 suggests that resource reclamation may be unnecessarily conservative: there is significant area between the reservation and usage lines. To test this, we picked a live production cell and adjusted the parameters of its resource estimation algorithm to an aggressive setting for a week by reducing the safety margin, and then to an medium setting that was mid-way between the baseline and aggressive settings for the next week, and then reverted to the baseline.

Figure 12 shows what happened. Reservations are clearly closer to usage in the second week, and somewhat less so in the third, with the biggest gaps shown in the baseline weeks (1st and 4th). As anticipated, the rate of out-of-memory (OOM) events increased slightly in weeks 2 and 3.[5] After reviewing these results, we decided that the net gains outweighed the downsides, and deployed the medium resource reclamation parameters to other cells.

6. Isolation

50% of our machines run 9 or more tasks; a 90%ile machine has about 25 tasks and will be running about 4500 threads [83]. Although sharing machines between applications increases utilization, it also requires good mechanisms to prevent tasks from interfering with one another. This applies to both security and performance.

6.1 Security isolation

We use a Linux chroot jail as the primary security isolation mechanism between multiple tasks on the same machine. To allow remote debugging, we used to distribute (and rescind) ssh keys automatically to give a user access to a machine only while it was running tasks for the user. For most users, this has been replaced by the borgssh command, which collaborates with the Borglet to construct an ssh connection to a shell that runs in the same chroot and cgroup as the task, locking down access even more tightly.

VMs and security sandboxing techniques are used to run external software by Google's AppEngine (GAE) [38] and Google Compute Engine (GCE). We run each hosted VM in a KVM process [54] that runs as a Borg task.

6.2 Performance isolation

Early versions of Borglet had relatively primitive resource isolation enforcement: post-hoc usage checking of memory, disk space and CPU cycles, combined with termination of tasks that used too much memory or disk and aggressive application of Linux's CPU priorities to rein in tasks that used too much CPU. But it was still too easy for rogue tasks to affect the performance of other tasks on the machine, so some users inflated their resource requests to reduce the number of tasks that Borg could co-schedule with theirs, thus decreasing utilization. Resource reclamation could claw back some of the surplus, but not all, because of the safety margins involved. In the most extreme cases, users petitioned to use dedicated machines or cells.

Now, all Borg tasks run inside a Linux cgroup-based resource container [17, 58, 62] and the Borglet manipulates the container settings, giving much improved control because the OS kernel is in the loop. Even so, occasional low-level resource interference (e.g., memory bandwidth or L3 cache pollution) still happens, as in [60, 83].

[5] The anomaly at the end of week 3 is unrelated to this experiment.

To help with overload and overcommitment, Borg tasks have an application class or *appclass*. The most important distinction is between the *latency-sensitive* (LS) appclasses and the rest, which we call *batch* in this paper. LS tasks are used for user-facing applications and shared infrastructure services that require fast response to requests. High-priority LS tasks receive the best treatment, and are capable of temporarily starving batch tasks for several seconds at a time.

A second split is between *compressible* resources (e.g., CPU cycles, disk I/O bandwidth) that are rate-based and can be reclaimed from a task by decreasing its quality of service without killing it; and *non-compressible* resources (e.g., memory, disk space) which generally cannot be reclaimed without killing the task. If a machine runs out of non-compressible resources, the Borglet immediately terminates tasks, from lowest to highest priority, until the remaining reservations can be met. If the machine runs out of compressible resources, the Borglet throttles usage (favoring LS tasks) so that short load spikes can be handled without killing any tasks. If things do not improve, Borgmaster will remove one or more tasks from the machine.

A user-space control loop in the Borglet assigns memory to containers based on predicted future usage (for prod tasks) or on memory pressure (for non-prod ones); handles Out-of-Memory (OOM) events from the kernel; and kills tasks when they try to allocate beyond their memory limits, or when an over-committed machine actually runs out of memory. Linux's eager file-caching significantly complicates the implementation because of the need for accurate memory-accounting.

To improve performance isolation, LS tasks can reserve entire physical CPU cores, which stops other LS tasks from using them. Batch tasks are permitted to run on any core, but they are given tiny scheduler shares relative to the LS tasks. The Borglet dynamically adjusts the resource caps of greedy LS tasks in order to ensure that they do not starve batch tasks for multiple minutes, selectively applying CFS bandwidth control when needed [75]; shares are insufficient because we have multiple priority levels.

Like Leverich [56], we found that the standard Linux CPU scheduler (CFS) required substantial tuning to support both low latency and high utilization. To reduce scheduling delays, our version of CFS uses extended per-cgroup load history [16], allows preemption of batch tasks by LS tasks, and reduces the scheduling quantum when multiple LS tasks are runnable on a CPU. Fortunately, many of our applications use a thread-per-request model, which mitigates the effects of persistent load imbalances. We sparingly use cpusets to allocate CPU cores to applications with particularly tight latency requirements. Some results of these efforts are shown in Figure 13. Work continues in this area, adding thread placement and CPU management that is NUMA-, hyperthreading-, and power-aware (e.g., [81]), and improving the control fidelity of the Borglet.

Tasks are permitted to consume resources up to their limit. Most of them are allowed to go beyond that for compressible resources like CPU, to take advantage of unused (slack) resources. Only 5% of LS tasks disable this, presumably to get better predictability; fewer than 1% of batch tasks do. Using slack memory is disabled by default, because it increases the chance of a task being killed, but even so, 10% of LS tasks override this, and 79% of batch tasks do so because it's a default setting of the MapReduce framework. This complements the results for reclaimed resources (§5.5). Batch tasks are willing to exploit unused as well as reclaimed memory opportunistically: most of the time this works, although the occasional batch task is sacrificed when an LS task needs resources in a hurry.

7. Related work

Resource scheduling has been studied for decades, in contexts as varied as wide-area HPC supercomputing Grids, networks of workstations, and large-scale server clusters. We focus here on only the most relevant work in the context of large-scale server clusters.

Several recent studies have analyzed cluster traces from Yahoo!, Google, and Facebook [20, 52, 63, 68, 70, 80, 82], and illustrate the challenges of scale and heterogeneity inherent in these modern datacenters and workloads. [69] contains a taxonomy of cluster manager architectures.

Apache Mesos [45] splits the resource management and placement functions between a central resource manager (somewhat like Borgmaster minus its scheduler) and multiple "frameworks" such as Hadoop [41] and Spark [73] using an offer-based mechanism. Borg mostly centralizes these functions using a request-based mechanism that scales quite well. DRF [29, 35, 36, 66] was initially developed for Mesos; Borg uses priorities and admission quotas instead. The Mesos developers have announced ambitions to extend Mesos to include speculative resource assignment and reclamation, and to fix some of the issues identified in [69].

YARN [76] is a Hadoop-centric cluster manager. Each application has a manager that negotiates for the resources it needs with a central resource manager; this is much the same scheme that Google MapReduce jobs have used to obtain resources from Borg since about 2008. YARN's resource manager only recently became fault tolerant. A related open-source effort is the *Hadoop Capacity Scheduler* [42] which provides multi-tenant support with capacity guarantees, hierarchical queues, elastic sharing and fairness. YARN has recently been extended to support multiple resource types, priorities, preemptions, and advanced admission control [21]. The Tetris research prototype [40] supports makespan-aware job packing.

Facebook's *Tupperware* [64], is a Borg-like system for scheduling cgroup containers on a cluster; only a few details have been disclosed, although it seems to provide a form of resource reclamation. Twitter has open-sourced *Aurora* [5], a Borg-like scheduler for long running services that runs on top of Mesos, with a configuration language and state machine similar to Borg's.

The *Autopilot* system from Microsoft [48] provides "automating software provisioning and deployment; system monitoring; and carrying out repair actions to deal with faulty software and hardware" for Microsoft clusters. The Borg ecosystem provides similar features, but space precludes a discussion here; Isaard [48] outlines many best practices that we adhere to as well.

Quincy [49] uses a network flow model to provide fairness- and data locality-aware scheduling for data-processing DAGs on clusters of a few hundred nodes. Borg uses quota and priorities to share resources among users and scales to tens of thousands of machines. Quincy handles execution graphs directly while this is built separately on top of Borg.

Cosmos [44] focuses on batch processing, with an emphasis on ensuring that its users get fair access to resources they have donated to the cluster. It uses a per-job manager to acquire resources; few details are publicly available.

Microsoft's *Apollo* system [13] uses per-job schedulers for short-lived batch jobs to achieve high throughput on clusters that seem to be comparably-sized to Borg cells. Apollo uses opportunistic execution of lower-priority background work to boost utilization to high levels at the cost of (sometimes) multi-day queueing delays. Apollo nodes provide a prediction matrix of starting times for tasks as a function of size over two resource dimensions, which the schedulers combine with estimates of startup costs and remote-data-access to make placement decisions, modulated by random delays to reduce collisions. Borg uses a central scheduler for placement decisions based on state about prior allocations, can handle more resource dimensions, and focuses on the needs of high-availability, long-running applications; Apollo can probably handle a higher task arrival rate.

Alibaba's *Fuxi* [84] supports data-analysis workloads; it has been running since 2009. Like Borgmaster, a central FuxiMaster (replicated for failure-tolerance) gathers resource-availability information from nodes, accepts requests from applications, and matches one to the other. The Fuxi incremental scheduling policy is the inverse of Borg's equivalence classes: instead of matching each task to one of a suitable set of machines, Fuxi matches newly-available resources against a backlog of pending work. Like Mesos, Fuxi allows "virtual resource" types to be defined. Only synthetic workload results are publicly available.

Omega [69] supports multiple parallel, specialized "verticals" that are each roughly equivalent to a Borgmaster minus its persistent store and link shards. Omega schedulers use optimistic concurrency control to manipulate a shared representation of desired and observed cell state stored in a central persistent store, which is synced to/from the Borglets by a separate link component. The Omega architecture was designed to support multiple distinct workloads that have their

own application-specific RPC interface, state machines, and scheduling policies (e.g., long-running servers, batch jobs from various frameworks, infrastructure services like cluster storage systems, virtual machines from the Google Cloud Platform). On the other hand, Borg offers a "one size fits all" RPC interface, state machine semantics, and scheduler policy, which have grown in size and complexity over time as a result of needing to support many disparate workloads, and scalability has not yet been a problem (§3.4).

Google's open-source *Kubernetes* system [53] places applications in Docker containers [28] onto multiple host nodes. It runs both on bare metal (like Borg) and on various cloud hosting providers, such as Google Compute Engine. It is under active development by many of the same engineers who built Borg. Google offers a hosted version called Google Container Engine [39]. We discuss how lessons from Borg are being applied to Kubernetes in the next section.

The high-performance computing community has a long tradition of work in this area (e.g., Maui, Moab, Platform LSF [2, 47, 50]); however the requirements of scale, workloads and fault tolerance are different from those of Google's cells. In general, such systems achieve high utilization by having large backlogs (queues) of pending work.

Virtualization providers such as VMware [77] and datacenter solution providers such as HP and IBM [46] provide cluster management solutions that typically scale to O(1000) machines. In addition, several research groups have prototyped systems that improve the quality of scheduling decisions in certain ways (e.g., [25, 40, 72, 74]).

And finally, as we have indicated, another important part of managing large scale clusters is automation and "operator scaleout". [43] describes how planning for failures, multitenancy, health checking, admission control, and restartability are necessary to achieve high numbers of machines per operator. Borg's design philosophy is similar and allows us to support tens of thousands of machines per operator (SRE).

8. Lessons and future work

In this section we recount some of the qualitative lessons we've learned from operating Borg in production for more than a decade, and describe how these observations have been leveraged in designing Kubernetes [53].

8.1 Lessons learned: the bad

We begin with some features of Borg that serve as cautionary tales, and informed alternative designs in Kubernetes.

Jobs are restrictive as the only grouping mechanism for tasks. Borg has no first-class way to manage an entire multi-job service as a single entity, or to refer to related instances of a service (e.g., canary and production tracks). As a hack, users encode their service topology in the job name and build higher-level management tools to parse these names. At the other end of the spectrum, it's not possible to refer to arbitrary subsets of a job, which leads to problems like inflexible semantics for rolling updates and job resizing.

To avoid such difficulties, Kubernetes rejects the job notion and instead organizes its scheduling units (pods) using *labels* – arbitrary key/value pairs that users can attach to any object in the system. The equivalent of a Borg job can be achieved by attaching a job:*jobname* label to a set of pods, but any other useful grouping can be represented too, such as the service, tier, or release-type (e.g., production, staging, test). Operations in Kubernetes identify their targets by means of a label query that selects the objects that the operation should apply to. This approach gives more flexibility than the single fixed grouping of a job.

One IP address per machine complicates things. In Borg, all tasks on a machine use the single IP address of their host, and thus share the host's port space. This causes a number of difficulties: Borg must schedule ports as a resource; tasks must pre-declare how many ports they need, and be willing to be told which ones to use when they start; the Borglet must enforce port isolation; and the naming and RPC systems must handle ports as well as IP addresses.

Thanks to the advent of Linux namespaces, VMs, IPv6, and software-defined networking, Kubernetes can take a more user-friendly approach that eliminates these complications: every pod and service gets its own IP address, allowing developers to choose ports rather than requiring their software to adapt to the ones chosen by the infrastructure, and removes the infrastructure complexity of managing ports.

Optimizing for power users at the expense of casual ones. Borg provides a large set of features aimed at "power users" so they can fine-tune the way their programs are run (the BCL specification lists about 230 parameters): the initial focus was supporting the largest resource consumers at Google, for whom efficiency gains were paramount. Unfortunately the richness of this API makes things harder for the "casual" user, and constrains its evolution. Our solution has been to build automation tools and services that run on top of Borg, and determine appropriate settings from experimentation. These benefit from the freedom to experiment afforded by failure-tolerant applications: if the automation makes a mistake it is a nuisance, not a disaster.

8.2 Lessons learned: the good

On the other hand, a number of Borg's design features have been remarkably beneficial and have stood the test of time.

Allocs are useful. The Borg alloc abstraction spawned the widely-used logsaver pattern (§2.4) and another popular one in which a simple data-loader task periodically updates the data used by a web server. Allocs and packages allow such helper services to be developed by separate teams. The Kubernetes equivalent of an alloc is the *pod*, which is a resource envelope for one or more containers that are always scheduled onto the same machine and can share resources. Kubernetes uses helper containers in the same pod instead of tasks in an alloc, but the idea is the same.

Cluster management is more than task management. Although Borg's primary role is to manage the lifecycles of tasks and machines, the applications that run on Borg benefit from many other cluster services, including naming and load balancing. Kubernetes supports naming and load balancing using the *service* abstraction: a service has a name and a dynamic set of pods defined by a label selector. Any container in the cluster can connect to the service using the service name. Under the covers, Kubernetes automatically load-balances connections to the service among the pods that match the label selector, and keeps track of where the pods are running as they get rescheduled over time due to failures.

Introspection is vital. Although Borg almost always "just works," when something goes wrong, finding the root cause can be challenging. An important design decision in Borg was to surface debugging information to all users rather than hiding it: Borg has thousands of users, so "self-help" has to be the first step in debugging. Although this makes it harder for us to deprecate features and change internal policies that users come to rely on, it is still a win, and we've found no realistic alternative. To handle the enormous volume of data, we provide several levels of UI and debugging tools, so users can quickly identify anomalous events related to their jobs, and then drill down to detailed event and error logs from their applications and the infrastructure itself.

Kubernetes aims to replicate many of Borg's introspection techniques. For example, it ships with tools such as cAdvisor [15] for resource monitoring, and log aggregation based on Elasticsearch/Kibana [30] and Fluentd [32]. The master can be queried for a snapshot of its objects' state. Kubernetes has a unified mechanism that all components can use to record events (e.g., a pod being scheduled, a container failing) that are made available to clients.

The master is the kernel of a distributed system. Borgmaster was originally designed as a monolithic system, but over time, it became more of a kernel sitting at the heart of an ecosystem of services that cooperate to manage user jobs. For example, we split off the scheduler and the primary UI (Sigma) into separate processes, and added services for admission control, vertical and horizontal autoscaling, re-packing tasks, periodic job submission (cron), workflow management, and archiving system actions for off-line querying. Together, these have allowed us to scale up the workload and feature set without sacrificing performance or maintainability.

The Kubernetes architecture goes further: it has an API server at its core that is responsible only for processing requests and manipulating the underlying state objects. The cluster management logic is built as small, composable micro-services that are clients of this API server, such as the replication controller, which maintains the desired number of replicas of a pod in the face of failures, and the node controller, which manages the machine lifecycle.

8.3 Conclusion

Virtually all of Google's cluster workloads have switched to use Borg over the past decade. We continue to evolve it, and have applied the lessons we learned from it to Kubernetes.

Acknowledgments

The authors of this paper performed the evaluations and wrote the paper, but the dozens of engineers who designed, implemented, and maintained Borg's components and ecosystem are the key to its success. We list here just those who participated most directly in the design, implementation, and operation of the Borgmaster and Borglets. Our apologies if we missed anybody.

The initial Borgmaster was primarily designed and implemented by Jeremy Dion and Mark Vandevoorde, with Ben Smith, Ken Ashcraft, Maricia Scott, Ming-Yee Iu, and Monika Henzinger. The initial Borglet was primarily designed and implemented by Paul Menage.

Subsequent contributors include Abhishek Rai, Abhishek Verma, Andy Zheng, Ashwin Kumar, Beng-Hong Lim, Bin Zhang, Bolu Szewczyk, Brian Budge, Brian Grant, Brian Wickman, Chengdu Huang, Cynthia Wong, Daniel Smith, Dave Bort, David Oppenheimer, David Wall, Dawn Chen, Eric Haugen, Eric Tune, Ethan Solomita, Gaurav Dhiman, Geeta Chaudhry, Greg Roelofs, Grzegorz Czajkowski, James Eady, Jarek Kusmierek, Jaroslaw Przybylowicz, Jason Hickey, Javier Kohen, Jeremy Lau, Jerzy Szczepkowski, John Wilkes, Jonathan Wilson, Joso Eterovic, Jutta Degener, Kai Backman, Kamil Yurtsever, Kenji Kaneda, Kevan Miller, Kurt Steinkraus, Leo Landa, Liza Fireman, Madhukar Korupolu, Mark Logan, Markus Gutschke, Matt Sparks, Maya Haridasan, Michael Abd-El-Malek, Michael Kenniston, Mukesh Kumar, Nate Calvin, Onufry Wojtaszczyk, Patrick Johnson, Pedro Valenzuela, Piotr Witusowski, Praveen Kallakuri, Rafal Sokolowski, Richard Gooch, Rishi Gosalia, Rob Radez, Robert Hagmann, Robert Jardine, Robert Kennedy, Rohit Jnagal, Roy Bryant, Rune Dahl, Scott Garriss, Scott Johnson, Sean Howarth, Sheena Madan, Smeeta Jalan, Stan Chesnutt, Temo Arobelidze, Tim Hockin, Todd Wang, Tomasz Blaszczyk, Tomasz Wozniak, Tomek Zielonka, Victor Marmol, Vish Kannan, Vrigo Gokhale, Walfredo Cirne, Walt Drummond, Weiran Liu, Xiaopan Zhang, Xiao Zhang, Ye Zhao, and Zohaib Maya.

The Borg SRE team has also been crucial, and has included Adam Rogoyski, Alex Milivojevic, Anil Das, Cody Smith, Cooper Bethea, Folke Behrens, Matt Liggett, James Sanford, John Millikin, Matt Brown, Miki Habryn, Peter Dahl, Robert van Gent, Seppi Wilhelmi, Seth Hettich, Torsten Marek, and Viraj Alankar. The Borg configuration language (BCL) and borgcfg tool were originally developed by Marcel van Lohuizen and Robert Griesemer.

We thank our reviewers (especially Eric Brewer, Malte Schwarzkopf and Tom Rodeheffer), and our shepherd, Christos Kozyrakis, for their feedback on this paper.

References

[1] O. A. Abdul-Rahman and K. Aida. Towards understanding the usage behavior of Google cloud users: the mice and elephants phenomenon. In *Proc. IEEE Int'l Conf. on Cloud Computing Technology and Science (CloudCom)*, pages 272–277, Singapore, Dec. 2014.

[2] Adaptive Computing Enterprises Inc., Provo, UT. *Maui Scheduler Administrator's Guide*, 3.2 edition, 2011.

[3] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: fault-tolerant stream processing at internet scale. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pages 734–746, Riva del Garda, Italy, Aug. 2013.

[4] Y. Amir, B. Awerbuch, A. Barak, R. S. Borgstrom, and A. Keren. An opportunity cost approach for job assignment in a scalable computing cluster. *IEEE Trans. Parallel Distrib. Syst.*, 11(7):760–768, July 2000.

[5] Apache Aurora. `http://aurora.incubator.apache.org/`, 2014.

[6] Aurora Configuration Tutorial. `https://aurora.incubator.apache.org/documentation/latest/configuration-tutorial/`, 2014.

[7] AWS. Amazon Web Services VM Instances. `http://aws.amazon.com/ec2/instance-types/`, 2014.

[8] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. Conference on Innovative Data Systems Research (CIDR)*, pages 223–234, Asilomar, CA, USA, Jan. 2011.

[9] M. Baker and J. Ousterhout. Availability in the Sprite distributed file system. *Operating Systems Review*, 25(2):95–98, Apr. 1991.

[10] L. A. Barroso, J. Clidaras, and U. Hölzle. *The datacenter as a computer: an introduction to the design of warehouse-scale machines*. Morgan Claypool Publishers, 2nd edition, 2013.

[11] L. A. Barroso, J. Dean, and U. Holzle. Web search for a planet: the Google cluster architecture. In *IEEE Micro*, pages 22–28, 2003.

[12] I. Bokharouss. GCL Viewer: a study in improving the understanding of GCL programs. Technical report, Eindhoven Univ. of Technology, 2008. MS thesis.

[13] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, Oct. 2014.

[14] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 335–350, Seattle, WA, USA, 2006.

[15] cAdvisor. `https://github.com/google/cadvisor`, 2014.

[16] CFS per-entity load patches. `http://lwn.net/Articles/531853`, 2013.

[17] cgroups. `http://en.wikipedia.org/wiki/Cgroups`, 2014.

[18] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 363–375, Toronto, Ontario, Canada, 2010.

[19] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. *ACM Trans. on Computer Systems*, 26(2):4:1–4:26, June 2008.

[20] Y. Chen, S. Alspaugh, and R. H. Katz. Design insights for MapReduce from diverse production workloads. Technical Report UCB/EECS–2012–17, UC Berkeley, Jan. 2012.

[21] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based scheduling: if you're late don't blame us! In *Proc. ACM Symp. on Cloud Computing (SoCC)*, pages 2:1–2:14, Seattle, WA, USA, 2014.

[22] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, Feb. 2012.

[23] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[24] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 201.

[25] C. Delimitrou and C. Kozyrakis. Quasar: resource-efficient and QoS-aware cluster management. In *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 127–144, Salt Lake City, UT, USA, 2014.

[26] S. Di, D. Kondo, and W. Cirne. Characterization and comparison of cloud versus Grid workloads. In *International Conference on Cluster Computing (IEEE CLUSTER)*, pages 230–238, Beijing, China, Sept. 2012.

[27] S. Di, D. Kondo, and C. Franck. Characterizing cloud applications on a Google data center. In *Proc. Int'l Conf. on Parallel Processing (ICPP)*, Lyon, France, Oct. 2013.

[28] Docker Project. `https://www.docker.io/`, 2014.

[29] D. Dolev, D. G. Feitelson, J. Y. Halpern, R. Kupferman, and N. Linial. No justified complaints: on fair sharing of multiple resources. In *Proc. Innovations in Theoretical Computer Science (ITCS)*, pages 68–75, Cambridge, MA, USA, 2012.

[30] ElasticSearch. `http://www.elasticsearch.org`, 2014.

[31] D. G. Feitelson. *Workload Modeling for Computer Systems Performance Evaluation*. Cambridge University Press, 2014.

[32] Fluentd. `http://www.fluentd.org/`, 2014.

[33] GCE. Google Compute Engine. `http://cloud.google.com/products/compute-engine/`, 2014.

[34] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proc. ACM Symp. on Operating Systems Principles (SOSP)*, pages 29–43, Bolton Landing, NY, USA, 2003. ACM.

[35] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: fair allocation of multiple resource types. In *Proc. USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 323–326, 2011.

[36] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: max-min fair sharing for datacenter jobs with constraints. In *Proc. European Conf. on Computer Systems (EuroSys)*, pages 365–378, Prague, Czech Republic, 2013.

[37] D. Gmach, J. Rolia, and L. Cherkasova. Selling T-shirts and time shares in the cloud. In *Proc. IEEE/ACM Int'l Symp. on Cluster, Cloud and Grid Computing (CCGrid)*, pages 539–546, Ottawa, Canada, 2012.

[38] Google App Engine. `http://cloud.google.com/AppEngine`, 2014.

[39] Google Container Engine (GKE). `https://cloud.google.com/container-engine/`, 2015.

[40] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *Proc. ACM SIGCOMM*, Aug. 2014.

[41] Apache Hadoop Project. `http://hadoop.apache.org/`, 2009.

[42] Hadoop MapReduce Next Generation – Capacity Scheduler. `http://hadoop.apache.org/docs/r2.2.0/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html`, 2013.

[43] J. Hamilton. On designing and deploying internet-scale services. In *Proc. Large Installation System Administration Conf. (LISA)*, pages 231–242, Dallas, TX, USA, Nov. 2007.

[44] P. Helland. Cosmos: big data and big challenges. `http://research.microsoft.com/en-us/events/fs2011/helland\_cosmos\_big\_data\_and\_big\_challenges.pdf`, 2011.

[45] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *Proc. USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, 2011.

[46] IBM Platform Computing. `http://www-03.ibm.com/systems/technicalcomputing/platformcomputing/products/clustermanager/index.html`.

[47] S. Iqbal, R. Gupta, and Y.-C. Fang. Planning considerations for job scheduling in HPC clusters. *Dell Power Solutions*, Feb. 2005.

[48] M. Isaard. Autopilot: Automatic data center management. *ACM SIGOPS Operating Systems Review*, 41(2), 2007.

[49] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proc. ACM Symp. on Operating Systems Principles (SOSP)*, 2009.

[50] D. B. Jackson, Q. Snell, and M. J. Clement. Core algorithms of the Maui scheduler. In *Proc. Int'l Workshop on Job Scheduling Strategies for Parallel Processing*, pages 87–102. Springer-Verlag, 2001.

[51] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim. Measuring interference between live datacenter applications. In *Proc. Int'l Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, Salt Lake City, UT, Nov. 2012.

[52] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An analysis of traces from a production MapReduce cluster. In *Proc. IEEE/ACM Int'l Symp. on Cluster, Cloud and Grid Computing (CCGrid)*, pages 94–103, 2010.

[53] Kubernetes. `http://kubernetes.io`, Aug. 2014.

[54] Kernel Based Virtual Machine. `http://www.linux-kvm.org`.

[55] L. Lamport. The part-time parliament. *ACM Trans. on Computer Systems*, 16(2):133–169, May 1998.

[56] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proc. European Conf. on Computer Systems (EuroSys)*, page 4, 2014.

[57] Z. Liu and S. Cho. Characterizing machines and workloads on a Google cluster. In *Proc. Int'l Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS)*, Pittsburgh, PA, USA, Sept. 2012.

[58] Google LMCTFY project (let me contain that for you). `http://github.com/google/lmctfy`, 2014.

[59] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. ACM SIGMOD Conference*, pages 135–146, Indianapolis, IA, USA, 2010.

[60] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-Up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proc. Int'l Symp. on Microarchitecture (Micro)*, Porto Alegre, Brazil, 2011.

[61] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pages 330–339, Singapore, Sept. 2010.

[62] P. Menage. Linux control groups. `http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt`, 2007–2014.

[63] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das. Towards characterizing cloud backend workloads: insights from Google compute clusters. *ACM SIGMETRICS Performance Evaluation Review*, 37:34–41, Mar. 2010.

[64] A. Narayanan. Tupperware: containerized deployment at Facebook. `http://www.slideshare.net/dotCloud/tupperware-containerized-deployment-at-facebook`, June 2014.

[65] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proc. ACM Symp. on Operating Systems Principles (SOSP)*, pages 69–84, Farminton, PA, USA, 2013.

[66] D. C. Parkes, A. D. Procaccia, and N. Shah. Beyond Dominant Resource Fairness: extensions, limitations, and indivisibilities. In *Proc. Electronic Commerce*, pages 808–825, Valencia, Spain, 2012.

[67] Protocol buffers. `https://developers.google.com/protocol-buffers/`, and `https://github.com/google/protobuf/`., 2014.

[68] C. Reiss, A. Tumanov, G. Ganger, R. Katz, and M. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proc. ACM Symp. on Cloud Computing (SoCC)*, San Jose, CA, USA, Oct. 2012.

[69] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proc. European Conf. on Computer Systems (EuroSys)*, Prague, Czech Republic, 2013.

[70] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das. Modeling and synthesizing task placement constraints in Google compute clusters. In *Proc. ACM Symp. on Cloud Computing (SoCC)*, pages 3:1–3:14, Cascais, Portugal, Oct. 2011.

[71] E. Shmueli and D. G. Feitelson. On simulation and design of parallel-systems schedulers: are we doing the right thing? *IEEE Trans. on Parallel and Distributed Systems*, 20(7):983–996, July 2009.

[72] A. Singh, M. Korupolu, and D. Mohapatra. Server-storage virtualization: integration and load balancing in data centers. In *Proc. Int'l Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 53:1–53:12, Austin, TX, USA, 2008.

[73] Apache Spark Project. `http://spark.apache.org/`, 2014.

[74] A. Tumanov, J. Cipar, M. A. Kozuch, and G. R. Ganger. Alsched: algebraic scheduling of mixed workloads in heterogeneous clouds. In *Proc. ACM Symp. on Cloud Computing (SoCC)*, San Jose, CA, USA, Oct. 2012.

[75] P. Turner, B. Rao, and N. Rao. CPU bandwidth control for CFS. In *Proc. Linux Symposium*, pages 245–254, July 2010.

[76] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proc. ACM Symp. on Cloud Computing (SoCC)*, Santa Clara, CA, USA, 2013.

[77] VMware VCloud Suite. `http://www.vmware.com/products/vcloud-suite/`.

[78] A. Verma, M. Korupolu, and J. Wilkes. Evaluating job packing in warehouse-scale computing. In *IEEE Cluster*, pages 48–56, Madrid, Spain, Sept. 2014.

[79] W. Whitt. Open and closed models for networks of queues. *AT&T Bell Labs Technical Journal*, 63(9), Nov. 1984.

[80] J. Wilkes. More Google cluster data. `http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html`, Nov. 2011.

[81] Y. Zhai, X. Zhang, S. Eranian, L. Tang, and J. Mars. HaPPy: Hyperthread-aware power profiling dynamically. In *Proc. USENIX Annual Technical Conf. (USENIX ATC)*, pages 211–217, Philadelphia, PA, USA, June 2014. USENIX Association.

[82] Q. Zhang, J. Hellerstein, and R. Boutaba. Characterizing task usage shapes in Google's compute clusters. In *Proc. Int'l Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, 2011.

[83] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *Proc. European Conf. on Computer Systems (EuroSys)*, Prague, Czech Republic, 2013.

[84] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pages 1393–1404. VLDB Endowment Inc., Sept. 2014.

Large-scale cluster management at Google with Borg: errata

2015-04-23

After the camera-ready copy was finalized, we noticed a few inadvertent omissions and ambiguities.

The user perspective

SREs do much more than system administration: they are the engineers responsible for Google's production services. They design and implement software, including automation systems, and manage applications, service infrastructure and platforms to ensure high performance and reliability at Google scale.

Related work

Borg leveraged much from its internal predecessor, the Global Work Queue system, which was initially developed by Jeff Dean, Olcan Sercinoglu, and Percy Liang.

Condor [1] has been widely used for aggregating collections of idle resources, and its ClassAds mechanism [2] supports declarative statements and automated attribute/property matching.

Acknowledgements

We accidentally omitted Brad Strand, Chris Colohan, Divyesh Shah, Eric Wilcox, and Pavanish Nirula.

References

[1] Michael Litzkow, Miron Livny, and Matt Mutka. "Condor - A Hunter of Idle Workstations". In *Proc. Int'l Conf. on Distributed Computing Systems (ICDCS)*, pages 104-111, June 1988.

[2] Rajesh Raman, Miron Livny, and Marvin Solomon. "Matchmaking: Distributed Resource Management for High Throughput Computing". In *Proc. Int'l Symp. on High Performance Distributed Computing (HPDC)*, Chicago, IL, USA, July 1998.

arXiv:2408.03714v1 [cs.CR] 7 Aug 2024

Centralized Defense: Logging and Mitigation of Kubernetes Misconfigurations with Open Source Tools

Eoghan Russell[a,b], Kapal Dev[b]

[a] *Intel, Ireland*
[b] *Department of Computer Science, Munster Technological University,Ireland*

Abstract

Kubernetes, an open-source platform for automating the deployment, scaling, and management of containerized applications, is widely used for its efficiency and scalability. However, its complexity and extensive configuration options often lead to security vulnerabilities if not managed properly. This paper presents a detailed analysis of misconfigurations in Kubernetes environments and their significant impact on system reliability and security. A centralized logging solution was developed to detect such misconfigurations, detailing the integration process with a Kubernetes cluster and the implementation of role-based access control. Utilizing a combination of open-source tools, the solution systematically identifies misconfigurations and aggregates diagnostic data into a central repository. The effectiveness of the solution was evaluated using specific metrics, such as the total cycle time for running the central logging solution against the individual open source tools.

Keywords: Central Logging Solution, Cloud Native Computing Foundation, Amazon Web Services, Application Programming Interface, Command Line Interface, Internet Protocol, Media Access Control, Domain Name System, Common Vulnerabilities Exposures, Information Technology, Role-Based Access Control

1. Introduction

Kubernetes, an open-source platform for automating the deployment, scaling, and management of containerized applications, is widely used for its efficiency and scalability. However, its complexity and extensive configuration options often lead to security vulnerabilities if not managed properly. Common issues in Kubernetes include excessive permissions, misconfigured network policies, and insecure default settings, all of which can create potential entry points for attackers. Ensuring robust security practices and continuous monitoring is essential to maintaining the integrity and reliability of Kubernetes clusters. Among these challenges, misconfigurations are a significant concern that can expose cloud-native applications to severe security threats.

While many open-source tools are available to identify misconfigurations within Kubernetes manifest files, a central logging solution does not exist. This paper proposes such a central logging solution which identifies misconfigurations and provides advisory defensive solutions regarding the misconfigurations identified. The results are then presented to a cluster administrator via the home page of the central logging solution.

The need for this central logging solution is motivated by the awareness that Kubernetes holds a pivotal position in the deployment of cloud-native applications and the serious risks posed by misconfigurations in Kubernetes manifest files. This awareness is informed by this author's daily usage of Kubernetes providing firsthand knowledge of the consequences of misconfigurations, the vulnerabilities they might introduce, to the avenues they open for potential system breaches by attackers.

Email addresses: `eoghanrussell1212@gmail.com` (Eoghan Russell), `kapal.dev@ieee.org` (Kapal Dev)

This paper proposes to identify the vulnerabilities associated with common misconfigurations and their potential impact on the Kubernetes environment. Additionally, it will determine the efficacy of the most popular open-source tools which specialise in the detection of Kubernetes misconfigurations. The proposed central logging solution will be evaluated for how effectively it aids in the mitigation of common security misconfigurations and its ability to seamlessly integrate with popular open-source tools. The delta in latency compared to running the individual open-source tools will be determined against viewing the results from the centralised logging solution.

This paper delves into Kubernetes security, focusing on how misconfigurations can create vulnerabilities within a Kubernetes environment. Section 1 introduces the paper, highlighting the security risks of Kubernetes misconfigurations, the lack of a central logging solution, and the research objectives this paper aims to address. Section 2 explores security vulnerabilities within Kubernetes, including real-world attack scenarios. It examines how Kubernetes misconfigurations can lead to significant issues and their link to security breaches, concluding with a review of open-source tools for detecting these misconfigurations. Section 3 explains the architecture of the proposed central logging system. Section 4 details the implementation of the central logging system using the selected tools. It evaluates the logging solution by analyzing metrics such as execution time from start to finish, frequency of reruns, and effectiveness of comparing outputs from different tools. Additionally, it offers a comparative analysis of the tools based on personal insights, highlighting their benefits and drawbacks. Finally, Section 5 concludes the paper by reviewing the implemented solution, discussing potential areas for further improvement, and providing a final observation.

The primary contributions of this paper are the development of a centralized logging solution to detect Kubernetes misconfigurations, the integration of multiple open-source tools to enhance the detection capabilities, the evaluation of the solution's effectiveness in real-world scenarios, and a comparative analysis of various open-source tools for identifying Kubernetes misconfigurations.

2. Literature Review

Academic research has highlighted security concerns with Kubernetes, posing risks to its adoption despite its advantages as a lightweight, efficient, and cost-effective solution [1]. Enterprises have sought to identify and address these concerns after several high-profile attacks have occurred. According to a survey conducted by CNCF in 2022, lack of user training and security of the Kubernetes environment are the top challenges faced when deploying containers, 40% of the 2063 respondents reported that security is their biggest challenge in production environments [2]. A StackRox survey conducted in 2020 suggests that 44% of organisations delay deployment due to security concerns [3]. The results also states that 94% of organisations experienced a security incident within the previous 12 months [3]. Two attacks which were highly reported by the news media occurred to Tesla and Capital One, both exposed vulnerabilities in the system structure. In 2018, Tesla experienced an intrusion due to a lack of security best practices, the Kubernetes console was not password-protected and malicious users were able to gain AWS credentials that were exposed in a container [4]. At Capital One, an attack on a Kubernetes cluster was found to have been caused by the misconfiguration of the firewall, allowing an attacker to query internal metadata [5].

A paper proposed by Shamim et al. discusses the growing concern of security vulnerabilities within a Kubernetes environment, through a systematic analysis of internet artifacts that led to the identification and categorization of 11 essential security practices. Their research identified key measures such as robust authentication, authorization, and continuous updates as critical to enhancing the security posture of Kubernetes deployments. Moreover, their research highlights the importance of a comprehensive, informed approach to safeguard sensitive data and infrastructure against evolving cyber threats, providing a crucial roadmap for practitioners aiming to mitigate vulnerabilities within these complex containerised infrastructures [6].

Yang et al. provides a crucial examination of security vulnerabilities in Kubernetes, pinpointing how excessive permissions granted to third-party applications can lead to critical vulnerabilities, enabling attackers to potentially seize control of entire Kubernetes clusters. Their investigation, covering all third-party applications listed by the Cloud Native Computing Foundation (CNCF) and services provided by leading cloud vendors, uncovers that a significant portion of these applications carry inherent risks due to overly

broad permissions. The study's impact is highlighted by the identification of eight new Common Vulnerabilities and Exposures (CVEs), demonstrating the serious security implications and underscoring the need for rigorous permission controls within the Kubernetes ecosystem to mitigate potential attack vectors effectively [7].

Initial Access	Execution	Persistence	Privilege Escalation	Defense Evasion	Credential Access	Discovery	Lateral Movement	Impact
Using Cloud credentials	Exec into container	Backdoor container	Privileged container	Clear container logs	List K8S secrets	Access the K8S API server	Access cloud resources	Data Destruction
Compromised images in registry	bash/cmd inside container	Writable hostPath mount	Cluster-admin binding	Delete K8S events	Mount service principal	Access Kubelet API	Container service account	Resource Hijacking
Kubeconfig file	New container	Kubernetes CronJob	hostPath mount	Pod / container name similarity	Access container service account	Network mapping	Cluster internal networking	Denial of service
Application vulnerability	Application exploit (RCE)		Access cloud resources	Connect from Proxy server	Applications credentials in configuration files	Access Kubernetes dashboard	Applications credentials in configuration files	
Exposed Dashboard	SSH server running inside container					Instance Metadata API	Writable volume mounts on the host	
							Access Kubernetes dashboard	
							Access tiller endpoint	

Figure 1: The Kubernetes attack matrix enumerates various methods attackers employ to accomplish the nine tactics specified in the Mitre ATT&CK framework [8]: Initial Access, Execution, Persistence, Privilege Escalation, Defense Evasion, Credential Access, Discovery, Lateral Movement, and Impact. [9]

The Kubernetes Threat Matrix, developed by the Azure Security Centre and based on the Mitre ATT&CK framework [8], outlines the complex vulnerabilities of Kubernetes clusters [9]. This matrix is a pivotal resource, offering a structured analysis of potential cybersecurity threats and the corresponding tactics attackers employ to infiltrate and compromise Kubernetes environments. Each tactic outlined within the matrix corresponds to a phase in the attack life cycle, from initial access to eventual impact, providing a comprehensive view of the attack vectors that adversaries might exploit. By leveraging the Mitre ATT&CK framework, a well-regarded knowledge base of cyber attack techniques, the Kubernetes Threat Matrix offers invaluable insights into securing Kubernetes clusters against a wide array of cybersecurity threats.

Initial Access: This tactic involves an attacker obtaining access to the cluster through various means, such as exploiting legitimate credentials, exploiting vulnerabilities in exposed endpoints, or compromising container images and inducing their execution within the cluster.

Execution: Following access, the attacker executes malicious activities inside the cluster. This may involve the creation of a malicious container or executing commands within an existing container, using tools like `kubectl exec`, assuming the necessary permissions are obtained.

Persistence: This tactic ensures the attacker maintains access to the cluster even after initial entry points are closed. Techniques include deploying containers with embedded malicious code, known as backdoor containers, to facilitate re-entry.

Privilege Escalation: Attackers enhance their access levels to exploit more resources and sensitive information. This could involve accessing a container with elevated privileges or exploiting Role-Based Access Control (RBAC) to assume roles like cluster-admin, granting extensive control over cluster resources.

Defense Evasion: Attackers employ methods to conceal their presence, such as erasing logs or camouflaging malicious containers with conventional naming schemes to blend into the cluster environment.

Credential Access: This involves attackers targeting and stealing credentials stored within the cluster, like Kubernetes Secrets, to access sensitive information and further exploit the cluster.

Discovery: Attackers explore the cluster to understand its layout and identify targets. This may involve querying the API Server, Kubelet API, Kubernetes Dashboard, or using the internal network communications between pods.

Lateral Movement: With knowledge of the cluster's architecture, attackers move within the cluster to reach their objectives. Techniques for discovery can also facilitate movement, leveraging the cluster's internal communications or administrative interfaces.

Impact: The ultimate goal of the attacker is to disrupt the cluster's normal operations negatively. This could involve data deletion, configuration changes, using the cluster for cryptocurrency mining, or executing a denial of service (DoS) attack to deny legitimate users access to services.

CVE	**Cause**	**Effect**	**Consequences**
2014-0047	Unprotected Resources	Exposed System	Gain Information
2014-3499	Incorrect Permission Management	Exposed System	Gain Privileges
2014-5277	Incorrect Recovery Mechanism	Restricted Violation	Gain Information
2014-5279	Unprotected Resources	Exposed System	Gain Privileges
2014-5280	Incorrect Permission Management	Restricted Violation	Execute code
2014-8179	Incorrect Configuration	Exposed System	Execute code
2014-5282	Improper Validation	Exposed System	Bypass
2014-6407	Incorrect Permission Management	Restriction Violation	Execute code
2023-30512	cfs-csi-cluster-role	Privilege Escalation	Access to entire cluster
2020-8559	Unvalidated redirect on proxied upgrade requests	Privilege Escalation	Full cluster compromise

Table 1: Common vulnerabilities, causes, effects, and consequences [10]

Table 1 presents an overview of common vulnerabilities, with a focus on the implications of misconfigurations within Kubernetes environments. The table categorizes these vulnerabilities by their Common Vulnerabilities and Exposures (CVE) identifiers, highlighting the causes, such as unprotected resources, incorrect permission management, and improper validation, which are prevalent in Kubernetes configurations. The effects and consequences range from exposed systems and restriction violations to full cluster compromise, emphasizing the critical impact misconfigurations can have on the security and integrity of Kubernetes clusters. Notably, vulnerabilities like 'cfs-csi-cluster-role' and 'Unvalidated redirect on proxied upgrade requests' directly illustrate how specific misconfigurations in Kubernetes can lead to severe consequences, including privilege escalation and complete cluster compromise.

Echoing the concerns highlighted in the table, the Red Hat 2023 Report reinforces the notion that misconfiguration remains a pressing challenge within Kubernetes environments, with 28% of users citing misconfigurations and vulnerabilities as their top concerns [11]. This apprehension is largely attributed to Kubernetes' extensive customization capabilities, which, while beneficial, introduce significant risk factors. Human error is also behind the vast majority of security breaches, which leads to the misconfigurations of production environments [12].

In the pursuit of fortifying Kubernetes deployments against security vulnerabilities, academic endeavors have meticulously explored the avenues through which system misconfigurations could be preemptively

identified and rectified. Rahman et al.'s seminal work [13] stands at the forefront of this exploration, offering an empirical study that delves into the intricacies of security misconfigurations within Kubernetes manifests. By scrutinizing 2,039 Kubernetes manifest files sourced from 92 open-source repositories, they meticulously categorized various forms of security misconfigurations, highlighting the paramount importance of addressing these vulnerabilities to safeguard against potential security threats. Their research not only shed light on the prevalence of such misconfigurations but also led to the development of SLI-KUBE, a pioneering static analysis tool designed to quantitatively assess the frequency of these misconfigurations [13].

This paper builds upon Rahman et al.'s foundational work, aiming to extend the discourse beyond the identification of misconfigurations by proposing a centralized logging solution. Such a solution is envisioned to seamlessly integrate with the Kubernetes environment, offering a comprehensive mechanism to alert system administrators of potential vulnerabilities across various layers of the Kubernetes stack. At the core of Kubernetes, pods represent the most fundamental unit, encapsulating one or more containers that collectively embody an application instance [14]. Configured through YAML-formatted Kubernetes manifest files, these pods are susceptible to security misconfigurations that could compromise the integrity and security of the entire system [13].

Building on the insights provided by Rahman et al. regarding the crucial role of static analysis tools in highlighting security misconfigurations within Kubernetes environments [13], this paper proposes an advancement through the implementation of a centralized logging system. This proposed system seeks to offer a comprehensive perspective on the security health of Kubernetes deployments, empowering system administrators with the capabilities to rapidly detect and address potential security vulnerabilities. By amalgamating the detailed detection and classification of security misconfigurations, as outlined by Rahman et al., with a novel logging mechanism, this paper aims to bolster the security robustness of Kubernetes infrastructures in the face of continually emerging cybersecurity challenges.

Following the comprehensive framework proposed by Rahman et al. [13], Mahajan's model introduces a proactive strategy by enforcing real-time misconfiguration checks prior to deployment, ensuring that any detected issues must be rectified before proceeding [15]. While this model emphasizes preventative measures, it does not offer a mechanism for the centralized collection, logging, or storage of the findings from static analysis tools, highlighting a critical area for enhancement. The landscape of existing open-source tools, including Checkov, KubeLinter, Datree, Trivy, and Snyk [16][17], presents a robust arsenal for the identification and notification of prevalent misconfigurations. Yet, the necessity to operate these tools in isolation introduces significant challenges in terms of resource consumption, operational complexity, and the risk of security oversights due to potential omissions. This scenario underscores the imperative for a unified solution that amalgamates the insights garnered from these disparate tools into a cohesive, centralized logging system. Such a system is envisaged to streamline the process of managing misconfiguration vulnerabilities by serving as a comprehensive repository and analysis point for security data across the Kubernetes deployment. By addressing the fragmentation inherent in the current use of multiple tools, which may lead to redundant or conflicting recommendations, the central logging solution proposed in this paper aims to simplify the resolution process by synthesizing outputs and guiding system administrators towards effective remediation strategies. This integration seeks to close the gap left by the piecemeal approach of existing tools, providing a singular, efficient platform for enhancing the security posture of Kubernetes environments [16] [18] [19] [20].

Beyond the initial identification of common security misconfigurations, further analysis into the types of vulnerabilities and the potential impacts on Kubernetes environments is essential. Misconfigurations such as the activation of hostIPC or inappropriate security contexts can lead to severe security implications, including cross-pod access and privilege escalation. These issues are further compounded when considering the dynamic and distributed nature of containerised environments, where the scale and speed of deployment can often outpace traditional security measures. Bose et al. emphasize that security-related defects in Kubernetes manifests are under-reported, suggesting a significant gap in the awareness and remediation of such vulnerabilities within the Kubernetes community [21]. Their research underscores the critical need for comprehensive security practices and the development of tools to systematically identify and mitigate security defects in Kubernetes configurations.

The risks associated with misconfigured Kubernetes manifests are not purely theoretical. There have been documented instances where such vulnerabilities have been exploited to conduct attacks ranging from data theft to the disruption of service. The work by Shamim et al. explores how malicious actors can leverage misconfigurations, such as those related to security best practices violations, to perform attacks like denial of service against Kubernetes clusters [18]. This research sheds light on the practical applications of theoretical vulnerabilities, reinforcing the importance of strict configuration management and the implementation of security best practices to safeguard against potential exploits.

The intricate nature of Kubernetes environments significantly amplifies the difficulties in ensuring secure configurations are maintained. Navigating through the myriad of options within Kubernetes manifest files demands a thorough comprehension of their potential security impacts. To uphold the security integrity of Kubernetes clusters, it's imperative to employ tools and methods that assist in identifying and correcting misconfigurations. A host of tools, including Kubesec, Kube-hunter, and Trivy, have been developed specifically to tackle cluster misconfigurations, playing a pivotal role in bolstering Kubernetes security [22].

However, the concept of integrating a unified logging framework, one that can consolidate outputs from a variety of such tools into a single, unified system, remains largely uncharted. This gap becomes even more pronounced when considering the challenges associated with presenting logged information in Kubernetes in a way that is both efficient and user-friendly for developers [23]. Although research efforts, like the one by Horalek et al. [24], have ventured into developing systems for aggregating and analyzing application logs within Kubernetes, these endeavors have mainly focused on operational aspects, overlooking the critical angle of security. Horalek et al.'s study, despite recognizing the complexity involved in delivering comprehensive logging solutions for Kubernetes applications, primarily highlighted operational benefits rather than delving into security ramifications. This oversight accentuates the urgent necessity for a centralized security logging system, aimed at merging and deciphering security-related data from various tools. Such a system would provide a comprehensive overview of the security posture within Kubernetes environments, addressing a crucial void in current research.

The open-source community has developed a plethora of tools specifically designed to identify misconfigurations in Kubernetes manifest files. These tools offer comprehensive scanning capabilities, enabling users to detect vulnerabilities, adherence to best practices, and potential security loopholes within their Kubernetes configurations.

Trivy is a comprehensive container and Kubernetes vulnerability scanner. It quickly scans container images and Kubernetes manifests to identify vulnerabilities, misconfigurations, and outdated dependencies using an extensive vulnerability database. Trivy integrates smoothly with Kubernetes, facilitating automated security checks across development and deployment workflows. It's a critical tool for maintaining the security of Kubernetes environments [25].

Kubesec evaluates Kubernetes manifests to ensure they meet security best practices. It checks configurations against a series of security benchmarks, focusing on pod security policies, the risk of privilege escalation, and network policies. Kubesec offers actionable insights and remediation advice, helping users secure their Kubernetes deployments more effectively [26].

Kube-score is a tool that analyzes Kubernetes manifests against best practices and standards. It provides a score and detailed feedback on configuration quality and reliability, identifying potential improvements and misconfigurations. kube-score guides developers and administrators in refining their Kubernetes configurations for better adherence to industry standards [27].

Kubeaudit is a Kubernetes-specific auditing tool that conducts thorough checks for security, reliability, and compliance. It uses a wide range of audit rules to examine Kubernetes manifests for misconfigurations and compliance issues, also supporting customizable rules for organization-specific audits. Kubeaudit is essential for ongoing auditing and compliance assurance in Kubernetes environments [28].

KubeLinter is a static analysis tool designed to identify misconfigurations in Kubernetes manifests and Helm charts. It checks for common configuration errors and flags the potential issues.

Each tool has their own interpretation of how severe a misconfiguration is to be treated, this is how these 4 tools treated severity levels in terms of misconfigurations:

- **Trivy:** Trivy classified its severity levels as Critical, High, Medium, Low.
- **Kube-score:** Kube-score evaluates configuration issues using a grading scale. These grades were converted into severity levels based off their grade where a grade of 10 is labeled as "CRITICAL," grades between 7 and 9.99 are considered "HIGH," grades between 5 and 6.99 are "MEDIUM," and grades below 5 are "LOW." Grades not fitting these categories are marked as "UNKNOWN."
- **Kube-linter:** Kube-linter does not assign explicit severity levels in its JSON output, focusing instead on the presence and specifics of misconfigurations.
- **Kubesec:** Kubesec assigns scores that can indirectly imply severity; higher scores indicate better security practices, whereas lower scores suggest critical vulnerabilities needing immediate attention.
- **Kubesec:** Kubesec assigns severity based on a point system, these points were converted into severity categories where a score of -30 is considered "CRITICAL," scores of -9 and -7 are "HIGH," scores of -3 and -1 are "MEDIUM," and scores of 1 and 3 are "LOW." Scores not explicitly listed are categorized as "UNKNOWN."

3. Architectural Design

This section explains the design of the proposed central logging solution and the proposed flow of the central logging solution.

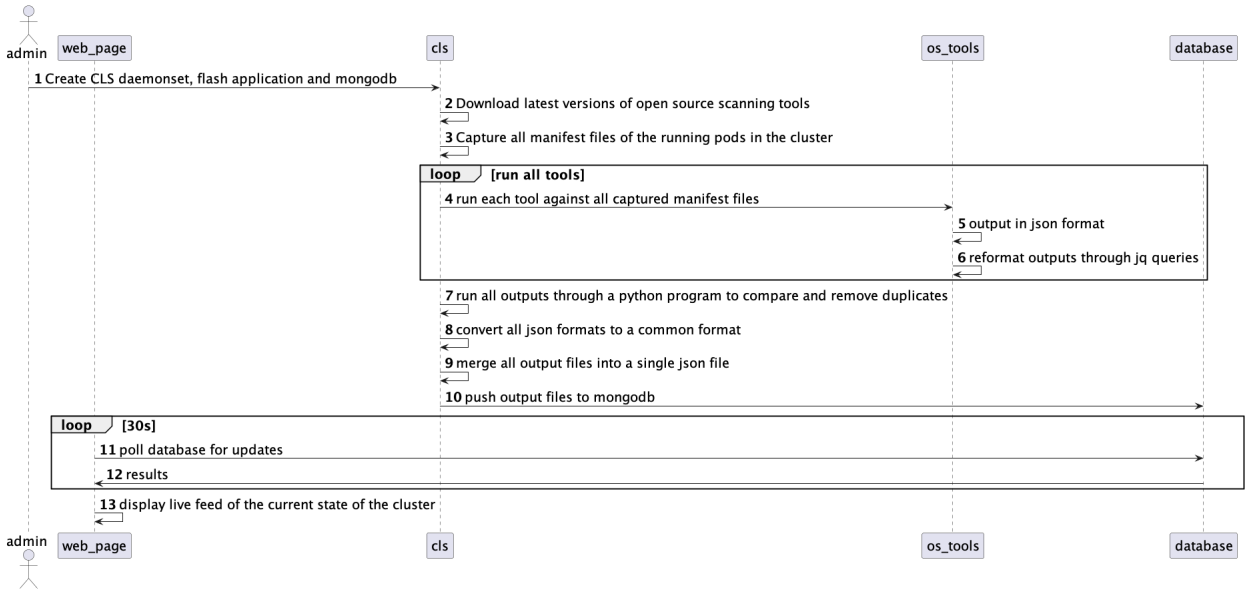


Figure 2: Central logging solution flow

Figure 2 depicts the proposed flow for the central logging solution. The flow is as follows:

1. An administrator will deploy the central logging solution, flask application and database in a Kubernetes environment.
2. The cls downloads the latest versions of the open source scanning tools.
3. The cls captures all manifest files of the running pods in the cluster and stores these in json files.
4. In a loop, each tool is ran against each manifest file.
5. The output is in json format.
6. The cls then reformats all json files through jq to only capture the required fields.
7. The cls passes all json output files to a comparative python program to try and remove duplicates across the tools.
8. All outputs are then converted into a common template.
9. All outputs are then combined into a single json file, representing 1 per live pod in the cluster.
10. The cls pushes all resulting files to the mongodb.
11. In a loop, the flash application is constantly polling the database for updates, refreshing every 30 seconds.
12. The web page displays a live feed of the current state of the cluster, and all misconfigurations and issues found across all the tools.

4. Implementation

The central logging solution is composed of several Docker containers: one for running the tools and another for the Flask application managing the web interface. The Central Logging Solution Dockerfile utilizes `python:3.12.2-alpine3.18` as a lightweight environment, setting `/app` as the workspace. It copies necessary bash scripts (`download_tools.sh`, `run_script.sh`, `entrypoint.sh`, `convert_jq.sh`, `merged.sh`) and Python files (`compare.py`, `sender.py`) to the container, grants execution permissions to the copied scripts, installs compilers (`g++`, `gcc`), libraries (`libffi-dev`, `musl-dev`, `openblas-dev`, `lapack-dev`), and tools (`bash`, `curl`, `jq`), installs `kubectl` for cluster communication, installs `pymongo` for database interaction, and sets `/app/entrypoint.sh` as the entry point script.

The Flask Application Dockerfile uses `python:3.9` as the base image, sets `/app` as the workspace, copies the Flask application code, installs Flask and `Flask-PyMongo`, exposes port 5002 for the Flask application, sets environment variables for Flask execution, and runs the Flask application using `flask run`.

The Kubernetes cluster was deployed using Docker Desktop, integrating Docker's containerization with Kubernetes' orchestration capabilities. Role-Based Access Control (RBAC) is essential for secure access to the cluster's resources. The setup includes Trivy Cluster Role Binding, which grants permissions to the default service account in the `security-namespace`, and Custom Cluster Role Binding, which links the view cluster role with a custom service account in the `security-namespace`.

The entrypoint script executes the following sequence: downloads and installs the latest versions of the tools, captures manifest files of all pods in the cluster, runs each tool against the manifests and formats the results, compares the results to remove duplicates, converts the outputs to a common template, merges all outputs into a single JSON file per pod, and uploads the JSON files to MongoDB.

The `run_script.sh` script captures manifest files and executes each tool. It initializes directories for outputs, retrieves namespaces and iterates through each to list and process all pods, runs tools (`Kubesec`, `Trivy`, `Kube-score`, `Kube-linter`) against each pod manifest and stores results in JSON format, and processes outputs using `jq` for formatting and storage.

The `compare.py` script eliminates duplicate results. It sets up directories and fetches file lists, loads common files and analyzes similarities using Python's `SequenceMatcher`, and stores unique issues in lists for further processing.

A Bash script uses `jq` to convert JSON files to a uniform structure and merges them into a single file per pod. A Python script automates importing JSON files into MongoDB, ensuring no duplicate data is uploaded.

The Flask application interfaces with MongoDB to present information about security vulnerabilities. The main index page displays an overview of vulnerabilities by severity, the severity level page lists vulnerabilities by specific severity levels, and the collection page provides detailed views of vulnerabilities in each collection. The application showcases various web pages: the CLS homepage displays the cluster's current status and vulnerability overview, the severity level page shows detailed information about high-severity vulnerabilities, and the collection page details security misconfigurations for individual collections.

4.1. Showcasing the running application

Current Vulnerabilities

Severity	Number of Vulnerabilities
HIGH	18
MEDIUM	71
LOW	246
CRITICAL	131
Unknown	88

Collection-wise Vulnerabilities

Collection Name	High	Medium	Low
kube-proxy-r5clt-result	3	7	21
testpod-result	1	6	20
vpnkit-controller-result	1	7	20
kube-controller-manager-docker-desktop-result	2	5	24
flask-app-result	1	5	19
mongo-result	1	6	20
kube-scheduler-docker-desktop-result	2	5	22
kube-apiserver-docker-desktop-result	2	6	30
coredns-5dd5756b68-cxwxm-result	1	6	15
storage-provisioner-result	1	7	20
etcd-docker-desktop-result	2	5	20
coredns-5dd5756b68-lfsjs-result	1	6	15

Figure 3: Homepage of CLS

Figure 3 illustrates the homepage of the central logging system, providing a comprehensive overview of the cluster's current status. It highlights the total number of vulnerabilities, categorized by severity levels. Additionally, it details the existing vulnerabilities within each collection, including a breakdown of their respective severity counts.

Back

HIGH Vulnerabilities

Collection Name	Severity	Title	AVDID	Description	Message	Resolution
kube-scheduler-docker-desktop-result	HIGH	Access to host network	AVD-KSV-0009	Sharing the host's network namespace permits processes in the pod to communicate with processes bound to the host's loopback adapter.	Pod 'kube-scheduler-docker-desktop' should not set 'spec.template.spec.hostNetwork' to true	Do not set 'spec.template.spec.hostNetwork' to true.
kube-scheduler-docker-desktop-result	HIGH	Root file system is not read-only	AVD-KSV-0014	An immutable root file system prevents applications from writing to their local disk. This can limit intrusions, as attackers will not be able to tamper with the file system or write foreign executables to disk.	Container 'kube-scheduler' of Pod 'kube-scheduler-docker-desktop' should set 'securityContext.readOnlyRootFilesystem' to true	Change 'containers[].securityContext.readOnlyRootFilesystem' to 'true'.
mongo-result	HIGH	Root file system is not read-only	AVD-KSV-0014	An immutable root file system prevents applications from writing to their local disk. This can limit intrusions, as attackers will not be able to tamper with the file system or write foreign executables to disk.	Container 'mongo' of Pod 'mongo' should set 'securityContext.readOnlyRootFilesystem' to true	Change 'containers[].securityContext.readOnlyRootFilesystem' to 'true'.
flask-app-result	HIGH	Root file system is not read-only	AVD-KSV-0014	An immutable root file system prevents applications from writing to their local disk. This can limit intrusions, as attackers will not be able to tamper with the file system or write foreign executables to disk.	Container 'flask-app' of Pod 'flask-app' should set 'securityContext.readOnlyRootFilesystem' to true	Change 'containers[].securityContext.readOnlyRootFilesystem' to 'true'.
kube-proxy-r5clt-result	HIGH	Access to host network	AVD-KSV-0009	Sharing the host's network namespace permits processes in the pod to communicate with processes bound to the host's loopback adapter.	Pod 'kube-proxy-r5clt' should not set 'spec.template.spec.hostNetwork' to true	Do not set 'spec.template.spec.hostNetwork' to true.
kube-proxy-r5clt-result	HIGH	Root file system is not read-only	AVD-KSV-0014	An immutable root file system prevents applications from writing to their local disk. This can limit intrusions, as attackers will not be able to tamper with the file system or write foreign executables to disk.	Container 'kube-proxy' of Pod 'kube-proxy-r5clt' should set 'securityContext.readOnlyRootFilesystem' to true	Change 'containers[].securityContext.readOnlyRootFilesystem' to 'true'.

Figure 4: Severity level page

Figure 4 presents the *HIGH* severity page, which can be accessed by selecting **HIGH** from the homepage. This page displays all *HIGH* misconfiguration vulnerabilities across the entire cluster. It provides detailed information for each vulnerability, including the name of the collection, its severity level, the title, the AVDID, a description, a message, and a recommended resolution that an administrator can implement to rectify the misconfiguration.

Back

testpod-result Vulnerabilities

Severity	Title	AVDID	Description	Message	Resolution
HIGH	Root file system is not read-only	AVD-KSV-0014	An immutable root file system prevents applications from writing to their local disk. This can limit intrusions, as attackers will not be able to tamper with the file system or write foreign executables to disk.	Container 'alpine-container' of Pod 'testpod' should set 'securityContext.readOnlyRootFilesystem' to true	Change 'containers[].securityContext.readOnlyRootFilesystem' to 'true'.
MEDIUM	Can elevate its own privileges	AVD-KSV-0001	A program inside the container can elevate its own privileges and run as root, which might give the program control over the container and node.	Container 'alpine-container' of Pod 'testpod' should set 'securityContext.allowPrivilegeEscalation' to false	Set 'set containers[].securityContext.allowPrivilegeEscalation' to 'false'.
MEDIUM	Runs as root user	AVD-KSV-0012	Force the running image to run as a non-root user to ensure least privileges.	Container 'alpine-container' of Pod 'testpod' should set 'securityContext.runAsNonRoot' to true	Set 'containers[].securityContext.runAsNonRoot' to true.
MEDIUM	hostPath volumes mounted	AVD-KSV-0023	According to pod security standard 'HostPath Volumes', HostPath volumes must be forbidden.	Pod 'testpod' should not set 'spec.template.volumes.hostPath'	Do not set 'spec.volumes[*].hostPath'.
MEDIUM	Seccomp policies disabled	AVD-KSV-0104	A program inside the container can bypass Seccomp protection policies.	container "alpine-container" of pod "testpod" in "security-namespace" namespace should specify a seccomp profile	Specify seccomp either by annotation or by seccomp profile type having allowed values as per pod security standards
LOW	Default capabilities: some containers do not drop all	AVD-KSV-0003	The container should drop all default capabilities and add only those that are needed for its execution.	Container 'alpine-container' of Pod 'testpod' should add 'ALL' to 'securityContext.capabilities.drop'	Add 'ALL' to containers[].securityContext.capabilities.drop.
LOW	CPU not limited	AVD-KSV-0011	Enforcing CPU limits prevents DoS via resource exhaustion.	Container 'alpine-container' of Pod 'testpod' should set 'resources.limits.cpu'	Set a limit value under 'containers[].resources.limits.cpu'.
LOW	CPU requests not specified	AVD-KSV-0015	When containers have resource requests specified, the scheduler can make better decisions about which nodes to place pods on,	Container 'alpine-container' of Pod 'testpod' should set 'resources.requests.cpu'	Set 'containers[].resources.requests.cpu'.

Figure 5: Collection page

Figure 5 presents the collection page for the TestPod, providing a detailed view of the identified security misconfigurations for the single collection. This page is organized to display issues from highest to lowest severity and includes several key elements: the title, AVDID, a description of the misconfiguration, a detailed message regarding the issue, and the recommended resolution steps.

This section assesses the suggested model by using both quantitative metrics and qualitative analysis, concentrating on its effectiveness and performance. Furthermore, this section discusses whether the initial goals of this research were met and highlights the strengths and weaknesses of the proposed solution. It also details the challenges encountered during the paper.

4.2. Examining the results

The previous section outlines how the central logging solution is implemented, the outputs and metrics are then obtained and analysed in order to evaluate the reliability of the solution.

Table 2 summarizes the execution times of all tools and scripts over 20 runs, providing data on the minimum, maximum, average, median, and mode execution times.

- `run_script.sh`: This represents the overall execution time for all tools combined, varying between 29 to 35 seconds. The longer duration is due to Trivy needing to download its database during the initial run. Subsequent runs, which do not require this download, typically see a reduction in execution time by 5-6 seconds.

- `compare.py`: This measures the time required to compare the outputs of the tools with each other, exhibiting an exceptionally low execution time of 0 to 1 seconds. It consistently demonstrates speed and efficiency.

Script	Min Time (s)	Max Time (s)	Average Time (s)	Median Time (s)	Mode Time (s)
run_script.sh	29	35	29.95	30	30
compare.py	0	1	0.20	0	0
convert_jq.sh	0	1	0.75	1	1
merged.sh	0	1	0.30	0	0
sender.py	0	1	0.60	1	1
Total cycle time	30	37	32	32	32

Table 2: Combined execution times detailing the minimum, maximum, average, median, and mode times for each script across 20 runs

- `convert_jq.sh`: This indicates the time required to convert all outputs into a predefined template, which is remarkably fast, with execution times ranging from 0 to 1 seconds.
- `merged.sh`: This represents the time needed to merge all the output files from each of the tools into a single, larger file, ready for uploading to MongoDB.
- `sender.py`: This describes the duration required to upload the results to the MongoDB, with the execution times varying from 0 to 1 seconds.

The total cycle time spans from 30 to 37 seconds, with an average and most frequent execution time of 32 seconds, this reflects the combined impact of all the scripts on their overall processing, execution and completion time.

4.3. Execution of each tool

Each tool varies in execution time, so I aimed to assess the performance of each to determine which one takes the longest and if this should be considered a significant factor. One thing to note is that when running each tool individually, the pod manifest files must be gathered per tool, which impacts the time of each tools execution time.

Tool	Min Time (s)	Max Time (s)	Average Time (s)	Median Time (s)	Mode Time (s)
Trivy	24	29	24.3	24	24
Kubescore	4	5	4.8	5	5
Kubesec	6	8	6.9	7	7
Kubelint	5	6	5.8	6	6

Table 3: Tool execution times detailing the minimum, maximum, average, median, and mode times across 20 runs

Table 3 provides the execution times for the open source tools over 20 runs, illustrating their performance characteristics:

Trivy

Trivy consistently executed around 24 seconds, with its times ranging from a minimum of 24 to a maximum of 29 seconds, and an average of 24.3 seconds. The peak time of 29 seconds typically occurred during the initial run, as Trivy needed to download its database upon first execution, which prolonged the process. However, subsequent runs did not require a database download, resulting in quicker execution times.

Kubescore

Kubescore had an execution time ranging from 4 to 5 seconds, with an average of 4.8 seconds. This makes it the lowest execution time of the open source tools tested in the central logging solution.

Kubesec

Kubesec's execution times varied between 6 and 8 seconds, averaging 6.9 seconds. This made it quick and efficient at scanning manifest files.

Kubelint

Kubelint's execution times spanned from 5 to 6 seconds, with an average of 5.8 seconds. This resulted in a rapid and efficient scan of manifest files.

4.4. Security Tool Comparison

This section provides a comparative analysis of the open-source security tools, focusing on their effectiveness in detecting misconfigurations and vulnerabilities. We will examine and contrast the outputs from each tool to understand their strengths and weaknesses in identifying potential security risks.

To ensure a balanced evaluation of the tools, I will analyze and review their outputs based on the "testpod" manifest file. This will involve presenting a snippet from each tool based on finding the same misconfiguration and then discussing their results.

Trivy

Trivy Sample Output

```
{
  "Type": "Kubernetes Security Check",
  "ID": "KSV014",
  "AVDID": "AVD-KSV-0014",
  "Title": "Root file system is not read-only",
  "Description": "An immutable root file system prevents applications
                  from writing to their local disk. This can
                  limit intrusions, as attackers will not be
                  able to tamper with the file system or write
                  foreign executables to disk.",
  "Message": "Container 'alpine-container' of Pod 'testpod' should set
              'securityContext.readOnlyRootFilesystem' to true",
  "Resolution": "Change
                'containers[].securityContext.readOnlyRootFilesystem'
                to 'true'.",
  "Severity": "HIGH"
}
```

The Trivy scan of the "testpod" manifest file has uncovered a high-severity vulnerability (ID: KSV014, AVDID: AVD-KSV-0014) in the 'alpine-container' configuration within the pod. The container's root file system is writable, presenting a major security risk as it could allow unauthorized modifications and the installation of malicious executables. To address this issue, it is advised to set the securityContext.readOnlyRootFilesystem attribute of the container to true. This adjustment will enforce a read-only file system, significantly boosting the pod's security integrity. Trivy provides a clear and concise output, the severity rating helps administrators quickly identify and prioritise what is critical to securing the system.

Kubescore

Kubescore Sample Output

```
{
  "name": "Container Security Context ReadOnlyRootFilesystem",
  "id": "container-security-context-readonlyrootfilesystem",
  "comment": "Makes sure that all pods have a security
             context with read only filesystem set",
  "grade": 1,
  "severity": "LOW"
}
```

The KubeScore output checks if all pods have a read-only root file system set. It's assigned a severity of LOW and a grade of 1. The output is clear but could benefit from more detailed explanations for a better understanding. Notably, this LOW severity rating contrasts with the HIGH severity rating provided by Trivy for a similar security concern in the "testpod" manifest file. This difference highlights the need for administrators to critically assess and perhaps even seek multiple sources or tools when evaluating security issues, as each tool may have its own perspective on risk assessment.

Kubesec

Kubesec Sample Output

```
{
  "id": "ReadOnlyRootFilesystem",
  "selector":
  "containers[] .securityContext .readOnlyRootFilesystem == true",
  "reason": "An immutable root filesystem can prevent
          malicious binaries being added to PATH and increase
          attack cost",
  "points": 1,
  "severity": "LOW"
}
```

The KubeSec output for "ReadOnlyRootFilesystem" checks if containers have the securityContext.readOnlyRootFilesyst set to true, aiming to secure containers by making the root file system immutable, rating it as LOW severity. In contrast, Trivy's more detailed output assigns a HIGH severity to a similar setting, providing broader context and implications. KubeScore, like KubeSec, also rates it LOW but without extensive details. KubeSec's concise style, similar to KubeScore's, focuses on direct security benefits but lacks the depth found in Trivy, which could be more useful for administrators making informed security decisions. This again highlights how one tool could influence how security measures are prioritised and implemented.

Kubelint

Kubelint Sample Output

```
{
  "Check": "no-read-only-root-fs",
  "Message": "container \"alpine-container\" does not
             have a read-only root file system",
  "Remediation": "Set readOnlyRootFilesystem to true
                 in the container securityContext.",
  "Severity": "Unknown"
}
```

The Kube-Linter output identifies that the "alpine-container" lacks a read-only root file system and

recommends setting readOnlyRootFilesystem to true in the container's securityContext. Unlike Trivy, which assigns a HIGH severity to the same issue and provides extensive context, both KubeScore and KubeSec label this issue as LOW severity, indicating less urgency. However, Kube-Linter's "Unknown" severity could leave administrators unsure about how critically they need to address the misconfiguration identified. This highlights the discrepancies across different security tools, outlining the need for either a consistent security evaluation or else the use of multiple tools for a thorough security analysis.

Reviewing the outputs from Trivy, KubeScore, KubeSec, and Kube-Linter reveals significant differences in how each tool interprets and reports Kubernetes security vulnerabilities. While Trivy emphasizes the severity of issues by providing detailed context and a high urgency rating, KubeScore and KubeSec tend to offer more straightforward, less urgent assessments with a focus on practical remediation steps. Kube-Linter distinguishes itself by offering direct remediation advice but often leaves the severity as "Unknown," potentially complicating priority assessments for administrators. Each tool not only varies in its interpretation of security risks but also uses different keys in their JSON outputs, reflecting different approaches to structuring security data.

4.5. Challenges

Comparing the outputs from the security tools: Trivy, KubeScore, KubeSec, and Kube-Linter had its challenges due to the varying keys used in their JSON outputs. Each tool organizes its data differently, using unique keys to denote aspects like severity, recommendations, and identification tags. This inconsistency in data structure complicated the central logging solution, as the data could not be filtered quickly.

Additionally, each tool interprets and assigns different severity levels. Trivy assigned a HIGH severity to a vulnerability that KubeScore or KubeSec considered a LOW severity. Whereas Kube-Linter categorised it as an "Unknown" severity level, which made converting all the results to a predefined template more difficult. Also, if an administrator only used 1 security tool and it denoted a misconfiguration as LOW, then it could be ignored or skipped.

5. Conclusions and Future Work

This paper presented a centralized logging solution aimed at mitigating Kubernetes misconfigurations using open-source tools. The proposed solution integrates multiple security tools to identify and log misconfigurations in Kubernetes manifests, providing a consolidated view for administrators. By automating the detection and aggregation of misconfiguration data, the solution significantly enhances the visibility of potential security issues within a Kubernetes cluster.

The implemented solution successfully achieved its primary goals: it demonstrated the feasibility of integrating various open-source tools into a unified logging system, evaluated the effectiveness of these tools in a real-world Kubernetes environment, and provided a detailed comparative analysis of their outputs. The central logging solution was able to identify critical security vulnerabilities, streamline the diagnostic process, and present the results in an accessible format via a web interface. This work contributes to the field of Kubernetes security by offering a practical tool for improving the security posture of Kubernetes deployments.

For an ideal central logging solution, the best scenario would involve a drag-and-drop feature that allows for easy integration of new tools. However, the current system struggles with this because of the variability in the key/value pairs used in their JSON outputs. A potential future approach could involve using machine learning to categorise and reconcile these differing key/value pairs.

Another area for future development could concentrate on refining how messages and descriptions from each tool are compared to eliminate redundant results. Currently, this is handled using Python's "SequenceMatcher" library, which identifies similar string sequences without considering the context of the messages. Implementing a machine learning strategy could improve the effectiveness of these comparisons by analyzing the context within the messages.

In today's technological landscape, containerised applications are increasingly popular, with Kubernetes being adopted by major organizations like Google, Adidas, Spotify, Bose, and the United States Department

of Defense. This widespread adoption has led to the emergence of security vulnerabilities, particularly those stemming from cluster misconfigurations. This dissertation has concentrated on developing a centralized logging solution that leverages popular open-source tools to identify misconfigurations in Kubernetes manifest files. Prior to this paper, there was no unified solution that integrated multiple open-source tools.

This solution serves as a proof of concept for those interested in delving deeper into misconfigurations detected by open-source tools in Kubernetes environments. There is potential for further enhancements to elevate this solution to a production-grade level, making it an invaluable resource for administrators to identify and address security vulnerabilities effectively.

References

[1] S. K. Mondal, R. Pan, H. D. Kabir, T. Tian, H.-N. Dai, Kubernetes in it administration and serverless computing: An empirical study and research challenges, The Journal of Supercomputing (2022) 1–51.

[2] CNCF Annual Survey 2022 (Jan. 2023).
URL https://www.cncf.io/reports/cncf-annual-survey-2022/

[3] Stackrox annual survey 2020 (2020).
URL https://cyso.com/wp-content/uploads/2020/04/Stackrox-State_of_Container_and_Kubernetes_Report-winter-2020.pdf

[4] D. Goodin, Tesla cloud resources are hacked to run cryptocurrency-mining malware (Feb. 2018).
URL https://arstechnica.com/information-technology/2018/02/tesla-cloud-resources-are-hacked-to-run-cryptocurrency-mining

[5] T. Taylor, 5 kubernetes security incidents and what we can learn from them (2019).
URL https://techgenix.com/5-kubernetes-security-incidents/

[6] M. S. I. Shamim, F. A. Bhuiyan, A. Rahman, Xi commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices, 2020 IEEE Secure Development (SecDev) (2020) 58–64.

[7] N. Yang, W. Shen, J. Li, X. Liu, X. Guo, J. Ma, Take over the whole cluster: Attacking kubernetes via excessive permissions of third-party applications, in: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, 2023, pp. 3048–3062.

[8] Mitre att&ckÂ® (Mar. 2024).
URL https://attack.mitre.org/

[9] Y. Weizman, Threat matrix for Kubernetes (Apr. 2020).
URL https://www.microsoft.com/en-us/security/blog/2020/04/02/attack-matrix-kubernetes/

[10] V. B. Mahajan, S. B. Mane, Detection, analysis and countermeasures for container based misconfiguration using docker and kubernetes, in: 2022 International Conference on Computing, Communication, Security and Intelligent Systems (IC3SIS), IEEE, pp. 1–6.

[11] State of Kubernetes security report 2023 (Apr. 2023).
URL https://www.redhat.com/en/resources/state-kubernetes-security-report-2023

[12] Dbir report 2023 - introduction.
URL https://www.verizon.com/business/resources/reports/dbir/2023/introduction/

[13] A. Rahman, S. I. Shamim, D. B. Bose, R. Pandita, Security misconfigurations in open source kubernetes manifests: An empirical study, ACM Transactions on Software Engineering and Methodology 32 (4) (2023) 1–36.

[14] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, S. Kim, Horizontal pod autoscaling in kubernetes for elastic container orchestration, Sensors 20 (16) (2020) 4621.

[15] V. B. Mahajan, S. B. Mane, Detection, analysis and countermeasures for container based misconfiguration using docker and kubernetes, in: 2022 International Conference on Computing, Communication, Security and Intelligent Systems (IC3SIS), IEEE, 2022, pp. 1–6.

[16] A. Rahman, S. I. Shamim, D. B. Bose, R. Pandita, Security misconfigurations in open source kubernetes manifests: An empirical study, ACM Transactions on Software Engineering and Methodology 32 (4) (2023) 1–36.

[17] P. Saxena, Container image security with trivy and istio inter-service secure communication in kubernetes, Thesis (2023).

[18] S. I. Shamim, Mitigating security attacks in kubernetes manifests for security best practices violation, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, pp. 1689–1690.

[19] S. Bagheri, H. Kermabon-Bobinnec, S. Majumdar, Y. Jarraya, L. Wang, M. Pourzandi, Warping the defence timeline: Non-disruptive proactive attack mitigation for kubernetes clusters, in: ICC 2023-IEEE International Conference on Communications, IEEE, 2023, pp. 777–782.

[20] H. Kermabon-Bobinnec, M. Gholipourchoubeh, S. Bagheri, S. Majumdar, Y. Jarraya, M. Pourzandi, L. Wang, Prospec: Proactive security policy enforcement for containers, in: Proceedings of the Twelfth ACM Conference on Data and Application Security and Privacy, 2022, pp. 155–166.

[21] D. B. Bose, A. Rahman, S. I. Shamim, 'under-reported' security defects in kubernetes manifests, in: 2021 IEEE/ACM 2nd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS), IEEE, 2021, pp. 9–12.

[22] B. Karakaş, et al., Enhancing security in communication applications deployed on kubernetes: Best practices and service mesh analysis (2023).

[23] D. Koryugin, Analysing and alerting on application logs within kubernetes infrastructure, B.S. thesis, University of Twente (2023).

[24] J. Horalek, P. Urbanik, V. Sobeslav, T. Svoboda, Proposed solution for log collection and analysis in kubernetes environment, in: International Conference on Nature of Computation and Communication, Springer, 2022, pp. 9–22.

[25] A. Security, Trivy - a simple and comprehensive vulnerability scanner for containers and other artifacts, https://github.com/aquasecurity/trivy (2023).

[26] ControlPlane, Kubesec - security risk analysis for kubernetes resources, https://github.com/controlplaneio/kubesec (2023).

[27] Zegl, kube-score - kubernetes object analysis with recommendations for improved reliability and security, https://github.com/zegl/kube-score (2023).

[28] Shopify, Kubeaudit - audit your kubernetes clusters against common security controls, https://github.com/Shopify/kubeaudit (2023).

CHAOSEATER: Fully Automating Chaos Engineering with Large Language Models

Daisuke Kikuta, Hiroki Ikeuchi, Kengo Tajiri
NTT Corporation
daisuke.kikuta@ntt.com

Project page : https://ntt-dkiku.github.io/chaos-eater
Code : https://github.com/ntt-dkiku/chaos-eater
Demo : https://huggingface.co/spaces/oookiku/chaos-eater

arXiv:2501.11107v2 [cs.SE] 16 Apr 2025

Abstract

Chaos Engineering (CE) is an engineering technique aimed at improving the resiliency of distributed systems. It involves artificially injecting specific failures into a distributed system and observing its behavior in response. Based on the observation, the system can be proactively improved to handle those failures. Recent CE tools implement the automated execution of predefined CE experiments. However, defining these experiments and improving the system based on the experimental results still remain manual. To reduce the costs of the manual operations, we propose CHAOSEATER, a system for automating the entire CE operations with Large Language Models (LLMs). It predefines the agentic workflow according to a systematic CE cycle and assigns subdivided operations within the workflow to LLMs. CHAOSEATER targets CE for Kubernetes systems, which are managed through code (i.e., Infrastructure as Code). Therefore, the LLMs in CHAOSEATER perform software engineering tasks to complete CE cycles, including requirement definition, code generation, debugging, and testing. We evaluate CHAOSEATER through case studies on both small and large Kubernetes systems. The results demonstrate that it stably completes reasonable single CE cycles with significantly low time and monetary costs. The CE cycles are also qualitatively validated by human engineers and LLMs.

1 Introduction

Modern software applications are built on distributed systems, where the entire systems are composed of networks of subdivided component services. This design, known as microservice architecture (Bucchiarone et al., 2020), enables scalable and continuous deployment while supporting the integration of heterogeneous technologies. On the other hand, the complex dependencies among small services can lead to unexpected and chaotic behavior in the entire system, even from minor failures. However, proactively predicting and addressing such complex behavior is challenging.

To address this and improve the resiliency of distributed systems, numerous organizations, including Netflix, Amazon, and Microsoft, have recently adopted Chaos Engineering (CE) (Basiri et al., 2016, 2019). Its concept is that *rather than predicting the chaotic behavior, let's observe it directly by artificially injecting the failures into the system*. Based on the actual observation, we can proactively rebuild a new system that is resilient to the assumed failures. Systematically, CE cycles through four phases for a target system:

1. ***Hypothesis***: Define steady states (i.e., normal behavior) of the system and a failure scenario. Then, make a hypothesis that *the steady states of the system are maintained even when the failures occur in the scenario*.
2. ***(Chaos) Experiment***: Inject the failures into the system while logging the system's response behavior.
3. ***Analysis***: Analyze the logged data and check if the hypothesis is satisfied. If so, this CE cycle is finished here. If not, move to (4).
4. ***Improvement***: Reconfigure the system to satisfy the hypothesis. The reconfigured system is tested again in (2) and (3), i.e., repeat (2) to (4) until the hypothesis is satisfied.

In recent years, several CE tools (Netflix, 2012; Amazon Web Services, 2021; Chaos Mesh, 2021; Microsoft, 2023) have advanced the automation of chaos-experiment execution. Moreover, monitoring tools (Prometheus, 2012; Grafana Labs, 2021) enable automating metric collection, aggregation,

and threshold-based testing during chaos experiments. Hence, the *experiment* and *analysis* phases have been mostly automated. However, defining a hypothesis in the *hypothesis* phase, planning a chaos experiment to test the hypothesis in the *experiment* phase, and reconfiguring the system in the *improvement* phase still remain manual. These manual operations require a complex set of skills, including domain knowledge in networking and CE, the ability to interpret system configurations, logs, and error messages, as well as generative problem-solving for requirement definition, experiment planning, and system reconfiguration. Consequently, while the costs of these operations remain high, their automation has not been achieved yet with existing algorithmic approaches.

We believe that Large Language Models (LLMs) are the key to overcoming this challenge. LLMs have recently shown promising capabilities across a wide range of general tasks required for CE, including natural language processing, coding, and network operation (Zhao et al., 2023; Jiang et al., 2024; Ahmed et al., 2024; Piovesan et al., 2024). Moreover, LLMs have demonstrated promising performance in software engineering (SE) tasks (Yang et al., 2024b; Cognition Labs, 2024), which is crucial for automating CE. Infrastructure as Code (IaC) enables software systems to be managed through code, and recent CE tools can manage chaos experiments in the same way. Therefore, CE operations on software systems can be regarded as SE tasks, and the promising results in automating SE tasks suggest the feasibility of automating CE. Given those general capabilities, domain knowledge in networking, and the compatibility between IaC and LLMs for SE, LLMs are a strong candidate for fully automating the CE cycle.

Here, we propose CHAOSEATER, a system for automating the entire CE cycle with LLMs. It predefines the agentic workflow according to the systematic CE cycle and assigns subdivided CE operations within the workflow to each LLM with a specific role. The predefined workflow ensures that the multiple LLMs collaboratively perform CE operations as intended. Considering the compatibility between IaC and LLMs for SE, CHAOSEATER especially targets CE on Kubernetes (K8s) (Kubernetes, 2014) systems, which are software systems managed through code. Therefore, the LLMs perform SE tasks to complete CE cycles, including requirement definition, code generation, debugging, and testing. In this paper, we present the workflow design, a set of system prompts for creating LLM agents for each CE operation, the interface between LLMs and existing CE tools, and a technique for conducting consistent and transparent testing.

We evaluate CHAOSEATER through case studies on both small and large K8s systems. The results demonstrate that it stably completes reasonable single CE cycles with significantly low time and monetary costs ($0.2–0.8 and 11–25m). The validity of these CE cycles is also confirmed by two human engineers and three different LLMs.

The main contributions of this paper are organized as follows:

- We are the first to propose a system for fully automating the systematic CE cycle with LLMs, which significantly reduces time and monetary costs in CE cycles. This proposal would be a starting point towards the full automation of system resiliency improvement.
- We make all resources of CHAOSEATER public. This release provides a general development practice for constructing complex systems that combine LLMs and existing tools.
- We evaluate CHAOSEATER quantitatively in terms of cost and stability, and qualitatively in detail for each phase by both human engineers and LLMs. The results provide its fine-grained potential, limitations, and future directions (see Appendix B).

2 Proposed System: CHAOSEATER

In this section, we describe a technical overview of CHAOSEATER. Figure 1 shows its simplified agentic workflow. It takes as input instructions for the CE cycle (optional) and a folder containing K8s manifests (Kubernetes, 2014) and a Skaffold configuration file (Google, 2019). In short, K8s manifests are system configuration files that define the resources (i.e., small services) that constitute a system, while a Skaffold configuration file defines the process to automatically deploy those resources in a K8s cluster. It then conducts a CE cycle for those inputs through five divided phases: *pre-processing*, *hypothesis*, *experiment*, *analysis*, *improvement*, and *post-processing* phases. Finally, it outputs a summary of the completed CE cycle and a modified folder containing K8s manifests that have been reconfigured to satisfy the hypothesis defined in the *hypothesis* phase, along with their corresponding Skaffold configuration file.

In the following, we describe CHAOSEATER's

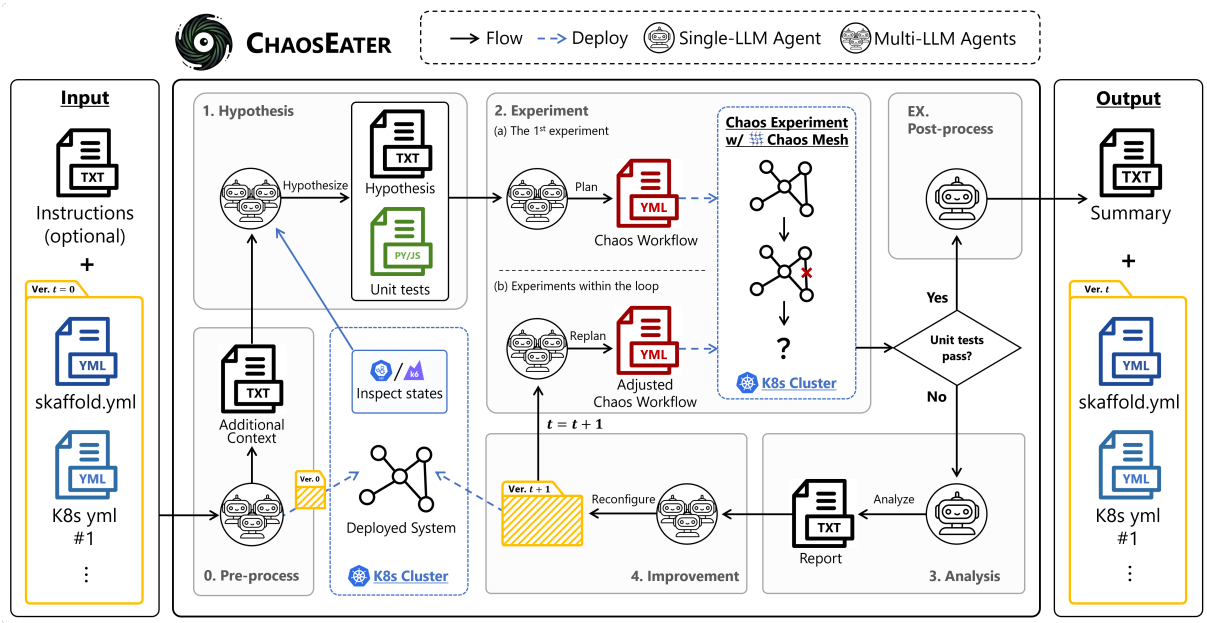


Figure 1: A simplified agentic workflow of CHAOSEATER. CHAOSEATER follows the workflow to autonomously complete the systematic CE cycle using LLM agents and existing tools. Note that only the representative inputs and outputs of agents are illustrated here. The two K8s clusters within the workflow refer to the same one.

workflow design from input to output, breaking it down into the five phases. Note that, in this paper, we refer to each LLM assigned a specific role (i.e., a subdivided CE operation) as an LLM agent, and that the underlying LLMs do not require additional fine-tuning. See Appendix D for implementation details, including the detailed agentic workflow, system prompt templates, graphical user interface, and system deployment.

2.1 Phase 0: Pre-processing

Given the user input, CHAOSEATER first deploys the user's system to the K8s cluster by running the Skaffold configuration file. Then, each LLM agent sequentially fills in the implicit context of the user's input. The filled context includes summary of the K8s manifests, their potential issues for resiliency and redundancy, and a possible application, which will be provided as auxiliary information in the subsequent phases. The filtering of harmful prompts in user instructions is also performed here.

2.2 Phase 1: Hypothesis

The *hypothesis* phase defines the system's resiliency for an assumed failure scenario, which corresponds to the requirements definition from a fault tolerance perspective. Following the principles of CE (Basiri et al., 2016), CHAOSEATER first defines steady states and defines a failure scenario.

Steady-state definition Steady states are expected, normal behaviors of a system. Each steady state is defined by a pair of a state value and a threshold, and a steady state is considered satisfied when the state value meets the threshold. Therefore, the state values must be measurable outputs of the system, such as the number of active resources, error rates, and response time.

Given the pre-processed user input, an LLM agent first defines a measurable state critical to maintaining the system's application. Another agent then inspects the current value of the state using either K8s API or k6 (Grafana Labs, 2021). The inspection is conducted by a Python or JavaScript script written by the agent. Based on the inspected value, an agent defines the threshold for the state, which, according to the definition of a steady state, must be satisfied under the current (normal) state. Finally, an agent adds threshold-based assertions to the inspection script to generate a unit test script that validates whether the steady state is satisfied. These processes are repeated to list multiple steady states without duplication until an agent determines that the number of steady states is sufficient. The unit test script is used for mechanically validating the steady state during the *experiment* phase; we call this approach of having LLMs judge validity through unit test code *Validation as Code* (VaC), which ensures consistency and

transparency in the validation process.

Failure definition Given the pre-processed user inputs and the steady states, an LLM agent proposes a failure scenario that may occur in the system (e.g., a surge in access due to a promotional campaign, cyber attack, etc.), and defines a sequence of failures that simulate the scenario. CHAOSEATER employs Chaos Mesh (Chaos Mesh, 2021), a CE tool that can manage chaos experiments on K8s systems through code; therefore, the failures are selected from ones supported by Chaos Mesh. After drafting failures, an another agent refines the detailed parameters for each Chaos Mesh failure, such as the scope of the failure injection, the failure sub-type, the failure strength. See Appendix D.3 and D.4.3 for supported failures and mechanism of defining failures, respectively.

At this point, the hypothesis can be reframed as *all VaC scripts pass, even when the defined failure injections are performed*.

2.3 Phase 2: (Chaos) Experiment

The *experiment* phase plans a chaos experiment to validate the hypothesis and executes it.

Experiment planning To enable systematic planning, we propose dividing a chaos experiment into three stages: pre-validation, failure-injection, and post-validation. In the pre-validation stage, VaC scripts are run to ensure that the steady states are satisfied under normal conditions. In the failure-injection stage, failure injections are performed. If some steady states needs to be validated concurrently, the corresponding VaC scripts are also run here. In the post-validation stage, VaC scripts are run to ensure that steady states have been properly recovered after the failure injections.

Given the pre-processed user inputs and the hypothesis, an LLM agent first determines the duration of each stage. Then, other agents determine the VaC scripts and failure injections to be executed in each stage, along with their execution timing and durations. Finally, an agent writes a summary of the chaos experiment timeline, which is referenced during the *analysis* phase to identify the causes of the hypothesis not being satisfied in the chaos experiment. The three-stage chaos-experiment plan is then converted to a Chaos Mesh workflow manifest, which enables automated failure injection and hypothesis validation via VaC scripts according to the schedule defined in the manifest. See Appendix D.4.4 for our proposed algorithm and LLMs' output format for this conversion.

Experiment replanning Resource types and metadata of K8s manifests may be reconfigured during the *improvement* phase. Therefore, inspection targets in VaC scripts and scopes of failure injections must be updated accordingly between the *improvement* phase and the next experiment execution. Given the original and reconfigured K8s manifests, as well as their previous configurations, each LLM agent proposes new inspection targets and new scopes of failure injections. Then, a new ChaosMesh workflow manifest is generated by updating the corresponding parts in the previous one. Note that this update only makes minor adjustments to reflect the changes in K8s manifests, without altering the original intent of the chaos experiment.

Experiment execution After the Chaos Mesh workflow manifest is generated, CHAOSEATER applies it to the K8s cluster. Then, the scheduled failure injections and hypothesis validation are automatically executed by Chaos Mesh. In the meantime, CHAOSEATER simply waits for the experiment to complete.

2.4 Phase 3: Analysis

After the chaos experiment is finished, CHAOSEATER mechanically checks whether the VaC scripts have passed. If all of them have passed, that means the current system configurations (i.e., K8s manifests) already satisfy the hypothesis. Therefore, CHAOSEATER finishes the current CE cycle at this point and moves to the *post-processing* phase. If at least one has failed, CHAOSEATER moves to the next *improvement* phase after analyzing the experimental results. In this analysis, given the K8s manifests, the timeline of the chaos experiments, and the list of failed VaC scripts with their logs, an LLM agent identifies the cause of the fails and then generates a report containing the causes and recommended countermeasures.

2.5 Phase 4: Improvement

The *improvement* phase reconfigures the K8s manifests to satisfy the hypothesis. Given the K8s manifests, the hypothesis, the experiment plan, and the improvement loop history, an LLM agent reconfigures the K8s manifests so that all the VaC scripts pass in the chaos experiment. There are three reconfiguration modes: `create`, `delete`, and `replace`. The agent first selects modes

while specifying file names, and then writes the reconfigured K8s manifests only when `create` or `replace` is specified. The file management algorithm of CHAOSEATER then edits the folder from the previous improvement loop (in the first improvement, it corresponds to the user's input folder) according to the agent's output.

Improvement loop After the reconfiguration, CHAOSEATER applies the reconfigured K8s manifests to the K8s cluster. Then, they will be validated again through the *experiment* and *analysis* phases. That is, as in the systematic CE cycle, CHAOS-EATER also repeats the *experiment*, *analysis*, *improvement* phases until the hypothesis is satisfied. We define this loop as the improvement loop. The improvement loop history refers to the history of the experimental results, their analysis reports, and their reconfigurations within this improvement loop, which suppresses the repetition of the same reconfiguration.

2.6 Extra Phase: Post-processing

After the CE cycle is completed, CHAOSEATER finalizes its entire process by summarizing the completed CE cycle. An LLM agent summarizes the user's input and the four completed phases. Finally, CHAOSEATER provides the user with the summary of the completed CE cycle and the folder containing K8s manifests that have been reconfigured to satisfy the hypothesis defined in the *hypothesis* phase, along with their Skaffold configuration file.

3 Case Study

CE cycles should not be evaluated solely based on whether appropriate reconfigurations are performed; it is equally important to evaluate whether each phase leading up to it is meaningful. Therefore, rather than creating a benchmark that quantifies evaluation in the binary manner, we here evaluate CHAOSEATER through in-depth case studies focused on two critical cases.

The first case, NGINX, is a small-scale system consisting of two K8s manifests (i.e., two resources): `pod.yaml` and `service.yaml`. The former defines a `Pod` resource including a Nginx server, and the latter defines `Service` resource routing TCP traffic to the `Pod`. To verify whether CHAOSEATER can improve the system when there are resiliency issues, we intentionally configure the resource with a non-resilient setting; we set `restartPolicy` to `Never` in `Pod.yaml`. With this configuration, once the `Pod` goes down, it will never restart, resulting in extended service outages. The second case, SOCKSHOP (Weaveworks, 2023), is a practical and large-scale e-commerce system that consists of 29 manifests, which define the resources and databases for front-end pages, user information, order, payment, shipping, and so on. The number of replicas of all the `Deployment` resources is originally set to one. However, this setting could lead to downtime of the single replica when it goes down. To narrow down this original resiliency issue to a single point, we increase the replicas for `Deployment` resources other than `front-end-dep.yaml` to two, while keeping a single replica for `front-end-dep.yaml`. This relatively reduces the redundancy/resiliency of the front-end resource. In this case study, we validate whether CHAOSEATER correctly identifies and addresses these resiliency issues through a reasonable CE cycle.

To maintain the autonomy of CHAOSEATER, we input only the instruction to keep each chaos experiment within one minute (access methods are also input for SOCKSHOP). We use gpt-4o-2024-08-06 (OpenAI, 2024) as the underlying LLMs. To improve the reproducibility of this case study, its temperature is set to 0 with the random seed fixed at 42. We run a single CE cycle for each system five times under the same settings. In the following, we first discuss quantitative metrics and then qualitatively validate the operations within single CE cycles by LLM-as-a-judge and human engineers. See Appendix C and E for more details on the evaluation and results.

Costs and stability Table 1 shows the time and monetary costs of single CE cycles for each target system. Although we do not have statistical data on the actual working time and labor costs for the same CE cycles performed by human engineers, these total operational time and monetary costs ($0.21 and 11m) are obviously lower than that. For SOCKSHOP, the monetary cost increases by approximately four times ($0.84), and the time doubled (25m). However, these values are still intuitively lower than those of human engineers. Even with the number of resources increasing by more than ten times compared to NGINX, the cost increase remains minimal, demonstrating that CHAOSEATER maintains low costs even for large-scale systems.

In terms of stability, CHAOSEATER successfully

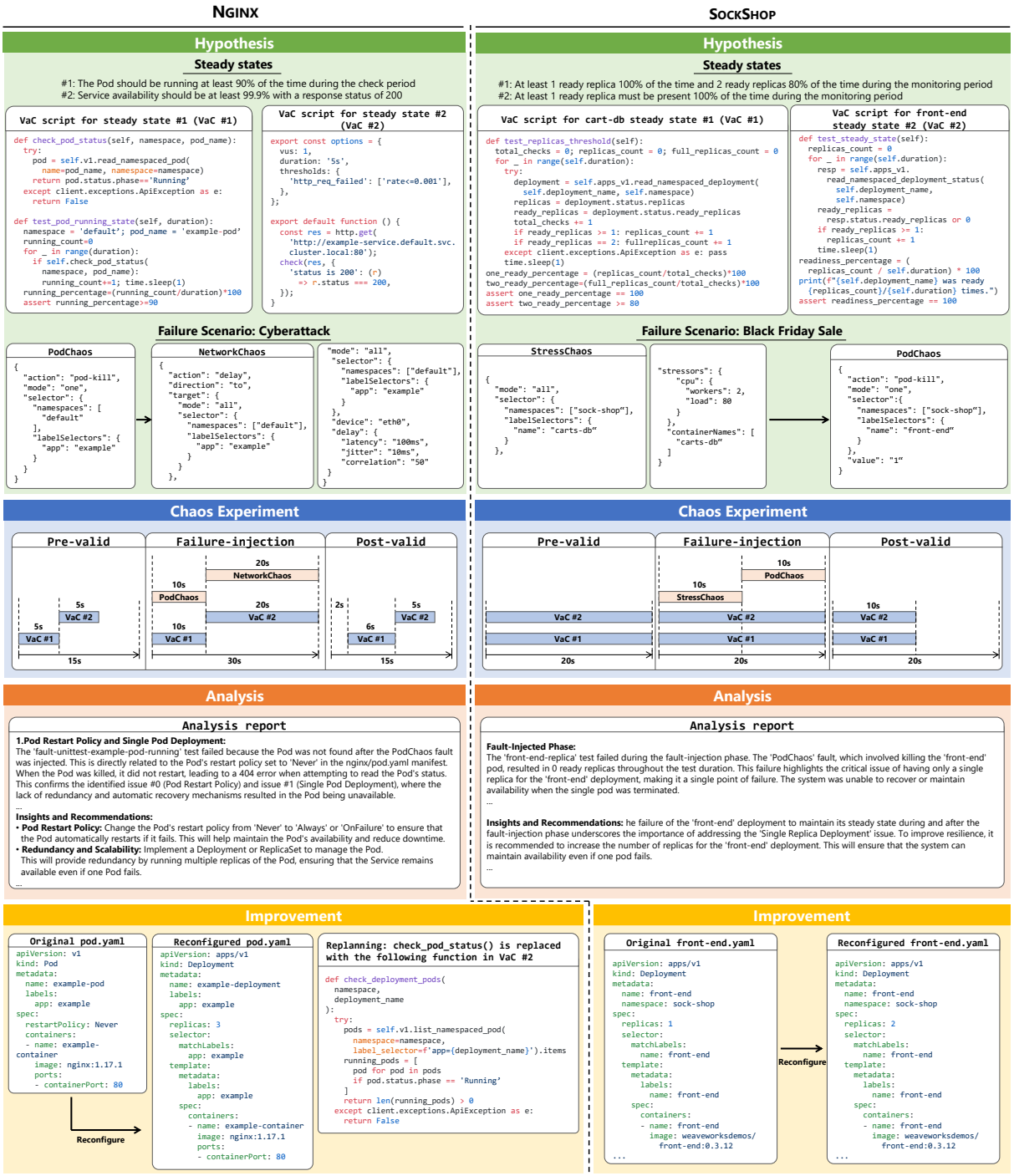


Figure 2: The highlighted outputs for NGINX and SOCKSHOP. See Appendix E.1 and E.3 for their full versions.

completes the CE cycle for each system without runtime errors in all five runs. It also correctly reconfigures NGINX in all five runs and SOCKSHOP in four out of five runs. Even in the non-reconfigured case, we confirm that a valid CE cycle is completed without requiring reconfigurations.

Qualitative validation To validate CE operations completed by CHAOSEATER, we select one of the five runs for each target system and qualitatively evaluate the four phases and the overall process using a five-point scale. Here, the scale is designed so that a score of 3 or higher is a positive rating. The evaluators are two external human engineers and three LLMs: GPT-4o, Claude Sonnet 3.5, and Gemini Pro 1.5. The LLM evaluators evaluate each CE cycle five times with a temperature of 0, and the final score is calculated as the average of these five evaluations. See Appendix C for details on the evaluation scale and other settings.

Table 1: Time and monetary costs of single CE cycles conducted by CHAOSEATER. The values for each phase are averaged across runs that did not skip that phase, while the values for overall are averaged across runs that involved system reconfiguration. API costs are calculated from the official OpenAI API pricing table in September 2024. Abbreviations of each phase name are as follows: 'All' is the overall process; 'Pre' is *pre-processing*; 'Hyp.' is *hypothesis*; 'Expt.' is *experiment*; 'Anlys.' is *analysis*; 'Imp.' is *improvement*; 'Post' is *post-processing*.

	NGINX							SOCKSHOP						
Metric	All	Pre	Hyp.	Expt.	Anlys.	Imp.	Post	All	Pre	Hyp.	Expt.	Anlys.	Imp.	Post
Input tokens	59k	2.6k	25k	13k	4.4k	5.5k	8.2k	284k	30k	150k	57k	14k	15k	18k
Output tokens	5.9k	0.5k	2.5k	1.7k	0.6k	0.2k	0.4k	13k	5.7k	3.8k	1.8k	0.7k	0.6k	0.5k
API cost ($)	0.21	0.01	0.09	0.05	0.02	0.02	0.02	0.84	0.13	0.41	0.16	0.04	0.04	0.05
Time	11m	21s	2.6m	4.4m	50s	12s	21s	25m	4.6m	4.3m	3.3m	36s	4.3m	21s

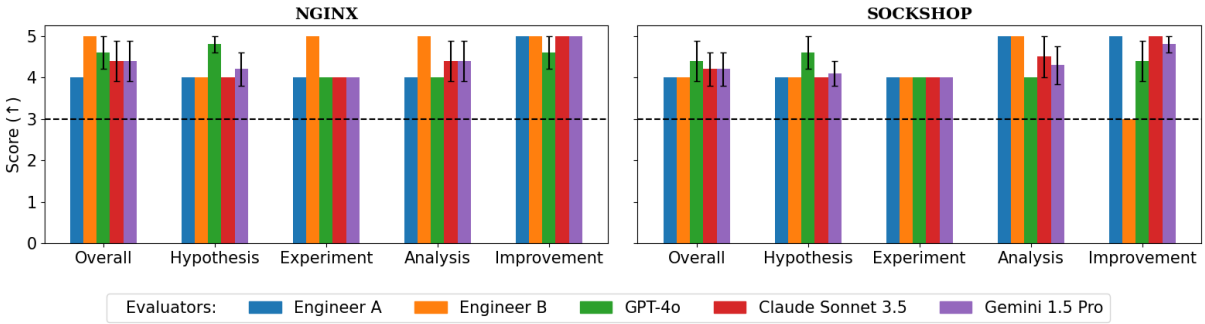


Figure 3: Qualitative evaluation results of CE cycles for each system. A score of 3 or higher is a positive rating.

Figure 2 shows the highlighted outputs from the evaluated cycles for NGINX and SOCKSHOP. For both cases, CHAOSEATER defined pod availability as one of the steady states, and then hypothesized that they are maintained even in scenarios where `Pods` go down, such as cyberattacks or Black Friday sales. Through these experiments, it successfully identified the issues of `restartPolicy` and the number of replicas, and solved them by replacing `Pod` with a `Deployment` resource and increasing the number of replicas, respectively. The authors have qualitatively confirmed that these operations are appropriate for addressing the issues intentionally introduced into the original systems.

Figure 3 shows the results of the qualitative evaluation of the two cycles conducted by the evaluators. The results show that all evaluators rated every phase above the threshold for a positive rating for both systems, demonstrating that CHAOSEATER completed reasonable single CE cycles. In their reviews, they pointed out the need for additional CE cycles to achieve broader resiliency improvement and suggested the inclusion of more complex failure sequences, such as cases where a single failure indirectly affects multiple resources through complex dependencies. However, all of them agreed that, as a single CE cycle addressing the given issues, the operations are effective and appropriate. In conclusion, the validity of the CE cycles is confirmed by the authors, the human engineers, and LLMs. See Appendix E for more detailed descriptions of the CHAOSEATER's outputs and the full text of the reviews.

4 Conclusion

In this paper, we proposed CHAOSEATER, an LLM-based system for fully automating single CE cycles. We presented its technical details and validated it through case studies on small and large K8s systems. The results demonstrated that CHAOSEATER successfully completes reasonable single CE cycles with significantly low time and monetary costs.

On the other hand, the current version has several limitations, such as restriction to development environments, support only for K8s manifest reconfiguration, and limited capability in vulnerability discovery (see Appendix B for comprehensive discussion on limitations and future directions). As the automatic generation of software applications by LLMs has become widespread in recent years, the automation of improving their infrastructure resiliency is becoming increasingly important. To support this, we will improve CHAOSEATER by addressing its current limitations.

Acknowledgements

We would like to thank NTT TechnoCross members, especially Kensuke Yagura, Yoshitaka Sadohara, and Tamotsu Ohyama, for trying out CHAOSEATER and sharing their valuable feedback. We also thank the anonymous reviewers for their constructive feedback and valuable suggestions. Finally, the CHAOSEATER would not have been possible without LLM providers and relevant OSS projects. We deeply appreciate their continuous efforts and dedication.

References

Tasnim Ahmed, Nicola Piovesan, Antonio De Domenico, and Salimur Choudhury. 2024. Linguistic intelligence in large language models for telecommunications.

Peter Alvaro, Kolton Andrus, Chris Sanden, Casey Rosenthal, Ali Basiri, and Lorin Hochstein. 2016. Automating failure testing research at internet scale. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, page 17–28, New York, NY, USA. Association for Computing Machinery.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *Preprint*, arXiv:2108.07732.

Lina Bariah, Hang Zou, Qiyang Zhao, Belkacem Mouhouche, Faouzi Bader, and Merouane Debbah. 2023. Understanding telecom language through large language models. *Preprint*, arXiv:2306.07933.

Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. 2016. Chaos engineering. *IEEE Software*, 33(3):35–41.

Ali Basiri, Lorin Hochstein, Nora Jones, and Haley Tucker. 2019. Automating chaos experiments in production. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 31–40.

Martin Bedoya, Sara Palacios, Daniel Díaz-López, Estefania Laverde, and Pantaleone Nespoli. 2024. Enhancing devsecops practice with large language models and security chaos engineering. *International Journal of Information Security*, 23:3765–3788.

Antonio Bucchiarone, Nicola Dragoni, Schahram Dustdar, Patricia Lago, Manuel Mazzara, Victor Rivera, and Andrey Sadovykh, editors. 2020. *Microservices, Science and Engineering*. Springer.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *Preprint*, arXiv:2207.10397.

Hongyang Chen, Pengfei Chen, Guangba Yu, Xiaoyun Li, and Zilong He. 2024. Microfi: Non-intrusive and prioritized request-level fault injection for microservice applications. *IEEE Transactions on Dependable and Secure Computing*, 21(5):4921–4938.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Yinfang Chen, Manish Shetty, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Jonathan Mace, Chetan Bansal, Rujia Wang, and Saravan Rajmohan. 2025. Aiopslab: A holistic framework to evaluate ai agents for enabling autonomous clouds. *Preprint*, arXiv:2501.06706.

Suman De. 2021. A study on chaos engineering for improving cloud software quality and reliability. In *2021 International Conference on Disruptive Technologies for Multi-Disciplinary Research and Applications (CENTCON)*, volume 1, pages 289–294.

Firefly. 2022. Aiac (github repository).

Google. 2019. Skaffold (github repository).

Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, Saizhuo Wang, Kun Zhang, Yuanzhuo Wang, Wen Gao, Lionel Ni, and Jian Guo. 2025. A survey on llm-as-a-judge. *Preprint*, arXiv:2411.15594.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *Preprint*, arXiv:2401.14196.

Cho-Jui Hsieh, Si Si, Felix Yu, and Inderjit Dhillon. 2024. Automatic engineering of long prompts. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 10672–10685, Bangkok, Thailand. Association for Computational Linguistics.

Shengran Hu, Cong Lu, and Jeff Clune. 2024. Automated design of agentic systems. *Preprint*, arXiv:2408.08435.

Yudong Huang, Hongyang Du, Xinyuan Zhang, Dusit Niyato, Jiawen Kang, Zehui Xiong, Shuo Wang, and Tao Huang. 2023. Large language models for networking: Applications, enabling techniques, and challenges.

Hiroki Ikeuchi, Akio Watanabe, and Yousuke Takahashi. 2023. Coverage based failure injection toward efficient chaos engineering. In *ICC 2023 - IEEE International Conference on Communications*, pages 4571–4577.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *Preprint*, arXiv:2406.00515.

Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. Self-evolve: A code evolution framework via large language models. *Preprint*, arXiv:2306.02907.

K8sGPT. 2023. K8sgpt (github repository).

Imtiaz Karim, Kazi Samin Mubasshir, Mirza Masfiqur Rahman, and Elisa Bertino. 2023. SPEC5G: A dataset for 5G cellular network protocol analysis. In *Findings of the Association for Computational Linguistics: IJCNLP-AACL 2023 (Findings)*, pages 20–38, Nusa Dua, Bali. Association for Computational Linguistics.

Dominik Kesim, André van Hoorn, Sebastian Frank, and Matthias Haussler. 2020. Identifying and prioritizing chaos experiments by using established risk analysis techniques. pages 229–240.

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2023. Decomposed prompting: A modular approach for solving complex tasks. In *The Eleventh International Conference on Learning Representations*.

Kubernetes. 2014. Kubernetes (github repository).

Kubernetes. 2018. kind (github repository).

Minchan Kwon, Gaeun Kim, Jongsuk Kim, Haeil Lee, and Junmo Kim. 2024. StablePrompt : Automatic prompt tuning using reinforcement learning for large language model. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 9868–9884, Miami, Florida, USA. Association for Computational Linguistics.

LangChain. 2023. Langchain (github repository).

Zelong Li, Shuyuan Xu, Kai Mei, Wenyue Hua, Balaji Rama, Om Raheja, Hao Wang, He Zhu, and Yongfeng Zhang. 2024. Autoflow: Automated workflow generation for large language model agents. *Preprint*, arXiv:2407.12821.

Ali Maatouk, Fadhel Ayed, Nicola Piovesan, Antonio De Domenico, Merouane Debbah, and Zhi-Quan Luo. 2023. Teleqna: A benchmark dataset to assess large language models telecommunications knowledge. *Preprint*, arXiv:2310.15051.

Ehud Malul, Yair Meidan, Dudu Mimran, Yuval Elovici, and Asaf Shabtai. 2024. Genkubesec: Llm-based kubernetes misconfiguration detection, localization, reasoning, and remediation. *Preprint*, arXiv:2405.19954.

Yukai Miao, Yu Bai, Li Chen, Dan Li, Haifeng Sun, Xizheng Wang, Ziqiu Luo, Yanyu Ren, Dapeng Sun, Xiuting Xu, Qi Zhang, Chao Xiang, and Xinchi Li. 2023. An empirical study of netops capability of pre-trained large language models. *Preprint*, arXiv:2309.05557.

Microsoft. 2023. Azure chaos studio (product page).

Netflix. 2012. Chaos monkey (github repository).

OpenAI. 2024. Gpt-4o system card. *Preprint*, arXiv:2410.21276.

Nicola Piovesan, Antonio De Domenico, and Fadhel Ayed. 2024. Telecom language models: Must they be large? *Preprint*, arXiv:2403.04666.

Prometheus. 2012. Prometheus (github repository).

Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. 2023. Automatic prompt optimization with "gradient descent" and beam search. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

Pulumi. 2023. Pulumi ai (github repository).

Robusta. 2023. Kubernetes chatgpt bot (github repository).

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Ojaswa Sharma, Mudit Verma, Saumya Bhadauria, and Praveen Jayachandran. 2022. A guided approach towards complex chaos selection, prioritisation and injection. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 91–93.

Snowflake. 2020. Streamlit (github repository).

Hong Sun, Xue Li, Yinchuan Xu, Youkow Homma, Qi Cao, Min Wu, Jian Jiao, and Denis Charles. 2023. Autohint: Automatic prompt optimization with hint generation. *Preprint*, arXiv:2307.07415.

Amazon Web Services. 2021. Aws fault injection service (product page).

Chaos Mesh. 2021. Chaos mesh (github repository).

Cognition Labs. 2024. Introducing devin, the first ai software engineer (official blog).

Grafana Labs. 2021. k6 (github repository).

Kennedy A. Torkura, Muhammad I.H. Sukmana, Feng Cheng, and Christoph Meinel. 2019. Security chaos engineering for cloud services: Work in progress. In *2019 IEEE 18th International Symposium on Network Computing and Applications (NCA)*, pages 1–3.

Weaveworks. 2023. Sock shop (github repository (archived in 2023)).

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *Preprint*, arXiv:2407.01489.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. 2024a. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024b. Swe-agent: Agent-computer interfaces enable automated software engineering. *Preprint*, arXiv:2405.15793.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. 2024. Aflow: Automating agentic workflow generation. *Preprint*, arXiv:2410.10762.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A survey of large language models.

Wangchunshu Zhou, Yixin Ou, Shengwei Ding, Long Li, Jialong Wu, Tiannan Wang, Jiamin Chen, Shuai Wang, Xiaohua Xu, Ningyu Zhang, Huajun Chen, and Yuchen Eleanor Jiang. 2024. Symbolic learning enables self-evolving agents. *Preprint*, arXiv:2406.18532.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. 2024. GPTSwarm: Language agents as optimizable graphs. In *Forty-first International Conference on Machine Learning*.

Sertaç Özercan. 2023. Kubectl ai (github repository).

A Related Work

Chaos engineering Since Basiri et al. (2016) introduced its name, CE has gained attention and is currently employed in various services (Basiri et al., 2019; De, 2021). The major research question of CE is how we can efficiently test meaningful failures to identify system vulnerability. To address this, various optimization methods for failure selection have been proposed (Alvaro et al., 2016; Torkura et al., 2019; Kesim et al., 2020; Sharma et al., 2022; Ikeuchi et al., 2023; Chen et al., 2024). In application, several automation tools have been developed in both the open-source community (Chaos Mesh, 2021; Netflix, 2012) and the commercial sector (Amazon Web Services, 2021; Microsoft, 2023). While these technologies have advanced CE automation, its full automation has not yet been achieved due to the complexity of generative tasks, such as hypothesis formulation and system reconfiguration. Our work is the first to fully automate CE using LLMs.

LLMs for software engineering LLMs for coding have been actively explored from various aspects: Pretraining models (Chen et al., 2021; Rozière et al., 2024; Guo et al., 2024), prompt engineering (Chen et al., 2022; Jiang et al., 2023), and evaluation (Austin et al., 2021; Chen et al., 2021). For more general SE tasks, LLMs that solve issues in GitHub repositories have also emerged (Yang et al., 2024b; Cognition Labs, 2024; Xia et al., 2024). Since CE for software systems are regarded as SE, our work can be considered a part of this trend. Unlike existing SE benchmarks, CE requires autonomously defining and achieving its own goals. Our work demonstrates the potential of LLMs to tackle such complex SE tasks that are new to them.

LLMs for networking (NW) LLMs for networking have also been explored from various aspects in recent years: Datasets (Bariah et al., 2023; Karim et al., 2023), benchmarks (Maatouk et al., 2023; Miao et al., 2023), fine-tuned models (Bariah et al., 2023), an agent framework for NW-related tasks (Huang et al., 2023), and comprehensive evaluation (Ahmed et al., 2024; Piovesan et al., 2024). These works empirically demonstrate the promise of applying LLMs to the NW domain. In parallel with the research side, various NW applications have also been developed, especially for software systems. They range from LLM-based IaC code generation (Firefly, 2022; Özercan, 2023; Pulumi,

2023) to diagnostic tools (K8sGPT, 2023; Robusta, 2023) and misconfiguration remediation (Malul et al., 2024). Despite the advancements of LLMs in the NW domain, their application to CE remains unexplored. Our work is the first to demonstrate the capabilities of LLMs in CE, which involves complex NW operations.

Concurrent works In parallel with our work, some other projects have shown promising results in applying LLMs to CE. From a security perspective, Bedoya et al. (2024) leverage LLMs to construct attack-defense trees, which assist security analysts in designing security chaos experiments. The major differences from ours lie in their use of LLMs as a supplementary support tool and designing the LLM workflow specialized to the *hypothesis* phase based on the methodology of security CE. AIOpsLab (Chen et al., 2025) is an evaluation framework for LLM agents that automates the operations of cloud (i.e., software) systems. It includes failure injection as a feature and allows us to evaluate the capabilities of LLM agents in CE. The supported failures include both *functional failures*, such as system misconfigurations, and independent *symptomatic failures*, such as server downtime and network delays. However, while it evaluates whether LLM agents can provide appropriate cause analyses and solutions for functional failures, the evaluation for symptomatic failures is limited to identifying the presence of failure and pinpointing their locations. On the other hand, CHAOSEATER focuses on the symptomatic failures and is capable of performing more detailed and comprehensive CE by following the systematic CE cycle. For example, it autonomously makes a appropriate hypothesis for a given system, designs complex failure scheduling to simulate symptomatic failure scenarios, analyzes system behavior during symptomatic failure injection, and enables reconfigurations to address unexpected behavior due to the failures. While these concurrent works, including ours, share a common goal, each focuses on a different aspect of CE.

B Discussion

Broader impacts Numerous systems, including the increasing number of LLM applications in recent years, are built in the microservice architecture, and their number is expected to continue to grow in the future. By fully automating CE, it will be possible for anyone to easily build resilient systems. Moreover, it is also expected to combine CHAOSEATER with other LLM systems for creating software applications, such as improving the resiliency of applications created by other LLM systems through CHAOSEATER. Although CHAOSEATER is not yet at a practical level, we believe that CHAOSEATER would be a good starting point toward such use cases. Even at its current level, CHAOSEATER can be sufficiently used as training materials (including both good and bad practices) for the Chaos Game Day, which is a training exercise for CE engineers.

Limitations CHAOSEATER currently has the following limitations:

1. **Limited deployment environment**; Although CE should ideally be conducted in actual production environments, CHAOSEATER is currently only supported in development environments.
2. **Limited to GPT-4o**; CHAOSEATER's prompt templates are highly tuned only for GPT-4o. Therefore, other LLMs can not currently be used for CHAOSEATER.
3. **Limited to K8s manifest reconfiguration**; Software systems consist not only of K8s manifests but also of other types of codebases, such as HTML/CSS/JS and Python. Although K8s manifest reconfiguration can handle a majority of system resiliency issues, reconfiguration of all types of codebases is necessary to optimally improve system resiliency. However, CHAOSEATER currently supports reconfiguring only K8s manifests.
4. **Vulnerability discovery**; In the case study, CHAOSEATER improved systems with relatively simple resiliency issues. However, for systems that already possess a certain level of resiliency, CHAOSEATER fails to find new hidden issues through a CE cycle. Given that this is a challenging task even for human engineers, CHAOSEATER is currently considered to perform at a level comparable to, or lower than, that of engineers. To find such issues, it is necessary to conduct multiple CE cycles over extended operational periods.

Future directions Given the current limitations above, we share several future directions for CHAOSEATER and the full automation of system resiliency improvement:

1. **Production deployment and security**; If CHAOSEATER is deployed in production en-

vironments, further research on security will be necessary. This includes controlling more carefully the impact range of failures (i.e., blast radius), preventing CHAOSEATER from becoming a proxy for attacking production services, and proposing emergency response measures, such as a higher-level monitoring system that always monitors CHAOSEATER.

2. **Support for various LLMs**; As CHAOSEATER's prompt templates are tuned manually, supporting various LLMs significantly increases their management costs. To address this, automatic prompt tuning is considered an effective solution. Our current prompt templates may be used as the seed prompts.
3. **Fine-tuning LLMs specifically for CE**; Fine-tuning is necessary to improve the quality of CE cycles and expand supported LLM types. CHAOSEATER's outputs may be used as the instruction-tuning data.
4. **Evaluation frameworks**; As there are currently no datasets and benchmarks for the systematic CE cycle, we will construct them to enable more solid validation of CHAOSEATER. Besides, we plan to propose new metrics for quantitatively evaluating CE cycles conducted by CHAOSEATER. This is not easy because, even in cases where no improvements are made, CE cycles can still provide valuable insights. Therefore, the quality of CE cycles should not be judged solely based on whether improvements were made; metrics that consider its philosophical aspects are also necessary.
5. **Toward larger and more complex systems**; We need to incorporate the recent advances in the combination of LLMs and graphs to extract necessary sub-graphs from large system graphs. This sub-graph extraction is important to organize the agent's inputs in each phase.
6. **Full automation of long-term multiple CE cycles**; By using the CHAOSEATER's output as input for the next CE cycle, we can automate multiple CE cycles even with the current CHAOSEATER. However, we additionally need to develop techniques to manage the long-term history of completed CE cycles and continuous learning (if LLMs are fine-tuned).

C Evaluation Details

C.1 Parameters of LLM agents

The parameters of LLM agents are as follows: the temperature of LLMs to 0 with a random seed of 42; the maximum number of proposed steady states is set to 2; and the maximum number of iterations for both the verification improvement loops is set to 3. All other parameters of LLMs are set to their default values.

C.2 Quantitative Evaluation

Tokens and API costs We first count the input and output tokens using the tokenizer of GPT-4o (`o200k_base`). We then calculate the monetary costs based on the official OpenAI API pricing table in September 2024: $2.50 per 1 million input tokens and $10.00 per 1 million output tokens.

Running time We measure the running time using the `time` module in Python. We implement each phase as a separate Python class and measure the running time as the duration from the start to the termination of each class call. The specifications of the device used for the evaluation are as follows: AMD Epyc Milian x 2 (CPU) and MEM-DR432LC-ER32 DDR4-3200 x 32 (Memory).

C.3 Qualitative Evaluation

Evaluation scale Table 2 shows the evaluation scale. The criteria are designed so that a score of 3 or above is considered a positive rating. For each phase, a score of 3 corresponds to meeting the minimum required criteria, while scores of 4 and above are incrementally awarded for additional elements, such as contributions to the next CE cycle, quality or efficiency. On the other hand, scores of 1 or 2 correspond to operations that are either meaningless or that lead to undesirable modifications to the system. To determine these concrete criteria, we referred to the essential elements of CE mentioned in (Basiri et al., 2016).

Evaluators We employ two human engineers and three state-of-the-art LLMs. The human engineers are recruited from external IT companies across different countries. The LLM evaluators include gpt-4o-2024-0908-06 (GPT-4o), claude-3-5-sonnet-20240620 (Claude Sonnet 3.5), and gemini-1.5-pro (Gemini 1.5 pro). Following (Gu et al., 2025), the temperature is set to 0 for all LLMs to improve the consistency and reproducibility of

Table 2: Evaluation scale for each phase. The criteria are designed so that a score of 3 or higher is positive.

Phase	Score	Criteria
Overall	5	The cycle fixes critical issues in the system and offers meaningful insights for the next cycle according to the experiments conducted.
	4	The cycle fixes critical issues in the system.
	3	The cycle fixes minor issues in the system or offers meaningful insights for the next cycle according to the experiments conducted.
	2	The cycle neither changes the system nor offers meaningful insights for the next cycle according to the experiments conducted.
	1	The cycle worsens the system's resiliency or adds meaningless resiliency.
Hypothesis	5	The hypothesis is relevant to the system and meaningful. Additioanlly, the hypothesis leads to system improvement and offers meaningful insights for the next cycle.
	4	The hypothesis is relevant to the system and meaningful. Additionally, the hypothesis leads to system improvement or offers meaningful insights for the next cycle.
	3	The hypothesis is relevant to the system and meaningful. However, the hypothesis neither leads to system improvement nor offers meaningful insights for the next cycle.
	2	The hypothesis is relevant to the system, but is trivial and meaningless (e.g., hypothesis that is to be obviously satisfied).
	1	The hypothesis is irrelevant to the system.
Experiment	5	The experiment plan correctly serves to validate the hypothesis. Additionally, it is set up considering an complex, actual failure scenario.
	4	The experiment plan correctly serves to validate the hypothesis. Additionally, it is set up considering an actual failure scenario.
	3	The experiment plan correctly serves to validate the hypothesis.
	2	The experiment plan mostly serves to validate the hypothesis. However, there are some missed components.
	1	The experiment plan does serve to validate the hypothesis at all.
Analysis	5	The analysis reports correct and meaningful information. Additioanlly, it provides some meaningful insights for the improvement.
	4	The analysis reports correct and meaningful information. Additioanlly, it provides some insights for the improvement.
	3	The analysis reports correct and meaningful information.
	2	The analysis reports meaningless information.
	1	The analysis reports information that is not factual.
Improvement	5	The improvement succesully changes the system to satisfy the hypothesis in the first attempt.
	4	The improvement succesully changes the system to satisfy the hypothesis over two or more iterations.
	3	The improvement does not change the system.
	2	The improvement exceeded the number of attempts and stopped midway.
	1	The improvement worsens the system's resiliency or adds meaningless resiliency.

the evaluations. As mentioned earlier, all other parameters remain at their default values.

Evaluation instruction We give a system prompt for this qualitative evaluation to LLMs, where they are instructed to output the score and its corresponding reason for each phase in JSON format. In this format, the reason comes before the score. Prompt 1 and Example 1 show the system prompt and an example of text embedded in the placeholder `{ce_cycle_overview}`, respectively. This embedded overview is generated by embedding the outputs of CHAOSEATER into a static tem-

plate. We give almost the same instruction to the human engineers. Additionally, before the evaluation, we obtained their consent to use the evaluation results for the purpose of evaluating CHAOSEATER and for inclusion in this paper. They were also cautioned to avoid any evaluation bias that might arise from disclosing the purpose in advance.

Prompt 1: System prompt for reviewing each phase of completed CE cycles

System:

```
You are a professional reviewer for Chaos
Engineering.
Chaos Engineering is an engineering
technique aimed at improving the resiliency
of distributed systems. It involves
artificially injecting specific failures
into a distributed system and observing its
behavior in response. Based on the
observation, the system can be proactively
improved to handle those failures.
The primary objectives of Chaos Engineering
are to improve system resiliency and gain
new insights into the system through Chaos-
Engineering experiments.
Systematically, Chaos Engineering cycles
through four phases: hypothesis, experiment,
 analysis, and improvement phases.
  1) Hypothesis: Define steady states (i.e.,
   normal behavior) of the system and
  injected failures (i.e., faults). Then,
  make a hypothesis that "the steady states
  are maintained in the system even when the
   failures are injected".
  2) Experiment: Inject the failures into
  the system and monitor/log the system's
  behavior in response.
  3) Analysis: Analyze the logged data and
  check if the hypothesis is satisfied. If
  so, one CE cycle is finished here. If not,
   move to (4)
  4) Improvement: Reconfigure the system to
  satisfy the hypothesis. The reconfigured
  system is tested again in (2) and (3), i.e
  ., repeat (2) to (4) until the hypothesis
  is satisfied.
Given a Chaos Engineering cycle, you will
carefully review it according to the
following rules:
- The review must be specific, constructive,
 and insightful.
- The output should be formatted as a JSON
instance that conforms to the JSON schema
below.

As an example, for the schema {"properties":
 {"foo": {"title": "Foo", "description": "a
list of strings", "type": "array", "items":
{"type": "string"}}}, "required": ["foo"]}
the object {"foo": ["bar", "baz"]} is a well
-formatted instance of the schema. The
object {"properties": {"foo": ["bar", "baz
"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "hypothesis": {
      "title": "Hypothesis",
      "description": "The review of the
      hypothesis phase.",
      "allOf": [{"$ref": "#/definitions/
      HypothesisReview"}]
    },
    "experiment": {
      "title": "Experiment",
      "description": "The review of the
      experiment (planning) phase.",
      "allOf": [{"$ref": "#/definitions/
      ExperimentReview"}]
    },
    "analysis": {
      "title": "Analysis",
      "description": "The review of the
      analysis phase.",
      "allOf": [{"$ref": "#/definitions/
      AnalysisReview"}]
    },
    "improvement": {
      "title": "Improvement",
      "description": "The review of the
      improvement phase.",
      "allOf": [{"$ref": "#/definitions/
      ImprovementReview"}]
    },
    "overall": {
      "title": "Overall",
      "description": "The review of the
      entire Chaos Engineering cycle.",
      "allOf": [{"$ref": "#/definitions/
      OverallReview"}]
    }
  },
  "required": ["hypothesis", "experiment", "
  analysis", "improvement", "overall"],
  "definitions": {
    "HypothesisReview": {
      "title": "HypothesisReview",
      "type": "object",
      "properties": {
        "summary": {
          "title": "Summary",
          "description": "Summarize the
          overall of the hypothesis.",
          "type": "string"
        },
        "strengths": {
          "title": "Strengths",
          "description": "List the strengths
           of the hypothesis. Write them in
          bullet points (3 - 5 items).",
          "type": "string"
        },
        "weaknesses": {
          "title": "Weaknesses",
          "description": "List the
          weaknesses of the hypothesis.
          Write them in bullet points (3 - 5
           items).",
          "type": "string"
        },
        "score_reason": {
          "title": "Score Reason",
          "description": "Before write the
          score, please describe why you
          choose the score according to the
          rating scale table.",
          "type": "string"
        },
        "score": {
          "title": "Score",
          "description": "Here is the rating
           scale (descending order). You
          must choose the score from [1,
          5].\nThe higher the score, the
          better the hypothesis. The scores
          1, 2 are negative, The score 3 is
          accetable, the score 4 is positive
          , the score 5 is very positive.\
          nScore: criteria\n5: The
          hypothesis is relevant to the
          system and meaningful.
          Additioanlly, the hypothesis leads
           to system improvement and offers
          meaningful insights for the next
          cycle. \n4: The hypothesis is
          relevant to the system and
          meaningful. Additionally, the
          hypothesis leads to system
          improvement or offers meaningful
          insights for the next cycle.\n3:
```
```

```
        The hypothesis is relevant to the
        system and meaningful. However,
        the hypothesis neither leads to
        system improvement nor offers
        meaningful insights for the next
        cycle.\n2: The hypothesis is
        relevant to the system, but is
        trivial and meaningless (e.g.,
        hypothesis that is to be obviously
         satisfied).\n1: The hypothesis is
         irrelevant to the system.\n",
        "enum": [1, 2, 3, 4, 5],
        "type": "integer"
      }
    },
    "required": ["summary", "strengths", "
    weaknesses", "score_reason", "score"]
  },
  "ExperimentReview": {
    "title": "ExperimentReview",
    "type": "object",
    "properties": {
      "summary": {
        "title": "Summary",
        "description": "Summarize the
        overall of the experiment plan.",
        "type": "string"
      },
      "strengths": {
        "title": "Strengths",
        "description": "List the strengths
         of the experiment plan. Write
        them in bullet points (3 - 5 items
        ).",
        "type": "string"
      },
      "weaknesses": {
        "title": "Weaknesses",
        "description": "List the
        weaknesses of the experiment plan.
         Write them in bullet points (3 -
        5 items).",
        "type": "string"
      },
      "score_reason": {
        "title": "Score Reason",
        "description": "Before write the
        score, please describe why you
        choose the score according to the
        rating scale table.",
        "type": "string"
      },
      "score": {
        "title": "Score",
        "description": "Here is the rating
         scale (descending order). You
        must choose the score from [1,
        5].\nThe higher the score, the
        better the experiment plan.\n5:
        The experiment plan correctly
        serves to validate the hypothesis.
         Additionally, it is set up
        considering an complex, actual
        failure scenario.\n4: The
        experiment plan correctly serves
        to validate the hypothesis.
        Additionally, it is set up
        considering an actual failure
        scenario.\n3: The experiment plan
        correctly serves to validate the
        hypothesis.\n2: The experiment
        plan mostly serves to validate the
         hypothesis. However, there are
        some missed components.\n1: The
        experiment plan does serve to
        validate the hypothesis at all.\n
        ",
        "enum": [1, 2, 3, 4, 5],
        "type": "integer"
      }
    },
    "required": ["summary", "strengths", "
    weaknesses", "score_reason", "score"]
  },
  "AnalysisReview": {
    "title": "AnalysisReview",
    "type": "object",
    "properties": {
      "summary": {
        "title": "Summary",
        "description": "Summarize the
        overall of the analysis.",
        "type": "string"
      },
      "strengths": {
        "title": "Strengths",
        "description": "List the strengths
         of the analysis. Write them in
        bullet points (3 - 5 items).",
        "type": "string"
      },
      "weaknesses": {
        "title": "Weaknesses",
        "description": "List the
        weaknesses of the analysis. Write
        them in bullet points (3 - 5 items
        ).",
        "type": "string"
      },
      "score_reason": {
        "title": "Score Reason",
        "description": "Before write the
        score, please describe why you
        choose the score according to the
        rating scale table.",
        "type": "string"
      },
      "score": {
        "title": "Score",
        "description": "Here is the rating
         scale (descending order). You
        must choose the score from [1,
        5].\nThe higher the score, the
        better the analysis.\n5: The
        analysis reports correct and
        meaningful information.
        Additioanlly, it provides some
        meaningful insights for the
        improvement.\n4: The analysis
        reports correct and meaningful
        information. Additioanlly, it
        provides some insights for the
        improvement.\n3: The analysis
        reports correct and meaningful
        information.\n2: The analysis
        reports meaningless information.\
        n1: The analysis reports
        information that is not factual.\n
        ",
        "enum": [1, 2, 3, 4, 5],
        "type": "integer"
      }
    },
    "required": ["summary", "strengths", "
    weaknesses", "score_reason", "score"]
  },
  "ImprovementReview": {
    "title": "ImprovementReview",
    "type": "object",
    "properties": {
      "summary": {
        "title": "Summary",
        "description": "Summarize the
        overall of the improvement.",
        "type": "string"
      },
      "strengths": {
        "title": "Strengths",
        "description": "List the strengths
         of the improvement. Write them in
         bullet points (3 - 5 items).",
        "type": "string"
      },
      "weaknesses": {
        "title": "Weaknesses",
        "description": "List the
        weaknesses of the improvement.
        Write them in bullet points (3 - 5
         items).",
        "type": "string"
```

```
        },
        "score_reason": {
          "title": "Score Reason",
          "description": "Before write the
          score, please describe why you
          choose the score according to the
          rating scale table.",
          "type": "string"
        },
        "score": {
          "title": "Score",
          "description": "Here is the rating
           scale (descending order). You
          must choose the score from [1,
          5].\nThe higher the score, the
          better the improvement. \n5: The
          improvement succesully changes the
           system to satisfy the hypothesis
          in the first attempt.\n4: The
          improvement succesully changes the
           system to satisfy the hypothesis
          over two or more iterations.\n3:
          The improvement does not change
          the system.\n2: The improvement
          exceeded the number of attempts
          and stopped midway.\n1: The
          improvement worsens the system's
          resiliency or adds meaningless
          resiliency.\n",
          "enum": [1, 2, 3, 4, 5],
          "type": "integer"
        }
      },
      "required": ["summary", "strengths", "
      weaknesses", "score_reason", "score"]
    },
    "OverallReview": {
      "title": "OverallReview",
      "type": "object",
      "properties": {
        "summary": {
          "title": "Summary",
          "description": "Summarize the
          overall of the Chaos Engineering
          cycle.",
          "type": "string"
        },
        "strengths": {
          "title": "Strengths",
          "description": "List the strengths
           of the Chaos Engineering cycle.
          Write them in bullet points (3 - 5
           items).",
          "type": "string"
        },
        "weaknesses": {
          "title": "Weaknesses",
          "description": "List the
          weaknesses of the Chaos
          Engineering cycle. Write them in
          bullet points (3 - 5 items).",
          "type": "string"
        },
        "score_reason": {
          "title": "Score Reason",
          "description": "Before write the
          score, please describe why you
          choose the score according to the
          rating scale table.",
          "type": "string"
        },
        "score": {
          "title": "Score",
          "description": "Here is the rating
           scale (descending order). You
          must choose the score from [1, 5].
           \nThe higher the score, the
          better the CE cycle. The scores 1,
           2 are negative, The score 3 is
          accetable, the score 4 is positive
          , the score 5 is very positive.\
          nScore: criteria\n5: The cycle
          fixes critical issues in the
          system and offers meaningful
          insights for the next cycle
          according to the experiments
          conducted\n4: The cycle fixes
          critical issues in the system\n3:
          The cycle fixes minor issues in
          the system or offers meaningful
          insights for the next cycle
          according to the experiments
          conducted\n2: The cycle neither
          changes the system nor offers
          meaningful insights for the next
          cycle according to the experiments
           conducted\n1: The cycle worsens
          the system's resiliency or adds
          meaningless resiliency.\n",
          "enum": [1, 2, 3, 4, 5],
          "type": "integer"
        }
      },
      "required": ["summary", "strengths", "
      weaknesses", "score_reason", "score"]
    }
  }
}
```

User:
Here is the overview of a Chaos Engineering cycle to be reviewed:
{ce_cycle_overview}

Please review the above Chaos Engineering cycle.

Example 1: ce_cycle_overview

```
# Here is a Chaos Engineering cycle
## Step 0. User-input understanding
### Here is the overview of user inputs:
The system consists of the following K8s
manifest(s):K8s manifest: nginx/pod.yaml
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
  labels:
    app: example
spec:
  restartPolicy: Never
  containers:
  - name: example-container
    image: nginx:1.17.1
    ports:
    - containerPort: 80
```
Summary of nginx/pod.yaml:
- This manifest defines a Kubernetes Pod.
- The Pod is named 'example-pod'.
- It includes metadata with a label 'app:
example'.
- The Pod's restart policy is set to 'Never
', meaning it won't restart automatically if
 it fails.
- The Pod contains one container named '
example-container'.
- The container uses the 'nginx:1.17.1'
image, which is a specific version of the
Nginx web server.
- The container exposes port 80, which is
commonly used for HTTP traffic.

K8s manifest: nginx/service.yaml
```yaml
apiVersion: v1
kind: Service
metadata:
  name: example-service
spec:
  selector:
    app: example
  ports:
    - protocol: TCP
```
```

```
      port: 80
      targetPort: 80
```
Summary of nginx/service.yaml:
- This manifest defines a Kubernetes Service
.
- The Service is named 'example-service'.
- It uses the 'v1' API version.
- The Service selects pods with the label '
app: example'.
- It exposes the Service on port 80 using
the TCP protocol.
- The Service forwards traffic to the target
 port 80 on the selected pods.

The resiliency issues/weaknesses in the
system are as follows:
Issue #0: Pod Restart Policy
  - details: The Pod will not restart
  automatically if it fails, which can lead
  to downtime.
  - manifests having the issues: ['nginx/pod
  .yaml']
  - problematic config: restartPolicy: Never

Issue #1: Single Pod Deployment
  - details: Having a single Pod means there
   is no redundancy. If the Pod fails, the
  service will be unavailable.
  - manifests having the issues: ['nginx/pod
  .yaml']
  - problematic config: Only one Pod is
  defined without a Deployment or ReplicaSet
  .

The expected type of application on the
system (i.e., K8s manfests):
A simple web server application using Nginx
to serve HTTP content.; The manifests
provided define a Kubernetes Pod and a
Service. The Pod runs an Nginx container,
which is a popular web server used to serve
static content, reverse proxy, or load
balance HTTP traffic. The Service is
configured to expose this Pod on port 80,
which is the default port for HTTP traffic.
Given the use of Nginx and the configuration
 of the Service, it is logical to assume
that these manifests are intended to deploy
a simple web server application. The file
names and the use of Nginx further support
this assumption.

Chaos-Engineering instructions for the
system are as follows: - The Chaos-
Engineering experiment must be completed
within 1 minute.

Step 1. Hypothesis definition
Here is the overview of the hypothesis
for the system:
The hypothesis is "The steady states of the
sytem are maintained even when the fault
scenario occurs (i.e., when the faults are
injected)".
The steady states here are as follows:
2 steady states are defined.
1st steady state:
- Name: example-pod-running
- Description: The first issue to address is
 the Pod's restart policy set to 'Never'.
This is a critical issue because if the Pod
fails, it will not restart, leading to
potential downtime. Therefore, the steady
state should ensure that the Pod is running
and available. A measurable output for this
steady state is the number of running Pods,
which should be 1, as there is only one Pod
defined in the manifest. This steady state
will help verify that the Pod is up and
running, which is crucial given the restart
policy configuration.
- Threshold for the steady state: The pod
should be running at least 90% of the time
during the check period.; The steady state
we are considering is whether the 'example-
pod' is running. The current state shows
that the pod was checked 5 times over a
duration of 5 seconds, and it was running
each time, resulting in a running count of
5. This indicates that the pod is
consistently running during the check period
. Given the constraints of the chaos
engineering experiment, which must be
completed within 1 minute, we can set a
threshold that allows for some tolerance in
case of brief fluctuations. A reasonable
threshold would be that the pod should be
running at least 90% of the time during the
check period. This allows for a small margin
 of error while still ensuring that the pod
is generally available and running.
- Whether the steady state meets the
threshold is determined by the following
Python script with K8s API:
```
import os
import time
import argparse
from kubernetes import client, config
from unittest_base import K8sAPIBase

class TestPodRunningState(K8sAPIBase):
    def __init__(self):
        super().__init__()

    def check_pod_status(self, namespace,
    pod_name):
        try:
            pod = self.v1.
            read_namespaced_pod(name=
            pod_name, namespace=namespace)
            return pod.status.phase == '
            Running'
        except client.exceptions.
        ApiException as e:
            print(f"Exception when calling
            CoreV1Api->read_namespaced_pod:
            {e}")
            return False

    def test_pod_running_state(self,
    duration):
        namespace = 'default'
        pod_name = 'example-pod'
        running_count = 0

        # Check the pod status every second
        for the specified duration
        for _ in range(duration):
            if self.check_pod_status(
            namespace, pod_name):
                running_count += 1
            time.sleep(1)

        # Calculate the running percentage
        running_percentage = (running_count
        / duration) * 100

        # Assert that the running percentage
         is at least 90%
        assert running_percentage >= 90, f"
        Pod '{pod_name}' running percentage
        is below threshold: {
        running_percentage}%"

        print(f"Pod '{pod_name}' running
        status checked {duration} times.
        Running percentage: {
        running_percentage}%.")

def main():
    parser = argparse.ArgumentParser(
    description='Test if a pod is running at
     least 90% of the time.')
    parser.add_argument('--duration', type=
    int, default=5, help='Duration to check
    the pod status in seconds.')
    args = parser.parse_args()
```

```
    test = TestPodRunningState()
    test.test_pod_running_state(args.
    duration)

if __name__ == '__main__':
    main()
```

2nd steady state:
- Name: example-service-availability
- Description: The next issue to address is the 'Single Pod Deployment', which is related to the lack of redundancy. This is a significant issue because if the single Pod fails, the service will be unavailable. To verify this, we can define a steady state that checks the availability of the service itself. A measurable output for this steady state is the service's response time or availability. Since the service is exposed on port 80, we can check if the service is responding to HTTP requests. This steady state will help verify that the service is available and responsive, which is crucial given the single Pod deployment configuration.
- Threshold for the steady state: Service availability should be at least 99.9% with a response status of 200.; The steady state we are considering is the availability of the 'example-service'. The k6 test results show that the service is currently responding with a 200 status code for all requests, indicating 100% availability. Given that the system consists of a single Pod, any failure in the Pod would result in the service being unavailable. Therefore, the threshold should ensure that the service remains available and responsive. To account for minor fluctuations and network latency, a reasonable threshold would be to maintain a high availability percentage, slightly below 100% to allow for brief, non-critical failures. A threshold of 99.9% availability is a common standard for web services, allowing for some tolerance while still ensuring high reliability.
- Whether the steady state meets the threshold is determined by the following K6 Javascript:
```
import http from 'k6/http';
import { check } from 'k6';

export const options = {
  vus: 1,
  duration: '5s',
  thresholds: {
    // Ensure that the service availability
    is at least 99.9%
    'http_req_failed': ['rate<=0.001'], //
    0.1% failure rate corresponds to 99.9%
    availability
  },
};

export default function () {
  const res = http.get('http://example-
  service.default.svc.cluster.local:80');
  check(res, {
    'status is 200': (r) => r.status ===
    200,
  });
}
```

The fault scenario here is as follows:

An assumed fault scenario is as follows:
- Event: Cyber Attack
- Used Chaos Engineering tool: Chaos Mesh
- Faults to simulate the event: [[Fault(name='PodChaos', name_id=0, params={'action': 'pod-kill', 'mode': 'one', 'selector': {'namespaces': ['default'], 'labelSelectors': {'app': 'example'}}})], [Fault(name='NetworkChaos', name_id=1, params={'action': 'delay', 'direction': 'to', 'target': {'mode': 'all', 'selector': {'namespaces': ['default'], 'labelSelectors': {'app': 'example'}}}, 'mode': 'all', 'selector': {'namespaces': ['default'], 'labelSelectors': {'app': 'example'}}, 'device': 'eth0', 'delay': {'latency': '100ms', 'jitter': '10ms', 'correlation': '50'}})]]
- Description: Given the system's weaknesses, a cyber attack targeting the single Pod and its network could be highly impactful. The Pod's restart policy set to 'Never' means that if the Pod fails, it will not restart, leading to downtime. Additionally, the single Pod deployment means there is no redundancy, so any failure will make the service unavailable. To simulate a cyber attack, we can start by injecting a PodChaos fault to kill the Pod, testing the system's ability to handle Pod failures. This will directly exploit the lack of redundancy and the restart policy issue. Next, we can simulate a network attack using NetworkChaos to introduce network latency, testing the service's ability to maintain availability under network stress. This sequence simulates a cyber attack by first taking down the Pod and then stressing the network, revealing the system's vulnerabilities in handling such scenarios.

Step 2.1. Chaos-Engineering experiment
Here is the overview of my Chaos-Engineering experiment to verify the hypothesis:
The entire time schedule of the Chaos-Engineering experiment is as follows (The experiment is divided into three phases: pre-validation, fault-injection, and post-validation phases):
Given the constraints of the chaos engineering experiment, which must be completed within 1 minute, we need to allocate time efficiently across the three phases: pre-validation, fault-injection, and post-validation. The pre-validation phase is crucial to ensure that the system is in a steady state before we introduce any faults. Since we have two steady states to validate, we should allocate a reasonable amount of time to check both the pod's running status and the service's availability. A duration of 15 seconds should be sufficient for pre-validation, allowing us to run the necessary checks multiple times. The fault-injection phase is where we introduce the chaos to observe the system's behavior under stress. Given the complexity of the faults (PodChaos and NetworkChaos), we should allocate the majority of the time to this phase to ensure that the faults have enough time to manifest and impact the system. A duration of 30 seconds is appropriate for fault injection, allowing us to observe the system's response to both pod failure and network latency. Finally, the post-validation phase is essential to verify that the system returns to its steady states after the faults are removed. We should allocate 15 seconds for post-validation, similar to the pre-validation phase, to ensure that the system stabilizes and meets the defined thresholds for steady states. This allocation results in a total experiment time of 60 seconds, which fits within the 1-minute constraint.
- Total experiment phase: 60s
- Pre-validation phase: 15s
- Fault-injection phase: 30s
- Post-validation phase: 15s

The details of the three phases are as follows:
Pre-validation Phase (15s):
In the pre-validation phase, we need to ensure that the system is in its expected steady state before we proceed with fault injection. Given the constraints of a 15-second total time for this phase, we will conduct two unit tests to verify the steady states: one for the pod's running status and another for the service's availability. These tests will be executed sequentially due to the short duration available, ensuring that each steady state is verified independently and thoroughly. The first test will check if the 'example-pod' is running at least 90% of the time over a 5-second period. This is crucial because the pod's restart policy is set to 'Never', and we need to confirm its availability before introducing any faults. The second test will verify the 'example-service' availability, ensuring it responds with a 200 status code at least 99.9% of the time over another 5-second period. This test is essential to confirm that the service is operational and responsive, given the single pod deployment. By staggering these tests, we can focus on each steady state individually, allowing us to identify any issues before proceeding to the fault injection phase.

Fault-injection Phase (30s):
In this fault-injection phase, we aim to simulate a cyber attack by injecting two types of faults: PodChaos and NetworkChaos. The total duration for this phase is 30 seconds, so we need to carefully schedule the faults and unit tests to fit within this timeframe.

First, we will inject the PodChaos fault to simulate a pod failure. This fault will be injected at the start of the phase (grace period of 0s) and will last for 10 seconds. This duration is chosen to allow enough time for the system to experience the impact of the pod being killed, given the pod's restart policy is set to 'Never'.

Simultaneously, we will run the unit test for the 'example-pod-running' steady state to verify if the pod is running at least 90% of the time during the fault injection. This test will also start at 0s and run for 10 seconds, aligning with the PodChaos duration.

Next, we will inject the NetworkChaos fault to simulate network latency. This fault will start at 10 seconds (after the PodChaos fault ends) and will last for 20 seconds. This staggered approach allows us to observe the system's behavior under network stress after the pod failure has been simulated.

During the NetworkChaos fault, we will run the unit test for the 'example-service-availability' steady state. This test will start at 10 seconds and run for 20 seconds, matching the NetworkChaos duration. This ensures we are checking the service's availability and response time while the network is under stress.

By staggering the faults and aligning the unit tests with the fault durations, we can effectively observe the system's behavior under each fault condition and verify if the steady states are maintained.

Post-validation Phase (15s):
In the post-validation phase, we need to ensure that the system has returned to its steady states after the fault injection. Given the 15-second time constraint, we will perform quick checks to verify the steady states. The two steady states to verify are: 1) the 'example-pod' is running, and 2) the 'example-service' is available. We will execute these checks sequentially due to the short duration, ensuring each test has enough time to gather meaningful data. The first test will check the pod's running status, followed by the service availability test. This order is logical because the pod must be running for the service to be available. Each test will have a brief grace period to allow the system to stabilize after the fault injection, followed by a short duration to perform the checks.

The summary of the above experiment plan:
The chaos engineering experiment is structured into three phases: pre-validation, fault-injection, and post-validation, all to be completed within a total of 60 seconds.

In the pre-validation phase, which lasts for 15 seconds, two unit tests are conducted sequentially to ensure the system is in a steady state before fault injection. The first test, named 'pre-unittest-example-pod-running', checks the 'example-pod' running status. It starts immediately at the beginning of the phase and runs for 5 seconds. Following this, the second test, 'pre-unittest-example-service-availability', begins at the 5-second mark and also runs for 5 seconds, verifying the service's availability.

The fault-injection phase spans 30 seconds and involves two types of faults: PodChaos and NetworkChaos. Initially, the PodChaos fault, named 'fault-podchaos', is injected at the start of the phase and lasts for 10 seconds. Concurrently, the 'fault-unittest-example-pod-running' unit test runs for the same duration to verify the pod's status during the fault. After the PodChaos fault concludes, the NetworkChaos fault, named 'fault-networkchaos', begins at the 10-second mark and continues for 20 seconds. Simultaneously, the 'fault-unittest-example-service-availability' test runs for 20 seconds, starting at the same time as the NetworkChaos fault, to check the service's availability under network stress.

Finally, the post-validation phase, also 15 seconds long, ensures the system returns to its steady states. The 'post-unittest-example-pod-running' test starts after a 2-second grace period and runs for 6 seconds to verify the pod's status. Subsequently, the 'post-unittest-example-service-availability' test begins at the 8-second mark and runs for 5 seconds, checking the service's availability. This sequential execution allows for a brief stabilization period before each test.

To automatically conduct the above experiment plan with Chaos Mesh, the following Chaos-Mesh-Worfklow file was created (by applying it to the cluster, the experiment plan will be automatically executed according to the Chaos-Mesh-Worfklow file):
```yaml

apiVersion: chaos-mesh.org/v1alpha1
kind: Workflow
metadata:
  name: chaos-experiment
spec:
  entry: the-entry
  templates:
```

```
#-------------------------------
# entry point of whole workflow
#-------------------------------
- name: the-entry
  templateType: Serial
  deadline: 30m51s
  children:
    - pre-validation-phase
    - fault-injection-phase
    - post-validation-phase

#-------------------------------------
# Entry point of pre-validation-phase
#-------------------------------------
- name: pre-validation-phase
  templateType: Serial
  deadline: 10m10s
  children:
    - pre-validation-overlapped-workflows

- name: pre-validation-suspend-workflow
  templateType: Serial
  deadline: 5m10s
  children:
    - pre-validation-suspend
    - pre-unittest-example-service-
      availability

- name: pre-validation-suspend
  templateType: Suspend
  deadline: 5s

- name: pre-validation-overlapped-workflows
  templateType: Parallel
  deadline: 5m10s
  children:
    - pre-unittest-example-pod-running
    - pre-validation-suspend-workflow

# Definitions of children of
# pre-validation-phase
- name: pre-unittest-example-pod-running
  templateType: Task
  deadline: 5m5s
  task:
    container:
      name: pre-unittest-example-pod-
            running-container
      image: chaos-eater/k8sapi:1.0
      imagePullPolicy: IfNotPresent
      command: ["/bin/bash", "-c"]
      args: ["python
        unittest_example-pod-running.py
        --duration 5"]
      volumeMounts:
        - name: pvc-volume
          mountPath: /chaos-eater
    volumes:
      - name: pvc-volume
        persistentVolumeClaim:
          claimName: pvc

- name: pre-unittest-example-service-
        availability
  templateType: Task
  deadline: 5m5s
  task:
    container:
      name: pre-unittest-example-service-
            availability-container
      image: grafana/k6:latest
      command: [
        "k6", "run", "--duration",
        "5s", "--quiet",
        "unittest_service-availability.js"]
      volumeMounts:
        - name: pvc-volume
          mountPath: /chaos-eater
    volumes:
      - name: pvc-volume
        persistentVolumeClaim:
          claimName: pvc

#-------------------------------------
# Entry point of fault-injection-phase
#-------------------------------------
- name: fault-injection-phase
  templateType: Serial
  deadline: 10m30s
  children:
    - fault-injection-overlapped-workflows

- name: fault-injection-parallel-workflow
  templateType: Parallel
  deadline: 5m10s
  children:
    - fault-unittest-example-pod-running
    - fault-podchaos

- name: fault-injection-suspend-workflow
  templateType: Serial
  deadline: 5m30s
  children:
    - fault-injection-suspend
    - fault-injection-parallel-workflows

- name: fault-injection-suspend
  templateType: Suspend
  deadline: 10s

- name: fault-injection-parallel-workflows
  templateType: Parallel
  deadline: 5m20s
  children:
    - fault-unittest-service-availability
    - fault-networkchaos

- name: fault-injection-overlapped-workflows
  templateType: Parallel
  deadline: 5m30s
  children:
    - fault-injection-parallel-workflow
    - fault-injection-suspend-workflow

# Definitions of children of
# pre-validation-phase
# unit tests
- name: fault-unittest-example-pod-running
  templateType: Task
  deadline: 5m10s
  task:
    container:
      name: fault-unittest-example-pod-
            running-container
      image: chaos-eater/k8sapi:1.0
      imagePullPolicy: IfNotPresent
      command: ["/bin/bash", "-c"]
      args: ["python
        unittest_example-pod-running_mod0.py
        --duration 10"]
      volumeMounts:
        - name: pvc-volume
          mountPath: /chaos-eater
    volumes:
      - name: pvc-volume
        persistentVolumeClaim:
          claimName: pvc

- name: fault-unittest-service-availability
  templateType: Task
  deadline: 5m20s
  task:
    container:
      name: fault-unittest-example-service-
            availability-container
      image: grafana/k6:latest
      command: ["k6", "run", "--duration",
        "20s", "--quiet",
        "unittest_service-availability.js"]
      volumeMounts:
        - name: pvc-volume
          mountPath: /chaos-eater
    volumes:
      - name: pvc-volume
        persistentVolumeClaim:
          claimName: pvc

# fault_injections
- name: fault-podchaos
  templateType: PodChaos
  deadline: 10s
  podChaos:
```

```
    action: pod-kill
    mode: one
    selector:
      labelSelectors:
        app: example
      namespaces:
        - default


- name: fault-networkchaos
  templateType: NetworkChaos
  deadline: 20s
  networkChaos:
    action: delay
    delay:
      correlation: '50'
      jitter: 10ms
      latency: 100ms
    device: eth0
    direction: to
    mode: all
    selector:
      labelSelectors:
        app: example
      namespaces:
        - default
    target:
      mode: all
      selector:
        labelSelectors:
          app: example
        namespaces:
          - default


#--------------------------------------
# Entry point of post-validation phase
#--------------------------------------
- name: post-validation-phase
  templateType: Serial
  deadline: 10m11s
  children:
    - post-validation-overlapped-workflows

- name: post-validation-suspend-workflow
  templateType: Serial
  deadline: 5m8s
  children:
    - post-validation-suspend
    - post-unittest-example-pod-running

- name: post-validation-suspend
  templateType: Suspend
  deadline: 2s

- name: post-validation-suspend-workflow2
  templateType: Serial
  deadline: 5m11s
  children:
    - post-validation-suspend2
    - post-unittest-service-availability

- name: post-validation-suspend2
  templateType: Suspend
  deadline: 6s

- name: post-validation-overlapped-workflows
  templateType: Parallel
  deadline: 5m11s
  children:
    - post-validation-suspend-workflow
    - post-validation-suspend-workflow2

# Definitions of children of
# pre-validation-phase
- name: post-unittest-example-pod-running
  templateType: Task
  deadline: 5m6s
  task:
    container:
      name: post-unittest-example-pod-
            running-container
      image: chaos-eater/k8sapi:1.0
      imagePullPolicy: IfNotPresent
      command: ["/bin/bash", "-c"]
      args: ["python
            unittest_example-pod-running.py
            --duration 6"]
          volumeMounts:
            - name: pvc-volume
              mountPath: /chaos-eater
        volumes:
          - name: pvc-volume
            persistentVolumeClaim:
              claimName: pvc

    - name: post-unittest-service-availability
      templateType: Task
      deadline: 5m5s
      task:
        container:
          name: post-unittest-service-
                availability-container
          image: grafana/k6:latest
          command: ["k6", "run", "--duration",
            "5s", "--quiet",
            "unittest_service-availability.js"]
          volumeMounts:
            - name: pvc-volume
              mountPath: /chaos-eater
        volumes:
          - name: pvc-volume
            persistentVolumeClaim:
              claimName: pvc
```

Step 2.2, 3, 4. Experiment execution, analysis and improvement (reconfiguring the system to satisfy the hypothesis)
Here is the improvement history:
Experiment result (1st try)
Passed unittests:
- pre-unittest-example-pod-running
- pre-unittest-example-service-availability
Failed unittests:
- fault-unittest-example-pod-running

```log
Exception when calling CoreV1Api->
read_namespaced_pod: (404)
Reason: Not Found
HTTP response headers: HTTPHeaderDict({'
Audit-Id': '71a5fb6f-8e8c-4e31-9bc2-
db80df08c498', 'Cache-Control': 'no-cache,
private', 'Content-Type': 'application/json
', 'X-Kubernetes-Pf-Flowschema-Uid': '
c4624bd9-7fc7-42c6-bcb8-4235110a860d', ... '
Content-Length': '190'})
HTTP response body: {"kind":"Status"...-
Control': 'no-cache, private', 'Content-Type
': 'application/json', 'X-Kubernetes-Pf-
Flowschema-Uid': 'c4624bd9-7fc7-42c6-bcb8
-4235110a860d', 'X-Kubernetes-Pf-
Prioritylevel-Uid': '4706085f-6263-43ae-93f5
-b4a61de8b6be', 'Date': 'Sun, 24 Nov 2024
13:25:27 GMT', 'Content-Length': '190'})
HTTP response body: {"kind":"Status","
apiVersion":"v1","metadata":{},"status":"
Failure","message":"pods \"example-pod\" not
 found","reason":"NotFound","details":{"name
":"example-pod","kind":"pods"},"code":404}
```

- fault-unittest-example-service-availability

```log
time="2024-11-24T13:25:29Z" level=warning
msg="Request Failed" error="Get \"http://
example-service.default.svc.cluster.local...
  http_reqs............: 26 1.269887/s
  iteration_duration...: avg=787.43ms min
  =287.51µs med=1.02s max=1.02s p(90)=1.02s
  p(95)=1.02s
  iterations...........: 26 1.269887/s
  vus..................: 1  min=1 max=1
  vus_max..............: 1  min=1 max=1
time="2024-11-24T13:25:49Z" level=error msg
="thresholds on metrics 'http_req_failed'
have been crossed"
```

- post-unittest-example-pod-running

```log
Traceback (most recent call last):
  File "/chaos-eater/sandbox/
  cycle_20241124_132128/hypothesis/
  unittest_example-pod-running_mod0.py",
  line 49, in <module>
Exception when calling CoreV1Api->
read_namespaced_pod: (404)
Reason: Not Found
HTTP response headers: HTTPHeaderDict({'
Audit-Id': '8a587dad-5e7d-44d6-b9a5-
dd6a14dc6125', 'Cache-Control': 'no-cache,
private', 'Content-Type': 'application/json
', 'X-Kubernetes-Pf-Flowschema-Uid': '
c4624bd9-7fc7-42c6-bcb8-4235110a860d', 'X-
Kubernetes-Pf-Priori...-Control': 'no-cache,
 private', 'Content-Type': 'application/json
', 'X-Kubernetes-Pf-Flowschema-Uid': '
c4624bd9-7fc7-42c6-bcb8-4235110a860d', ...,
'Content-Length': '190'})
HTTP response body: {"kind":"Status","
apiVersion":"v1","metadata":{},"status":"
Failure","message":"pods \"example-pod\" not
 found","reason":"NotFound","details":{"name
":"example-pod","kind":"pods"},"code":404}
```

- post-unittest-example-service-availability
```log
time="2024-11-24T13:26:02Z" level=warning
msg="Request Failed" error="Get \"http://
example-service.default.svc.cluster.local
:80\": dial tcp 10.96.152.112:80: connect:
connection refused"
  http_reqs............: 11 2.14294/s
  iteration_duration...: avg=466.61ms min
  =211.73µs med=3.98ms max=1.03s p(90)=1.02s
   p(95)=1.02s
  iterations...........: 11 2.14294/s
  vus..................: 1  min=1 max=1
  vus_max..............: 1  min=1 max=1
time="2024-11-24T13:26:08Z" level=error msg
="thresholds on metrics 'http_req_failed'
have been crossed"
```

Analysis report (1st try)
The chaos engineering experiment results indicate several critical issues in the system's configuration and its ability to handle faults, particularly in the context of the defined fault scenario. Here is a detailed analysis of the failures observed during the experiment:

1. **Pod Restart Policy and Single Pod Deployment**:
   - The 'fault-unittest-example-pod-running' test failed because the Pod was not found after the PodChaos fault was injected. This is directly related to the Pod's restart policy set to 'Never' in the `nginx/pod.yaml` manifest. When the Pod was killed, it did not restart, leading to a 404 error when attempting to read the Pod's status. This confirms the identified issue #0 (Pod Restart Policy) and issue #1 (Single Pod Deployment), where the lack of redundancy and automatic recovery mechanisms resulted in the Pod being unavailable.

2. **Service Availability**:
   - The 'fault-unittest-example-service-availability' test failed due to connection refusals, indicating that the service was unavailable. This failure is a consequence of the Pod being killed and not restarting, as the Service relies on the Pod to handle requests. The Service's selector in `nginx/service.yaml` targets Pods with the label 'app: example', and with the Pod being down, there were no endpoints available to serve the requests, leading to the connection refused errors.

3. **Post-Validation Failures**:
   - Both 'post-unittest-example-pod-running' and 'post-unittest-example-service-availability' tests failed, showing that the system did not recover to its steady state after the fault injection. The Pod remained unavailable, and consequently, the Service could not respond to HTTP requests. This further highlights the critical impact of the Pod's restart policy and the lack of redundancy.

Insights and Recommendations:
- **Pod Restart Policy**: Change the Pod's restart policy from 'Never' to 'Always' or 'OnFailure' to ensure that the Pod automatically restarts if it fails. This will help maintain the Pod's availability and reduce downtime.
- **Redundancy and Scalability**: Implement a Deployment or ReplicaSet to manage the Pod. This will provide redundancy by running multiple replicas of the Pod, ensuring that the Service remains available even if one Pod fails.
- **Service Resilience**: Consider using a LoadBalancer or Ingress to manage traffic to multiple Pods, enhancing the system's ability to handle failures and maintain service availability.

By addressing these issues, the system can improve its resilience against similar fault scenarios in the future, ensuring that the steady states are maintained even when faults are injected.

Improvement result (1st try)
1 K8s manifests are modified:
- The K8s manifest 'nginx/pod.yaml' was replaced.
Replace the Pod with a Deployment to manage the Pod's lifecycle and provide redundancy. This will ensure that multiple replicas of the Pod are running and can automatically restart if they fail.
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-deployment
  labels:
    app: example
spec:
  replicas: 3
  selector:
    matchLabels:
      app: example
  template:
    metadata:
      labels:
        app: example
    spec:
      containers:
      - name: example-container
        image: nginx:1.17.1
        ports:
        - containerPort: 80
```

Experiment result (2nd try)
Passed unittests:
- pre-unittest-example-pod-running
- pre-unittest-example-service-availability
- fault-unittest-example-pod-running
- fault-unittest-example-service-availability
- post-unittest-example-pod-running
- post-unittest-example-service-availability
Failed unittests:

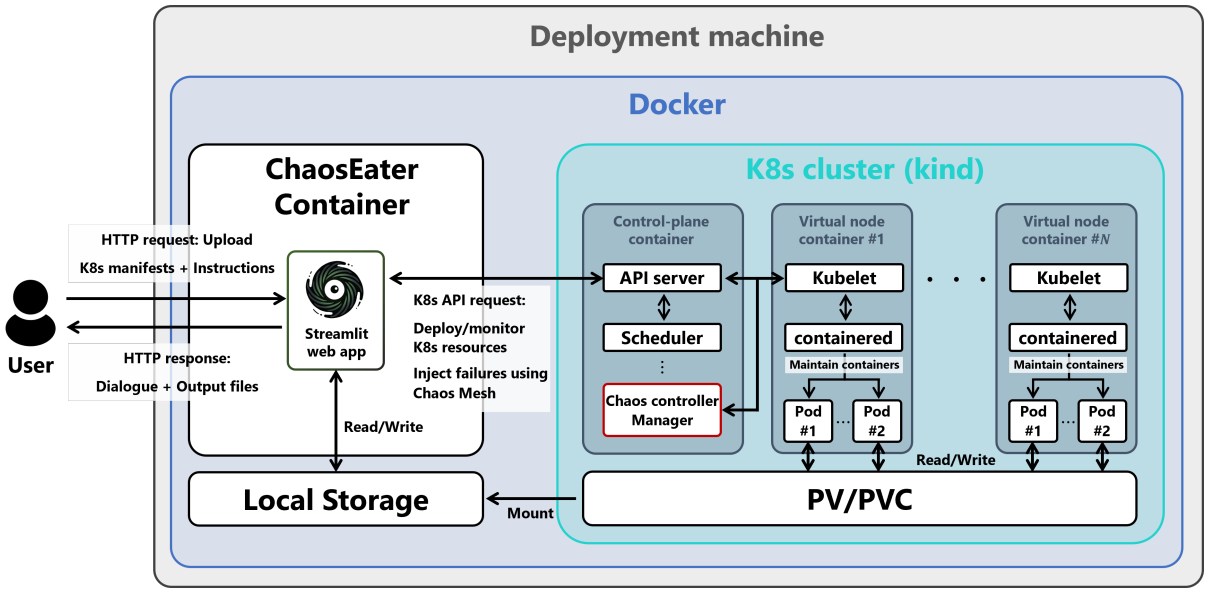


Figure 4: The deployment environment of CHAOSEATER.

D Implementation Details

D.1 System Deployment

Figure 4 illustrates the deployment environment of CHAOSEATER. CHAOSEATER currently supports only development environments, where K8s clusters are constructed on a single machine using kind (Kubernetes, 2018). The web application of CHAOSEATER is implemented with Streamlit (Snowflake, 2020) and is deployed in a Docker container. Users can interact with it via the Graphical User Interface (GUI). After receiving the user inputs, CHAOSEATER autonomously completes a CE cycle for them while calling LLM API. CHAOSEATER interacts with the K8s (kind) clusters via K8s API, requesting resource deployment, monitoring, and failure injection. See `https://github.com/ntt-dkiku/chaos-eater/blob/main/README.md#-quick-start` for specific instructions on setting up this environment.

D.2 Graphical User Interface

Figure 5 shows the GUI of CHAOSEATER. The GUI resembles a typical chatbot interface, with a sidebar that allows for detailed parameter settings. At a minimum, all you need to do is upload the K8s system files via the file uploader. Optionally, you can enter Chaos Engineering instructions in the chat box and control some parameters. The details of the GUI controls are as follows.

(a) LLM setting You may change the LLMs used by CHAOSEATER from the `model` drop-down button. The currently supported LLMs are GPT-4o (gpt-4o-2024-08-06 and gpt-4o-2024-05-13), Claude (claude-3-5-sonnet-20240620), Gemini (google/gemini-1.5-pro).

(b) Cluster setting Currently available clusters are listed in the `Cluster selection` drop-down button. When there are multiple kind clusters, you may change the working kind cluster from here. While the GUI browser is open, the selected cluster will be occupied, and other users will not see the same cluster in the dropdown button. If you check `Clean the cluster before/after run`, all resources in the selected cluster, excect for CHAOSEATER's, will be removed before/after running every single CE cycle. If you check `New deployment`, the input K8s system will be deployed in the preprocessing phase. If it is already deployed, you may uncheck it to skip the deployment.

(c) Parameter setting You can control the parameters of the LLM agents for CHAOSEATER. `Seed for LLMs` sets the random seed for the LLMs (this is only effective when using OpenAI models that support seed setting, such as GPT-4o). `Temperature for LLMs` sets the temperature of the LLMs. `Max. number of steady states` sets the maximum number of steady states proposed during the hypothesis phase. `Max retries` sets the maximum number of it-

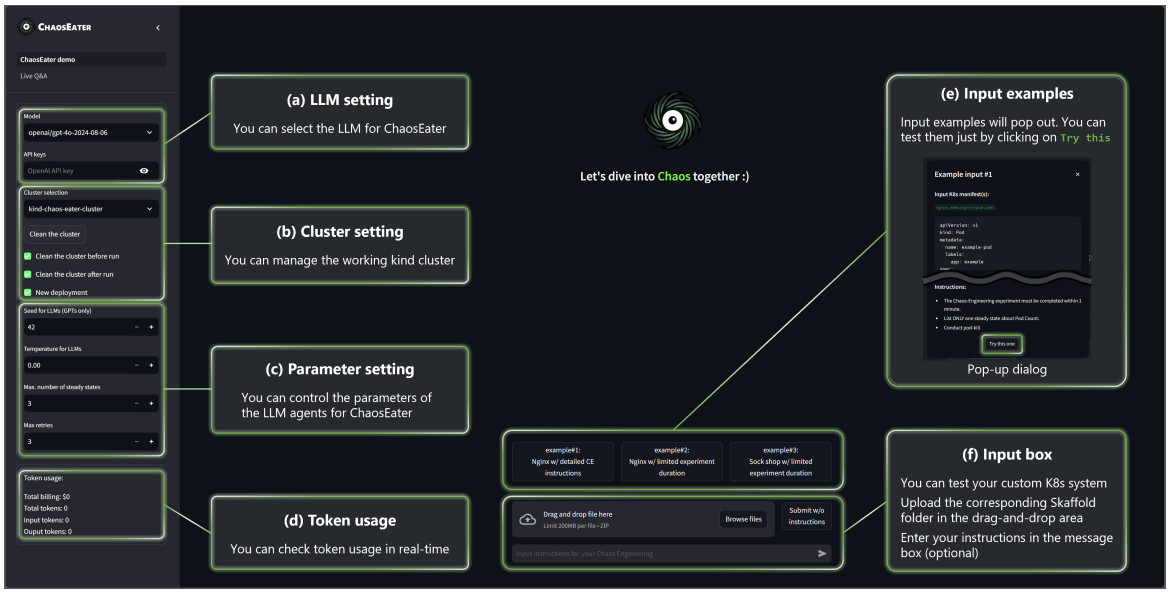


Figure 5: The GUI of ChaosEater

erations for the verification loop and improvement loop. If the loop exceeds this limit, an assertion error will occur, immediately terminating the app at that point.

(d) Token usage You can monitor token usage in real-time. The total cost is calculated based on the official pricing tables as of September 2024.

(e) Input examples We prepare three types of input examples. When you press each button, the content of the K8s manifests to be input and the instructions will be displayed in a dialog. Click the `Try this one` button for the example you wanna try, and a CE cycle will start for that input example.

(f) Input box You can try your custom system by inputting its data to the input box. First, input a zipped folder to the file uploader box following the input format instruction below (this step is mandatory). If you don't have any instructions for the CE cycle, click the `Submit w/o instructions` button, and a CE cycle will start for that input system. If you do, write your instructions in the chat box and click the send icon ▶ or `Enter`. Then, a CE cycle that follows the instructions will start for that input system.

Input format As input, ChaosEater currently supports only a zipped Skaffold project folder, which involves of a Skaffold configuration file and K8s manifests. The Skaffold configuration file must be placed in the root directory of the folder. The K8s manifests can be placed anywhere, but ensure that their relative paths are correctly specified in the manifests section of the Skaffold configuration file.

D.3 Supported Failures

ChaosEater supports most of the Chaos Mesh (Chaos Mesh, 2021) failure types except for `KernelChaos`. Table 3 shows the supported failure types and their short descriptions. In the following, we provide some remarks on the supported failure types.

`KernelChaos` is one of the strongest failures and can affect other `Pods` that share the same kernel as the target `Pod`. We believe that various failure scenarios can be sufficiently simulated with only other types of failures. Therefore, we have currently put the somewhat too strong `KernelChaos` on hold. We may consider adding `KernelChaos` in the future.

`PodChaos`, `HTTPChaos`, `StressChaos`, and `IOChaos` include the `duration` parameter, which specifies the duration of the failure injection. However, Chaos Mesh workflow manifests do not currently support the `duration` parameter. Therefore, we remove the `duration` parameter from the JSON output instructions when detailing the failure parameters in the failure definition of the *hypothesis* phase. Alternatively, ChaosEater specifies the duration of the failure injection using

[1] https://chaos-mesh.org/docs/basic-features/

Table 3: Chaos Mesh failure types supported by CHAOSEATER. The descriptions are quoted from the official document.[1]

Failure type	Description
`PodChaos`	It simulates Pod failures, such as Pod node restart, Pod's persistent unavailability, and certain container failures in a specific Pod.
`NetworkChaos`	It simulates network failures, such as network latency, packet loss, packet disorder, and network partitions.
`DNSChaos`	It simulates DNS failures, such as the parsing failure of DNS domain name and the wrong IP address returned.
`HTTPChaos`	It simulates HTTP communication failures, such as HTTP communication latency.
`StressChaos`	It simulates CPU race or memory race.
`IOChaos`	It simulates the I/O failure of an application file, such as I/O delays, read and write failures.
`TimeChaos`	It simulates the time jump exception.

the `deadline` parameter in the experiment planning of the *experiment* phase. This replacement is officially recommended.[2]

The original `PodChaos` supports three subtypes: pod-kill, pod-failure, and container-kill. On the other hand, CHAOSEATER supports only pod-kill and container-kill, but not pod-failure. Regarding the pod-failure, the official document says "*Pod Failure Chaos Experiment would change the image of each container in the target Pod to the 'pause image', which is a special image that does not perform any operations. if the container is configured without command, livenessProbe and readinessProbe, the container would be inspected as Running and Ready, although it had been changed to the 'pause image', and actually does not provide functionalities as normal or not-available.*"[3] Therefore, the state changes of a `Pod` caused by pod-failure depend on the configuration of livenessProbe and readinessProbe. Depending on their interval settings, it is possible for the `Pod` to be recognized as being in the Running state even while pod-failure is being injected. Considering this issue and some reported bugs[4][5], we have currently put pod-failure on hold.

[2] https://chaos-mesh.org/docs/create-chaos-mesh-workflow/#template-field-description

[3] https://chaos-mesh.org/docs/simulate-pod-chaos-on-kubernetes/

[4] https://github.com/chaos-mesh/chaos-mesh/issues/446

[5] https://github.com/chaos-mesh/chaos-mesh/issues/2523

D.4 Agentic Workflow

D.4.1 Overall Design

To ensure that the LLM agents perform as intended, CHAOSEATER fixes the general workflow according to the systematic CE cycle. It then guides LLM agents by assigning them subdivided CE operations within this workflow. CHAOSEATER prepares prompt templates for each agent,[6] which include placeholders where text is dynamically embedded. Therefore, once the user inputs the data, prompts for each agent are dynamically generated internally by embedding the input data and its intermediate outputs into the templates, and the agents that receive these prompts autonomously complete the workflow (i.e., CE cycle). This is how full automation is achieved. To facilitate data processing within CHAOSEATER, all agents output JSON data. This is achieved by instructing agents in their input prompts to output text in JSON format, and then parsing the output text as JSON data. CHAOSEATER uses the JSON output instruction and parser of LangChain (LangChain, 2023).

Figure 6 shows the agentic workflow of CHAOSEATER. It has 20 agents, and they are chained in series according to the systematic CE cycle, with rule-based algorithms and verification loops at several points. See Appendix D.5 for the system prompt templates for each agent. The ad-

[6] Instead of simply appending previous data and agent outputs to the conversation history to create the next agent's prompt, we create a new conversation for each agent every time and embed the organized previous data and agents' outputs within it. However, the verification loop, which will be discussed later, is an exception.

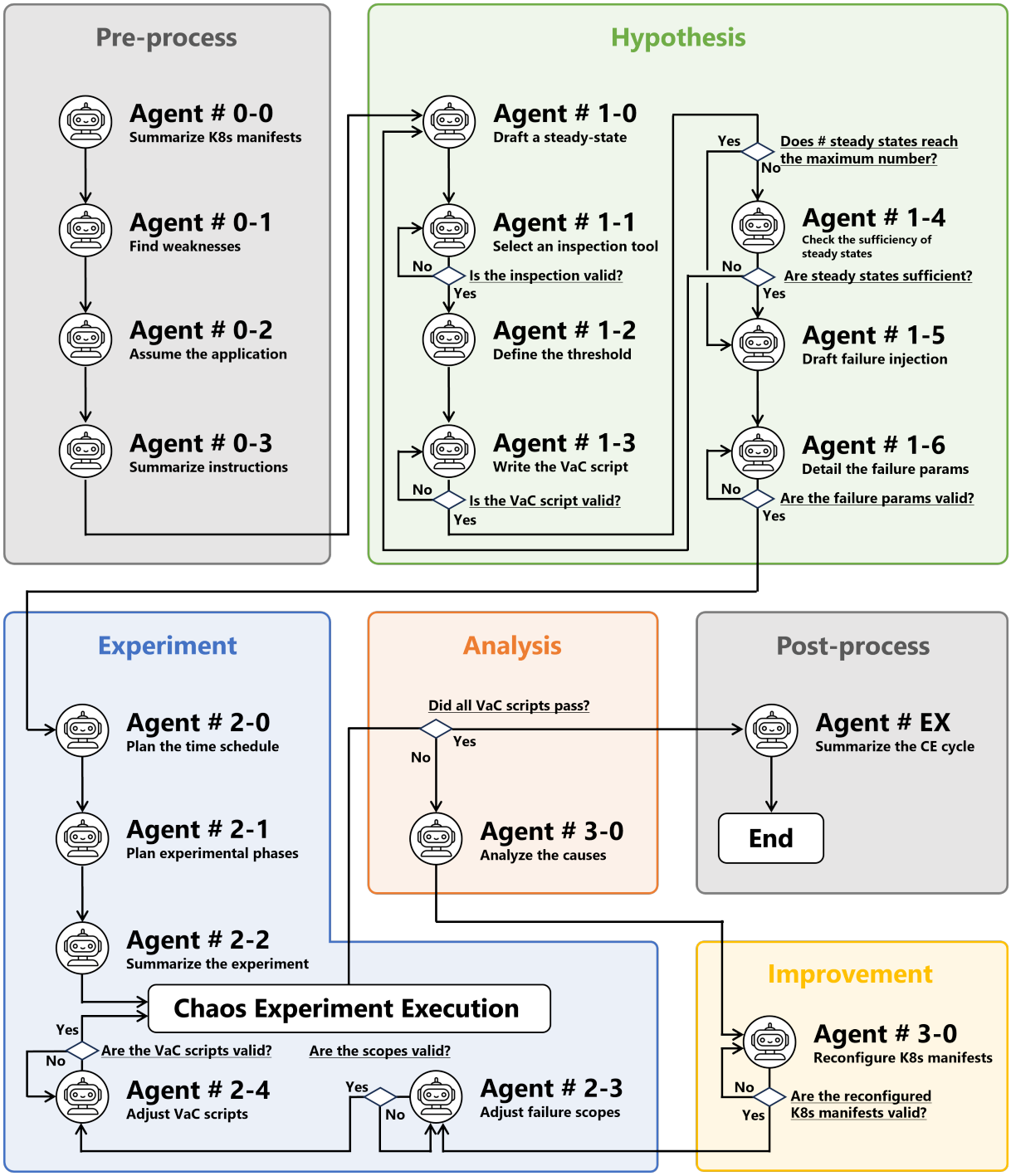


Figure 6: The agentic workflow of CHAOSEATER

vantages of dividing operations into smaller tasks and organizing the agentic workflow as shown in Figure 6 are as follows:

1. **Task performance improvement**; It is generally known that dividing complex tasks into smaller sub-tasks enhances the performance of LLMs in solving them[7] (Khot et al., 2023). In our case, this is also important for managing context length and ensuring the accuracy of JSON output. By dividing tasks, the required input context for each agent can be minimized, mitigating issues such as information loss in long contexts. Furthermore, the JSON output structure for each agent can be minimized to the necessary complexity, reducing improperly formatted JSON out-

[7] https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering/chain-prompts

puts.

2. **Flexibility and extensibility through agent modularization**; By modularizing agents for each divided task, it becomes easier to make partial system modifications, such as replacing specific agents. This modularization also makes it easier for team members to modify agents collaboratively during development.
3. **Towards an interactivity system**; The current CHAOSEATER is a fully automated system with no user interaction during the CE cycle. However, we plan to add interactive functionalities in the future. When refining outputs based on user feedback, the modularization allows querying only the minimum necessary agents, thereby improving the quality of re-generated outputs, similar to the first advantage.

Considering the above advantages, we empirically and manually optimized the workflow of agents and their system prompt templates.

On the other hand, agent modularization has some disadvantages, such as increased prompt management costs due to the greater number of prompts and the challenge of maintaining control over all agents as a unified system. Regarding the latter, in the case study of SOCKSHOP, different behaviors were observed despite the fixed seed and a temperature setting of 0. This implies the complex behavior of multi-LLM agents and the difficulty of their control. We believe that automatic prompt tuning (Yang et al., 2024a; Sun et al., 2023; Pryzant et al., 2023; Kwon et al., 2024; Hsieh et al., 2024) and automatic workflow optimization (Li et al., 2024; Zhuge et al., 2024; Zhou et al., 2024; Hu et al., 2024; Zhang et al., 2024) are promising for overcoming these challenges.

In the following, we provide a detailed explanation of the workflow design for each phase, which could not be fully covered on the main page.

D.4.2 Phase 0: Pre-processing

Given the user input, CHAOSEATER first deploys the user's system to the K8s cluster by running the Skaffold configuration file. Then, each agent sequentially processes the user inputs as follows:

1. Summarize each of the input K8s manifests separately.
2. Identify potential issues for resiliency and redundancy in the K8s manifests.
3. Assume a possible application of the K8s manifests.
4. Summarize user instructions for the CE cycle if provided. At the same time, filter out suspicious prompts, e.g., jailbreak prompts.

This phase is for deploying the user's system and explicitly filling in the implicit context of the user's input. In the subsequent phases, this added context will also be provided as input.

D.4.3 Phase 1: Hypothesis

Steady-state definition Given the pre-processed user inputs, each agent defines steady states as follows:

1. Select a measurable states critical to maintaining the system's application. If any weak configurations are identified from the K8s manifests, their related states are preferentially selected.
2. Select a tool to inspect the state. K8s API and k6 (Grafana Labs, 2021) are supported as the tool. Then, write the corresponding inspection script and inspect the current (normal) value of the state in the system by running the script.
3. Define the threshold for the state based on the inspected value. Note that, according to the definition of a steady state, the threshold must be satisfied under the current conditions.
4. Write a unit-test script that validates whether the steady state (i.e., the pair of the state and its threshold) is satisfied by adding threshold-based assertions to the corresponding inspection script.
5. Check whether the currently defined steady states are sufficient. If they are, the steady-state definition is complete here. Otherwise, return to the first step and define additional steady states.

The unit-test scripts are used in the *experiment* phase to mechanically validate the steady states during chaos experiments. As mentioned in the main page, we here call this unit-test-based validation approach *Validation as Code* (VaC). Naïve approaches, such as validating steady states using LLMs that take log data directly, do not guarantee consistency in the validation process and may even lead to incorrect judgments. On the other hand, with VaC, the validation process becomes fixed once a unit test is written, guaranteeing its consistency. Furthermore, the explicit definition of the process in code enhances its transparency. Figure 7 shows examples of VaC scripts for K8s

(a) VaC script for K8s API (Python)

```python
def check_podcount(label, expected_count, duration):
  consistent_count = True
  for i in range(duration):
    pods = self.v1.list_namespaced_pod(
      namespace='default',
      label_selector=label)
    pod_count = len(pods.items)
    print(f"current pod count: {pod_count}")
    consistent_count = pod_count == expected_count
    if not consistent_count:
      break
    time.sleep(1)
  assert consistent_count, "Pod count was inconsistent."
...
```

(b) VaC script for k6 (Javascript)

```javascript
export const options = {
  vus: 10,
  duration: '10s',
  thresholds: {
    http_req_duration: ['p(95)<500'],
  },
};

export default function () {
  const res = http.get('http://example.com');
  check(res, {'status was 200': (r) =>
    r.status == 200 });
  sleep(1);
}
```

Figure 7: Examples of unit-test scripts to validate steady states.

API (Python) and k6 (Javascript). k6 can collect communication metrics (e.g., response times, error rates, etc) while conducting load tests. In VaC, k6 is used to inspect the communication metrics, while K8s API is used to inspect the other states of K8s resources. Both scripts allow for adjusting test durations through command-line arguments. For k6, the script also sets an appropriate number of virtual users for the load tests.

In steps 2 and 4, scripts are repeatedly debugged until they terminate successfully. In this verification loop, as an exception, CHAOSEATER simply appends the previous agent's output and the resulting error messages to the initial conversation as conversation history, and uses it as the agent's prompt in the next loop. The verification loops discussed later also follow the same process.

Failure definition Given the pre-processed user inputs and the steady states, each agent defines failures that may occur in the system as follows:

1. Assume a failure scenario (e.g., a surge in access due to a promotional campaign, cyber attack, etc.) that may occur in the system. Then, define the sequence of failures that simulates the scenario and may affect the defined steady states. The failures are selected from the failure types supported in Chaos Mesh.
2. Define detailed parameters for each failure, such as the scope of the failure injection, the failure sub-type, the failure strength, etc.

In step 1, the agent outputs a 2D list of Chaos Mesh failure type names, arranged in the order of insertion. The inner lists involve concurrent failures, and the outer list represents the injection order of each concurrent failure set. For example, [[`StressChaos`, `NetworkChaos`], [`PodChaos`]] represents that `PodChaos` is injected after simultaneously injecting `StressChaos` and `NetworkChaos`.

PodChaos parameters

```yaml
action: pod-kill
  mode: one
  selector:
    labelSelectors:
      app: example
    namespaces:
      - default
```

Figure 8: An example of detailed parameters.

In step 2, the agent separately defines the detailed parameters of each failure. Each failure type requires a different parameter set. Therefore, given a failure type name, CHAOSEATER dynamically selects the corresponding JSON output instruction. Instructions for each of the seven supported failures are prepared in advance based on the official Chaos Mesh documentation. The agent then outputs the corresponding parameter set. Figure 8 shows an example of the parameter set of `PodChaos`. See also Dynamic instruction 1–7 for detailed parameter instruction for each failure type. The parameter sets output by the agent are verified through a verification loop, which repeatedly debugs them until their Chaos Mesh manifests pass the `kubectl apply -dry-run=server` command. The failure injection duration and more detailed injection timing are defined in the next chaos experiment planning (see next section), along with the duration and timing for running the VaC scripts.

D.4.4 Phase 2: (Chaos) Experiment

Experiment planning Given the pre-processed user inputs and the hypothesis, each agent plans a chaos experiment that validates the hypothesis by dividing it into the three stages (i.e., pre-validation, failure-injection, and post-validation stages) as follows:

1. Determine the duration of each stage.
2. Determine the VaC scripts and failure injec-

Task node (K8s API)

```
1 - name: {{ node_name }}
2   templateType: Task
3   deadline: {{ duration }}
4   task:
5     container:
6       name: {{ node_name }}-container
7       image: chaos-eater/k8sapi:1.0
8       imagePullPolicy: IfNotPresent
9       command: ["/bin/bash", "-c"]
10      args: [
11        "python /chaos-eater/{{ path_to_vac_script }}
12        --duration {{ duration }}"
13      ]
14 ...
```

Task node (k6)

```
1 - name: {{ node_name }}
2   templateType: Task
3   deadline: {{ duration }}
4   task:
5     container:
6       name: {{ node_name }}-container
7       image: grafana/k6:latest
8       command: [
9         "k6", "run", "--duration", "{{ duration }}",
10        "--quiet", "/chaos-eater/{{ path_to_vac_script }}"
11      ]
12 ...
```

Failure node

```
1 - name: {{ node_name }}
2   templateType: {{ name }}
3   deadline: {{ duration }}
4   {{ name }}:
5     {{ failure_params }}
```

Suspend node

```
1 - name: {{ node_name }}
2   templateType: Suspend
3   deadline: {{ duration }}
```

Serial group node

```
1 - name: {{ node_name }}
2   templateType: Serial
3   deadline: {{ duration }}
4   children:
5     {{ list_of_grouped_node_names }}
6 {{ code_snippets_of_grouped_nodes }}
```

Parallel group node

```
1 - name: {{ node_name }}
2   templateType: Parallel
3   deadline: {{ duration }}
4   children:
5     {{ list_of_grouped_node_names }}
6 {{ code_snippets_of_grouped_nodes }}
```

Figure 9: YAML code snippets for each node type. The blue double curly braces {{}} represent placeholders. `node_name` is a node identifier automatically generated from `name`. `code_snippets_of_grouped_nodes` is filled with the complete code snippets of nodes that are grouped either serially or in parallel. The other placeholders are filled with values from the items of the schedule list.

tions to be executed in each stage. For each of them, specify the duration and grace period within a range that does not exceed the duration of the stage.

3. Summarize the timeline of the chaos experiment in detail. This summary is referred to when analyzing the experiment results.

In step 2, the agent outputs a list of dictionaries (i.e., schedule list) separately for each stage, with each dictionary containing three keys: `name`, `grace_period`, and `duration`. The `name` is either a steady state name or a failure type name, and each corresponds one-to-one with the VaC script or failure injection defined in the hypothesis. The `grace_period` is the waiting time from the start of each stage until the execution of the VaC script or failure injection, allowing flexible adjustment of the execution timing. The `duration` is the execution period after the grace period.

After appending the corresponding failure and VaC parameters to each item in the schedule lists, CHAOSEATER converts these lists into a Chaos Mesh workflow manifest. This manifest enables the scheduling of VaC script execution and failure injection by constructing a workflow composed of the following three types of nodes: `failure` node for executing failure injection, `task` node for executing VaC scripts, and `suspend` node for waiting a specified duration. CHAOSEATER hierarchically groups these nodes to construct complex workflows. For convenience, we define each node as the corresponding code snippet shown in Figure 9. Therefore, the functions that convert items of the schedule list into nodes, as well as those that group multiple nodes, correspond to operations that fill the placeholder of these code snippets with the argument data.

Algorithm 1 shows the hierarchical node grouping algorithm. The input is a directory that organizes the schedule lists for each stage. The algorithm first iterates over the three stages. For each state, each item in the stage's schedule list is converted to the corresponding type of node (line 5–8). Next, each node that has a grace period greater than zero is serially grouped with a `suspend` node. This `suspend` node is placed before the grouped node and waits for the duration of the grace period of that node (line 10–12). Then, all the serially grouped nodes and remaining nodes are grouped in parallel (line 15). Finally, the nodes grouped in parallel in each stage are grouped serially (line 17). The output is the hierarchical group node, and by adding a header and other necessary elements to its code snippet, a Chaos Mesh workflow manifest can be generated. Figure 10 shows an example of hierarchically grouped nodes in this algorithm.

Algorithm 1: Hierarchical node grouping algorithm

```
Input  : A schedule list output by the agent schedule_list (dict of list of dict);
         e.g., schedule_list = {
                 "pre-valid" : [{"name" : "pod-running", "duration" : "20s", ...}, ...],
                 "failure-injection" : [{"name" : "PodChaos", "duration" : "20s", ...}, ...],
                 "post-valid" : [{"name" : "pod-running", "duration" : "20s", ...}, ...]
               }
Output : A Chaos Mesh workflow node consisting of hierarchically grouped nodes hierarchical_node
all_group ← []
for stage ∈ ["pre-valid", "failure-injection", "post-valid"] do
    parallel_group ← []
    for item ∈ schedule_list.stage do
        if IsFailureInjection(item) then
            node ← CreateFailureNode(item.name, item.duration, item.failure_params)
        else
            node ← CreateTaskNode(item.name, item.duration, item.vac_params)
        if item.grace_period > 0 then
            suspend ← CreateSuspendNode(item.grace_period);
            serial_group_node ← CreateSerialGroupNode([suspend, node]);
            parallel_group.append(serial_group_node);
        else
            parallel_group.append(node);
    parallel_group_node ← CreateParallelGroupNode()
    all_group.append(parallel_group_node)
hierarchical_node ← CreateSerialGroupNode(all_group)
return hierarchical_node
```

Experiment replanning (within the improvement loop) Resource types and metadata defined in the K8s manifests may be changed during the *improvement* phase. Therefore, replanning inspection targets in VaC scripts and the scope of failure injections is required between the *improvement* phase and the next experiment execution. Given the original and reconfigured K8s manifests, as well as the previous VaC scripts, an agent adjusts or retains the inspection-target specifications in the VaC scripts. The inspection targets refer to resource specifications (line 11 and 12 in (a)), the request DNS (line 13 in (b)), etc., in Figure 7. Given the original and reconfigured K8s manifests, as well as the previous failure-injection scope, another agent adjusts or retains the scope for the reconfigured manifests. The scope refers to the `selector` filed in Figure 8. These adjustments are also debugged through a verification loop. After the adjustments, CHAOSEATER regenerates a new ChaosMesh workflow manifest by replacing only the path of VaC scripts of `task` nodes and the `selector` field of `failure` nodes with adjusted ones. Note that this replanning only makes minor adjustments to reflect the changes in the K8s manifests, without altering the chaos experiment's original intent.

D.4.5 Phase 4: Improvement

Given the K8s manifests, the hypothesis, the experiment plan, and the improvement loop history, an agent reconfigures the K8s manifests so that all the VaC scripts pass in the chaos experiment. Here, the agent is instructed to make only minimal changes in a single improvement step, aiming for incremental improvement. This approach helps prevent undesired improvements, such as resolving issues through excessive resource allocation. The improvement loop history stores the history of the experiment results, their analysis reports, and their reconfigurations, within the improvement loop. The history suppresses the repetition of the same reconfiguration.

There are three reconfiguration modes: `create`, `delete`, and `replace`. The agent first selects the reconfiguration modes while specifying file names, and then writes the reconfigured K8s manifests only for the `create` and `replace` modes. The file manager of CHAOSEATER then edits the folder from the previous improvement loop (in the first improvement, it corresponds to the user's input folder) according to the agent's output. Figure 11 illustrates this reconfiguration process.

The verification loop is also conducted here: the agent's output is debugged repeatedly until all the

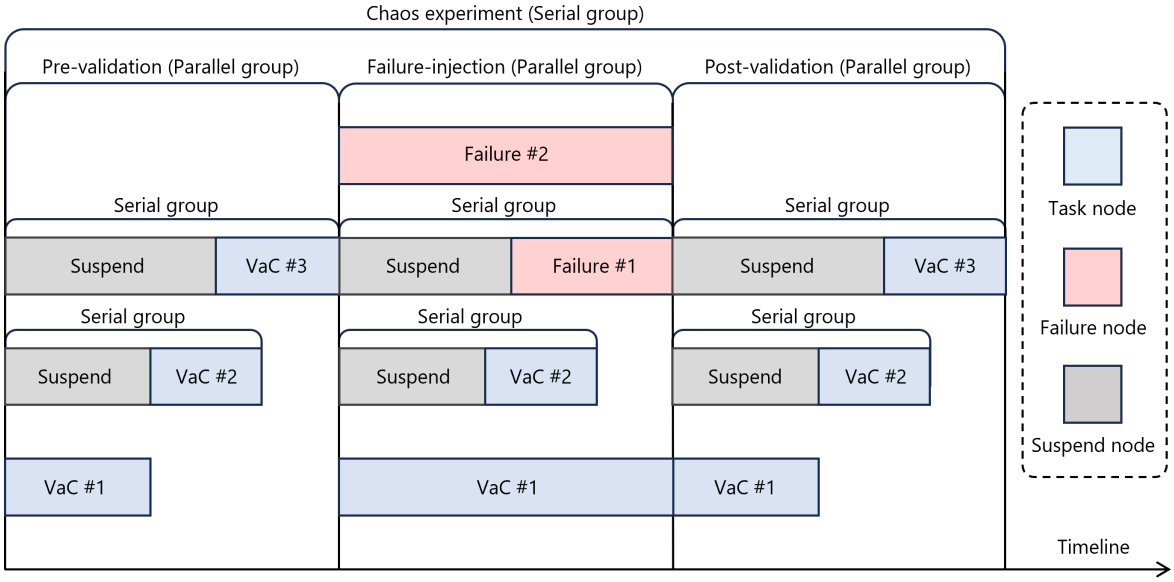


Figure 10: Hierarchical grouping for implementing a complex chaos experiment plan in Chaos Mesh.

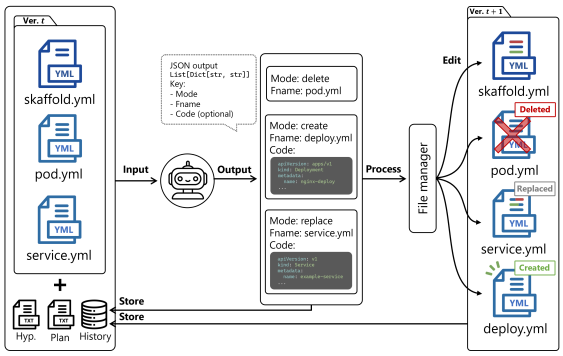


Figure 11: Reconfiguration process by the agent and a file management algorithm.

K8s manifests in the edited folder are correctly applied to the K8s cluster without any errors.

D.5 System prompt templates

In this section, we share all prompt templates for LLM agents of CHAOSEATER. Words enclosed in blue curly braces {} denote placeholders that are replaced with user input or the previous agent's outputs. Each placeholder has a unique variable name, and placeholders with the same variable name across different prompt templates will have the same text embedded. Examples of the text embedded in each placeholder are also provided right after their associated prompt templates. On the other hand, words enclosed in red curly braces {} denote placeholders that are replaced with pre-defined dynamic instructions. These pre-defined instructions are dynamically selected and embedded in the placeholders according to the conditions. These dynamic instructions are also provided right after their associated prompt templates. Words enclosed in green curly braces {} denote placeholders that are replaced with the same text as previously introduced, in order to avoid repetition.

Although it is omitted in each prompt template, a prefill will be inserted at the end of every prompt template. Specifically, the first key in the JSON output will be added as an AI's message, as shown in Figure 12. It eliminates redundant outputs and improves the stability of the JSON outputs.

The term "fault" appears in the prompt templates and the output of CHAOSEATER. this is aligned with the terminology used in Chaos Mesh and has the same meaning as "failure".

The prefill for the Agent #0-0

```
AI:
```json
{\"k8s_summary\":
```

Figure 12: An example of prefills.

D.5.1 Pre-processing

Prompt 2: Agent # 0-0 for summarizing K8s manifests

```
System:
System: You are a professional Kubernetes (
k8s) engineer.
Given a K8s manifest, please summarize it
according to the following rules:
```
```

- The summary must be written in bullet points.
- Summarize the functions of the K8s manifest in a way that is understandable to even beginners.
- The output should be formatted as a JSON instance that conforms to the JSON schema below.

As an example, for the schema {\"properties\": {\"foo\": {\"title\": \"Foo\", \"description\": \"a list of strings\", \"type\": \"array\", \"items\": {\"type\": \"string\"}}}, \"required\": [\"foo\"]}\nthe object {\"foo\": [\"bar\", \"baz\"]} is a well-formatted instance of the schema. The object {\"properties\": {\"foo\": [\"bar\", \"baz\"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "k8s_summary": {
      "title": "K8S Summary",
      "description": "Summary of the K8s manifest. Summarize it in bullet points like '- the 1st line\n- the second line...'",
      "type": "string"
    }
  },
  "required": [
    "k8s_summary"
  ]
}
```

Human:
K8s manifest
{k8s_yaml}

Please summarize the above K8s manifest.

Example 2: **k8s_yaml**

```nginx/pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
  labels:
    app: example
spec:
  restartPolicy: Never
  containers:
  - name: example-container
    image: nginx:1.17.1
    ports:
    - containerPort: 80
```

Prompt 3: Agent # 0-1 for finding potential weaknesses

System:
You are a professional Kubernetes (K8s) engineer.
Given K8s manifests for a system, you will identify their potential issues for resiliency and redundancy when failures occur in the system.
Always keep the following rules:
- List each issue with its name, associated K8s manifest(s), potential issues due to fault injection, and the configuration causing the issues (no need to suggest improvements).
- If the same issue exists in different manifests, merge them into a single issue, specifying all the associated manifest names.
- The output should be formatted as a JSON instance that conforms to the JSON schema below.

As an example, for the schema {\"properties\": {\"foo\": {\"title\": \"Foo\", \"description\": \"a list of strings\", \"type\": \"array\", \"items\": {\"type\": \"string\"}}}, \"required\": [\"foo\"]}\nthe object {\"foo\": [\"bar\", \"baz\"]} is a well-formatted instance of the schema. The object {\"properties\": {\"foo\": [\"bar\", \"baz\"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "issues": {
      "title": "Issues",
      "description": "List issues with its name, potential issues due to fault injection, and manifest configuration causing the issues (no need to suggest improvements).",
      "type": "array",
      "items": {
        "$ref": "#/definitions/K8sIssue"
      }
    }
  },
  "required": [
    "issues"
  ],
  "definitions": {
    "K8sIssue": {
      "title": "K8sIssue",
      "type": "object",
      "properties": {
        "issue_name": {
          "title": "Issue Name",
          "description": "Issue name",
          "type": "string"
        },
        "issue_details": {
          "title": "Issue Details",
          "description": "potential issues due to fault injection",
          "type": "string"
        },
        "manifests": {
          "title": "Manifests",
          "description": "manifest names having the issues",
          "type": "array",
          "items": {
            "type": "string"
          }
        },
        "problematic_config": {
          "title": "Problematic Config",
          "description": "problematic configuration causing the issues (no need to suggest improvements).",
          "type": "string"
        }
      },
      "required": [
        "issue_name",
        "issue_details",
        "manifests",
        "problematic_config"
      ]
    }
  }
}
```

Human:
Here are the K8s manifests for my system.

{k8s_yamls}

Please list issues for each K8s manifest.

Example 3: **k8s_yamls**

```nginx/pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
  labels:
    app: example
spec:
  restartPolicy: Never
  containers:
  - name: example-container
    image: nginx:1.17.1
    ports:
    - containerPort: 80
```

```nginx/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: example-service
spec:
  selector:
    app: example
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

Prompt 4: Agent # 0-2 for assuming an application

System:
You are a professional Kubernetes (k8s) engineer.
Given k8s manifests and dependencies between them, please assume a real-world application (service) of the manifests according to the following rules:
- If the application is explicitly specified in the instructions, assume it.
- You can leverage any given information, including file name, manifests, and dependencies, to guess the purpose of the manifests.
- The output should be formatted as a JSON instance that conforms to the JSON schema below.

As an example, for the schema {\"properties\": {\"foo\": {\"title\": \"Foo\", \"description\": \"a list of strings\", \"type\": \"array\", \"items\": {\"type\": \"string\"}}}, \"required\": [\"foo\"]}\nthe object {\"foo\": [\"bar\", \"baz\"]} is a well-formatted instance of the schema. The object {\"properties\": {\"foo\": [\"bar\", \"baz\"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "thought": {
      "title": "Thought",
      "description": "Before assuming an application, reason logically why you assume it for the given manifests. e.g., from file name, instructions, or other elements?",
      "type": "string"
    },
    "k8s_application": {
      "title": "K8S Application",
      "description": "Specify what the service (application) offers to users.",
      "type": "string"
    }
  },
  "required": [
    "thought",
    "k8s_application"
  ]
}
```

Human:
{user_input}

Please assume a real-world application of the manifests.

Example 4: **user_input**

K8s manifest:
```nginx/pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
  labels:
    app: example
spec:
  restartPolicy: Never
  containers:
  - name: example-container
    image: nginx:1.17.1
    ports:
    - containerPort: 80
```
Summary of nginx/pod.yaml:
- This manifest defines a Kubernetes Pod.
- The Pod is named 'example-pod'.
- It includes metadata with a label 'app: example'.
- The Pod's restart policy is set to 'Never', meaning it won't restart automatically if it fails.
- The Pod contains one container named 'example-container'.
- The container uses the 'nginx:1.17.1' image.
- The container exposes port 80, which is typically used for HTTP traffic.

K8s manifest:
```nginx/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: example-service
spec:
  selector:
    app: example
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```
Summary of nginx/service.yaml:
- This manifest defines a Kubernetes Service.
- The Service is named 'example-service'.
- It uses the 'v1' API version.
- The Service selects pods with the label 'app: example'.
- It exposes the Service on port 80 using the TCP protocol.
- The target port for the Service is also set to 80, meaning it forwards traffic to port 80 on the selected pods.

Prompt 5: Agent # 0-3 for summarizing user instructions

System:
You are a professional Chaos Engineering practitioner.
Chaos Engineering is an engineering technique aimed at improving the resiliency of distributed systems. It involves artificially injecting specific failures into a distributed system and observing its behavior in response. Based on the observation, the system can be proactively improved to handle those failures.
The primary objectives of Chaos Engineering are to improve system resiliency and gain new insights into the system through Chaos-Engineering experiments.\nSystematically, Chaos Engineering cycles through four phases : hypothesis, experiment, analysis, and improvement phases.
1) Hypothesis: Define steady states (i.e., normal behavior) of the system and injected failures (i.e., faults). Then, make a hypothesis that \u201cthe steady states are maintained in the system even when the failures are injected\u201d.
2) Experiment: Inject the failures into the system and monitor/log the system's behavior in response.
3) Analysis: Analyze the logged data and check if the hypothesis is satisfied. If so, one CE cycle is finished here. If not, move to (4)
4) Improvement: Reconfigure the system to satisfy the hypothesis. The reconfigured system is tested again in (2) and (3), i.e., repeat (2) to (4) until the hypothesis is satisfied.

Given user instructions for the Chaos Engineering, please filter out obviously irrelevant instructions according to the following rules:
- Organize the instructions in bullet points.
- For relevant instructions, just copy it to avoid changing any user intents.\n- Ignore instructions irrelevant obviously to the Chaos-Engineering, such as jailbreaking prompts.
- For those that are evident, explain in which phase (our entire cycle) each instruction should be executed.
- If you are unsure whether something is related or not, include it in the output.
- The output should be formatted as a JSON instance that conforms to the JSON schema below.

As an example, for the schema {\"properties\": {\"foo\": {\"title\": \"Foo\", \"description\": \"a list of strings\", \"type\": \"array\", \"items\": {\"type\": \"string\"}}}, \"required\": [\"foo\"]}\nthe object {\"foo\": [\"bar\", \"baz\"]} is a well-formatted instance of the schema. The object {\"properties\": {\"foo\": [\"bar\", \"baz\"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "ce_instructions": {
      "title": "Ce Instructions",
      "description": "Summary of the given instructions for the Chaos Engineering. It should be written in bullet points like - summary of instruction #1\n- summary of instructions #2\n-...",
      "type": "string"
    }
  },
  "required": [
    "ce_instructions"
  ]
}
```

Human:
Instructions
{ce_instructions}

Please filter out the above instructions for the CE.

Example 5: **ce_instructions**

The Chaos-Engineering experiment must be completed within 1 minute.

D.5.2 Hypothesis

Prompt 6: Agent # 1-0 for drafting a steady state

System:
You are a helpful AI assistant for Chaos Engineering.
Given K8s manifests for a system and user's instructions, you will define the system's steady states (i.e., normal behaviors) that are related to potential issues of the system.
Always keep the following rules:
- Define steady states one by one, starting with the steady state related to the K8s resource that is easiest to encounter issues when certain failures occur.
- Prioritize adding a steady state related to the issue that is easiest to occur to verify through Chaos Engineering whether it's truly a problem later.
- An added steady state must be a measurable output, such as the number of pods, throughput, error rates, latency percentiles, etc.
- An added steady state must be specific to a SINGLE K8s resource (i.e., manifest) having potential issues for resiliency and redundancy.
- An added steady state must be different from the already defined ones.
- The output should be formatted as a JSON instance that conforms to the JSON schema below.

As an example, for the schema {\"properties\": {\"foo\": {\"title\": \"Foo\", \"description\": \"a list of strings\", \"type\": \"array\", \"items\": {\"type\": \"string\"}}}, \"required\": [\"foo\"]}\nthe object {\"foo\": [\"bar\", \"baz\"]} is a well-formatted instance of the schema. The object {\"properties\": {\"foo\": [\"bar\", \"baz\"]}} is not well-formatted.

Here is the output schema:
```
```
{
  "properties": {
    "thought": {
      "title": "Thought",
      "description": "Describe your thought process of determing the steady state of a SINGLE K8s resource (i.e., manifest) that is easiest to encounter the issues. Describe also the details of the steady state itself.",

```
      "type": "string"
    },
    "manifest": {
      "title": "Manifest",
      "description": "The targeted K8s-
      manifest name. Specify a SINGLE
      manifest.",
      "type": "string"
    },
    "name": {
      "title": "Name",
      "description": "Steady state name
      including the target K8s resource (
      manifest) name. Please write it using
      a-z, A-Z, and 0-9.",
      "type": "string"
    }
  },
  "required": [
    "thought",
    "manifest",
    "name"
  ]
}
```

Human:
Here is the overview of my system:
{user_input2}

Please follow the instructions below regarding Chaos Engineering:
{ce_instructions}

Steady states already defined are as follows:
{predefined_steady_states}

The plan for defining the next state is as follows:
{prev_check_thought}

Now, define a steady state that are different from the already defined steady states.

Example 6: `user_input2`

The system consists of the following K8s manifest(s):
//**{user_input}**//

The resiliency issues/weaknesses in the system are as follows:
Issue #0: Pod Restart Policy
- details: The Pod will not restart automatically if it fails, which can lead to downtime.
- manifests having the issues: ['nginx/pod.yaml']
- problematic config: restartPolicy: Never

Issue #1: Single Pod Deployment
- details: Using a single Pod without a controller like Deployment or ReplicaSet can lead to lack of redundancy and no automatic recovery if the Pod fails.
- manifests having the issues: ['nginx/pod.yaml']
- problematic config: kind: Pod

The expected type of application on the system (i.e., K8s manifests):
Web server application using Nginx to serve HTTP content.; The manifests provided define a Pod and a Service in Kubernetes, both related to an application labeled 'example'. The Pod runs an Nginx container, which is a popular web server and reverse proxy server. The Service is configured to expose this Pod on port 80, which is the default port for HTTP traffic. Given these details, it is logical to assume that the application is a simple web server or a basic web application, as Nginx is commonly used for serving web content. The file names and the use of Nginx further support this assumption.

Example 7: `predefined_steady_states`

1 steady states are defined.
1st steady states:
- Name: example-pod-running
- Description: The first issue to address is the Pod Restart Policy Issue, as it is directly related to the Pod's ability to recover from failures. Since the Pod is configured with a 'Never' restart policy, it will not restart automatically if it fails, leading to potential downtime. Therefore, the steady state should verify that the Pod is running and not in a failed state. This can be measured by checking the number of running Pods, which should be 1, as there is only one Pod defined in the manifest.
- Threshold for the steady state: The 'example-pod' must be in the 'Running' state at least 54 out of 60 seconds (90% of the time) during a 1-minute monitoring period.; The steady state we are considering is whether the 'example-pod' is running. The current state shows that the pod was running for 5 out of 5 seconds, which is 100% of the time during the monitoring period. Given the constraints of the system, such as the single pod deployment and the 'Never' restart policy, we need to set a threshold that accounts for potential minor fluctuations while still ensuring the pod is mostly available. Since the chaos experiment must be completed within 1 minute, we can set the threshold to require the pod to be running at least 90% of the time during a 1-minute monitoring period. This allows for some brief interruptions while still maintaining a high level of availability.
- Whether the steady state meets the threshold is determined by the following Python script with K8s API:
```
import os
import time
import argparse
from kubernetes import client, config
from unittest_base import K8sAPIBase

class TestPodRunningState(K8sAPIBase):
  def __init__(self):
    super().__init__()
    def check_pod_running_state(self,
    duration):
      pod_name = 'example-pod'
      namespace = 'default'
      running_count = 0
      for _ in range(duration):
        try:
          pod = self.v1.read_namespaced_pod(
          name=pod_name, namespace=namespace
          )
          if pod.status.phase == 'Running':
            running_count += 1
            print(f\"Pod {pod_name} status:
            {pod.status.phase}\")
        except client.exceptions.
        ApiException as e:
          print(f\"Exception when calling
          CoreV1Api->read_namespaced_pod: {e
          }\")
        time.sleep(1)
        # Calculate the percentage of time
```

```
        the pod was running
        running_percentage = (running_count
        / duration) * 100
        print(f\"Pod was running {
        running_count} out of {duration}
        seconds, which is {
        running_percentage:.2f}% of the time
        .\")
        # Assert that the pod was running at
         least 90% of the time
        assert running_percentage >= 90, f\"
        Pod running percentage {
        running_percentage:.2f}% is below
        the threshold of 90%.\"

def main():
  parser = argparse.ArgumentParser(
  description='Test the running state of a
  Kubernetes Pod.')
  parser.add_argument('--duration', type=int
  , default=60, help='Duration to check the
  Pod status in seconds.')
  args = parser.parse_args()
  test = TestPodRunningState()
  test.check_pod_running_state(args.duration
  )

if __name__ == '__main__':
  main()
```

Example 8: **prev_check_thought**

```
The current steady state focuses on ensuring
 that the 'example-pod' is running at least
90% of the time during a 1-minute monitoring
 period. This addresses the Pod Restart
Policy Issue by verifying the pod's
availability despite its 'Never' restart
policy. However, the system also has a
Single Pod Deployment issue, which means
there is no redundancy. If the single pod
fails, the service will be disrupted. The
current steady state does not address this
redundancy issue. To ensure the system's
resilience, an additional steady state
should be defined to verify that the service
 remains available even if the single pod
fails. This could involve checking the
service's response or availability from an
external perspective, ensuring that the
service is accessible and responsive, which
would indirectly address the redundancy
issue by ensuring service continuity.
```

Prompt 7: Agent # 1-1 for defining an inspection strategy

System:

```
You are a helpful AI assistant for Chaos
Engineering.
Given Kubernetes (K8s) manifests for a
network system and its state type, you will
inspect the current value of the state type.
Always keep the following rules:
- You can use either K8s API (Python) or k6
(Javascript) to inspect the state.
- Use the K8s API for checking the current
state of K8s resources
- Use k6 for checking communication statuses
/metrics, such as request sending, response
time, latency, etc.
- If you use K8s API, consider appropriate
test duration. If you use k6, consider not
only appropriate test duration but also an
appropriate number of virtual users in the
load test.
- Pay attention to namespace specification.
If the namespace is specified in the
manifest, it is deployed with the namespace.
 If not, it is deployed with the 'default'
namespace.
- When sending requests to a K8s resources,
use their internal DNS names in the format:
```service-name.namespace.svc.cluster.local:
port```. For the port setting, use the
service port, not the targetPort or nodePort
. Ensure that the port matches the service
port defined in the manifest.
- If other request formats are provided by
the user, follow the user's format.
- The output should be formatted as a JSON
instance that conforms to the JSON schema
below.

As an example, for the schema {\"properties
\": {\"foo\": {\"title\": \"Foo\", \"
description\": \"a list of strings\", \"type
\": \"array\", \"items\": {\"type\": \"
string\"}}}, \"required\": [\"foo\"]}\nthe
object {\"foo\": [\"bar\", \"baz\"]} is a
well-formatted instance of the schema. The
object {\"properties\": {\"foo\": [\"bar\",
\"baz\"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "thought": {
      "title": "Thought",
      "description": "Describe your thoughts
       for the tool usage. e.g., the reason
      why you choose the tool and how to use
      .",
      "type": "string"
    },
    "tool_type": {
      "title": "Tool Type",
      "description": "Tool to inspect the
      steady state. Select from ['k8s', 'k6
      '].",
      "enum": [
        "k8s",
        "k6"
      ],
      "type": "string"
    },
    "tool": {
      "title": "Tool",
      "description": "If tool_tyepe='k8s',
      write here K8sAPI. If tool_tyepe='k6',
       write here K6JS.",
      "anyOf": [
        {
          "$ref": "#/definitions/K8sAPI"
        },
        {
          "$ref": "#/definitions/K6JS"
        }
      ]
    }
  },
  "required": [
    "thought",
    "tool_type",
    "tool"
  ],
  "definitions": {
    "K8sAPI": {
      "title": "K8sAPI",
      "type": "object",
      "properties": {
        "duration": {
          "title": "Duration",
          "description": "Duration of the
          status check every second in a for
           loop. Set appropriate duration to
           check the current state of the
          system. The maximum duration is 5s
          .",
          "type": "string"
        },
```

```
        "script": {
          "title": "Script",
          "description": "Python script with
           K8s client libraries to inspect
          the current status of a K8s
          resource. Write only the content
          of the code, and for dictionary
          values, enclose them within a pair
           of single double quotes (\").
          Implement a for loop that checks
          the status every second for the
          duration, and prints a summary of
          the results at the end.\n- To
          support docker env, please
          configure the client as follows:
          ```\n# Load Kubernetes
          configuration based on the
          environment\n   if os.getenv('
          KUBERNETES_SERVICE_HOST'):\n
                 config.
          load_incluster_config()\n   else
          :\n   config.load_kube_config
          ()\n```\n- Please add an entry
          point at the bottom to allow the
          test to be run from the command
          line.\n- Please add argparse '--
          duration' (type=int) so that users
           can specify the loop duration.",
          "type": "string"
        }
      },
      "required": [
        "duration",
        "script"
      ]
    },
    "K6JS": {
      "title": "K6JS",
      "type": "object",
      "properties": {
        "vus": {
          "title": "Vus",
          "description": "The number of
          virtual users. You can run a load
          test with the number of virtual
          users.",
          "type": "integer"
        },
        "duration": {
          "title": "Duration",
          "description": "Duration of the
          load test. Set appropriate
          duration to check the current
          state of the system. The maximum
          duration is 5s.",
          "type": "string"
        },
        "script": {
          "title": "Script",
          "description": "k6 javascript to
          inspect the current state. Write
          only the content of the code, and
          for dictionary values, enclose
          them within a pair of single
          double quotes (\"). In options in
          the javascript, set the same 'vus'
           and 'duration' options as the
          above. The interval of status
          check must be 1s second(s). Set a
          threshold that triggers an error
          when a request failure is clearly
          occurring.",
          "type": "string"
        }
      },
      "required": [
        "vus",
        "duration",
        "script"
      ]
    }
  }
}
```

**Human:**
# Here is the overview of my system:
**{user_input2}**

# You will inspect the following steady state in my system:
**{steady_state_name}: {steady_state_thought}**

# Please follow the instructions below regarding Chaos Engineering:
**{ce_instructions}**

Please define the way to inspect "**{steady_state_name}**" in the system defined by the above k8s manifest(s).

In the verification loop, the prompts below will be stacked as history

**AI:**
**{output}**

**Human:**
Your current inspection script causes errors when conducted.
The error message is as follows:
**{error_message}**

Please analyze the reason why the errors occur, then fix the errors.
Always keep the following rules:
- NEVER repeat the same fixes that have been made in the past.
- Fix only the parts related to the errors without changing the original content.
- If requests failed, double-check if the service port is correct.
- You can change the tool (k8s -> k6 or k6 -> k8s) if it can keep the original intention.
- **{the same format instructions as in the System role}**

## Example 9: `steady_state_name`

example-pod-running-state

## Example 10: `steady_state_thought`

The first issue to address is the Pod's restart policy set to 'Never' in the 'nginx/pod.yaml' manifest. This is a critical issue because if the Pod fails, it will not restart automatically, leading to potential downtime. A steady state related to this issue would be to ensure that the Pod is running and available. This can be measured by checking the number of running Pods. Since there is only one Pod, the steady state is that the Pod should always be in a 'Running' state.

## Prompt 8: Agent # 1-2 for defining a threshold

**System:**
You are a helpful AI assistant for Chaos Engineering.
Given k8s manifests for a system, its steady
```

state, and the current value of the steady state, you will define the threshold for the steady state.
Always keep the following rules:
- The threshold must include reasonable tolerance that makes the threshold being more easiliy satisfied to account for some fluctuations.
- The current value of the steady state must satisfy the threshold (including tolerance) as the currrent value is the normal state and the threshold represents whether the system remains normal.
- If redundancy already exists in the resource, define at least the minimum required value as the threshold.
- Explicitly specify all values related to the threshold, such as the number of resources that must be satisfied, the percentage of time it must be satisfied within the monitoring period, etc.
- The output should be formatted as a JSON instance that conforms to the JSON schema below.

As an example, for the schema {\"properties\": {\"foo\": {\"title\": \"Foo\", \"description\": \"a list of strings\", \"type\": \"array\", \"items\": {\"type\": \"string\"}}}, \"required\": [\"foo\"]}\nthe object {\"foo\": [\"bar\", \"baz\"]} is a well-formatted instance of the schema. The object {\"properties\": {\"foo\": [\"bar\", \"baz\"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "thought": {
      "title": "Thought",
      "description": "Write your thought process to determine the threshold of the steady state.",
      "type": "string"
    },
    "threshold": {
      "title": "Threshold",
      "description": "the threshold of the steady state, which should be satisfied in the current state.",
      "type": "string"
    }
  },
  "required": [
    "thought",
    "threshold"
  ]
}
```

Human:
Here is the overview of my system:
{user_input2}

You will determine a reasonable threshold for the following steady state of my system:
{steady_state_name}: {steady_state_thought}

{inspection_summary}

Please follow the instructions below regarding Chaos Engineering:
{ce_instructions}

Now, please define a reasonable threshold for the steady state according to the above information.

Example 11: **inspection_summary**

The Python code of k8s client libraries to inspect the current state of the steady state and its result are the following:
Script:
```python
import os
import time
from kubernetes import client, config
def check_pod_status(namespace, pod_name, duration):
    # Load Kubernetes configuration based on the environment
    if os.getenv('KUBERNETES_SERVICE_HOST'):
        config.load_incluster_config()
    else:
        config.load_kube_config()

    v1 = client.CoreV1Api()
    running_count = 0

    for _ in range(duration):
        try:
            pod = v1.read_namespaced_pod(name=pod_name, namespace=namespace)
            if pod.status.phase == 'Running':
                running_count += 1
                print(f\"Pod status: {pod.status.phase}\")
        except client.exceptions.ApiException as e:
            print(f\"Exception when calling CoreV1Api->read_namespaced_pod: {e}\")
            time.sleep(1)
    print(f\"Pod was running {running_count} out of {duration} seconds.\")

if __name__ == '__main__':
    import argparse
    parser = argparse.ArgumentParser(description='Check the running state of a Pod.')
    parser.add_argument('--duration', type=int, default=5, help='Duration to check the Pod status in seconds.')
    args = parser.parse_args()
    check_pod_status(namespace='default', pod_name='example-pod', duration=args.duration)
```

Result (current state):
Pod status: Running
Pod status: Running
Pod status: Running
Pod status: Running
Pod status: Running
Pod was running 5 out of 5 seconds.

Prompt 9: Agent # 1-3-a for writing a VaC script of K8s Python API

System:
You are a helpful AI assistant for writing unit tests in Python.
Given the steady state, python script to inspect it, and its threshold, please write a Python unit test (including for-loop for certain duration) to verify if the steady state satisfies the threshold by adding assertion.
Always keep the following rules:
- Include as many comments as possible in your code so that humans can easily understand what you did later.
- Use the Kubernetes Python API.

- Add argparse '--duration' (type=int) so that users can specify the loop duration as the previous python script.
- NEVER use "unittest" module to use argparse.
- Create a unit test by inheriting from the 'K8sAPIBase' class below (available via ```from unittest_base import K8sAPIBase```):

```python
import os
from kubernetes import client, config

class K8sAPIBase:
    def __init__(self):
        # Load Kubernetes configuration
        based on the environment
        if os.getenv('
        KUBERNETES_SERVICE_HOST'):
            config.load_incluster_config()
        else:
            config.load_kube_config()

        # Create a Kubernetes API client
        self.v1 = client.CoreV1Api()
```

- Add an entry point at the bottom to allow the test to be run from the command line, as follows:

```
if __name__ == '__main__':
    main()
```

- The output should be formatted as a JSON instance that conforms to the JSON schema below.

As an example, for the schema {\"properties\": {\"foo\": {\"title\": \"Foo\", \"description\": \"a list of strings\", \"type\": \"array\", \"items\": {\"type\": \"string\"}}}, \"required\": [\"foo\"]}\nthe object {\"foo\": [\"bar\", \"baz\"]} is a well-formatted instance of the schema. The object {\"properties\": {\"foo\": [\"bar\", \"baz\"]}} is not well-formatted.

Here is the output schema:

```
{
  "properties": {
    "thought": {
      "title": "Thought",
      "description": "Describe how you add
      the threshold assertion to the
      inspection Python script.",
      "type": "string"
    },
    "code": {
      "title": "Code",
      "description": "Python unit test code.
       Implement a for loop that checks the
      status every second for the duration,
      and implement assertion for the
      summary at the end.\n- Please add a
      Add a entry point at the bottom to
      allow the test to be run from the
      command line.\n- Please add argparse
      '--duration' (type=int) so that users
      can specify the loop duration. Write
      only the content of the code, and for
      dictionary values, enclose them within
       a pair of single double quotes (\")
      .",
      "type": "string"
    }
  },
  "required": [
    "thought",
    "code"
  ]
}
```

Human:
The steady state:
{steady_state_name}: {steady_state_thought}

The steady state was inspected with the following python code of k8s client libraries:
{script (inspection_summary without results)}

The threshold of the steady state: **{steady_state_threshold}; {steady_state_threshold_description}**

Given the above steady state, command, and threshold, please write a Python unit test to check if the steady state satisfies the threshold.
The threshold in the unit test must exactly match the threshold defined above. Implement it to support variable durations. Use a representative value (e.g., percentage, ratio, etc.) for the threshold. NEVER use any fixed absolute values for the threshold.

In the verification loop, the prompts below will be stacked as history

AI:
{output}

User:
Your current unittest cause errors when coducted.
The error message is as follows:
{error_message}

Please analyze the reason why the errors occur, then fix the errors.
Always keep the following rules:
- Ensure that the implementation supports variable durations again.
- NEVER repeat the same fixes that have been made in the past.
- Fix only the parts related to the errors without changing the original content.
- **{the same format instructions as in the System role}**

Prompt 10: Agent # 1-3-b for writing a VaC script of K6 Javascript

System:
You are a helpful AI assistant for writing unit tests in k6.
Given a steady state, k6 javascript to inspect it, and its threshold, please write a k6 unit test to verify if the steady state satisfies the threshold by adding threshold options.
Always keep the following rules:
- Include as many comments as possible in your code so that humans can easily understand what you did later.
- Add "thresholds" in "options" section to the given k6 javascript.
- The output should be formatted as a JSON instance that conforms to the JSON schema below.

As an example, for the schema {\"properties\": {\"foo\": {\"title\": \"Foo\", \"description\": \"a list of strings\", \"type\": \"array\", \"items\": {\"type\": \"string\"}}}, \"required\": [\"foo\"]}\nthe object {\"foo\": [\"bar\", \"baz\"]} is a well-formatted instance of the schema. The object {\"properties\": {\"foo\": [\"bar\", \"baz\"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "thought": {
      "title": "Thought",
      "description": "Describe how you add
      the threshold check to the inspection
      K6 script.",
      "type": "string"
    },
    "code": {
      "title": "Code",
      "description": "K6 unit test code (
      javascript). Write only the content of
       the code, and for dictionary values,
      enclose them within a pair of single
      double quotes (\").",
      "type": "string"
    }
  },
  "required": [
    "thought",
    "code"
  ]
}
```

Human:
The steady state:
{steady_state_name}: {steady_state_thought}

The steady state can be inspected with the following k6 javascript:
{script (inspection_summary without results)}

The threshold of the steady state:
{steady_state_threshold}; {threshold_description}

Given the above steady state, k6 javascript, and threshold, please write a k6 unit test to check if the steady state satisfies the threshold by adding threshold options.
The threshold in the unit test must exactly match the threshold defined above.

In the verification loop, the prompts below will be stacked as history

AI:
{output}

User:
Your current unittest cause errors when coducted.
The error message is as follows:
{error_message}

Please analyze the reason why the errors occur, then fix the errors.
Always keep the following rules:
- Ensure that the implementation supports variable durations again.
- NEVER repeat the same fixes that have been made in the past.
- Fix only the parts related to the errors without changing the original content.
- **{the same format instructions as in the System role}**

Example 12: steady_state_threshold

The Pod should be in the 'Running' state at least 90\% of the time during the observation period.

Example 13: threshold_description

The steady state we are considering is the 'example-pod-running-state', which requires the Pod to be in a 'Running' state. The current state shows that the Pod was running 5 out of 5 seconds, which is 100\% of the time. To account for some fluctuations and ensure the threshold is reasonable, we can set a threshold that allows for a small percentage of time where the Pod might not be in the 'Running' state due to transient issues. A reasonable threshold could be that the Pod should be in the 'Running' state at least 90\% of the time during the observation period. This allows for some tolerance while still ensuring the Pod is mostly available.

Prompt 11: Agent # 1-4 for checking the listed steady states are sufficient

System:
You are a helpful AI assistant for Chaos Engineering.
Given K8s manifests for a system, user's instructions, and steady states already defined, you will determine whether an additional steady state needs to be defined.
Always keep the following rules:
- Clearly describe the reason for determining whether an additional steady state is needed.
- You may also cite the user's instructions as the reason.
- The output should be formatted as a JSON instance that conforms to the JSON schema below.

As an example, for the schema {\"properties\": {\"foo\": {\"title\": \"Foo\", \"description\": \"a list of strings\", \"type\": \"array\", \"items\": {\"type\": \"string\"}}}, \"required\": [\"foo\"]}\nthe object {\"foo\": [\"bar\", \"baz\"]} is a well-formatted instance of the schema. The object {\"properties\": {\"foo\": [\"bar\", \"baz\"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "thought": {
      "title": "Thought",
      "description": "Describe your thought
      process of determing whether an
      additional steady states is needed.",
      "type": "string"
    },
    "requires_addition": {
      "title": "Requires Addition",
      "description": "The necessity of an
      additional steady state. If it is
      needed, select 'True'; otherwise
      select 'False'.",
      "type": "boolean"
    }
  },
  "required": [
    "thought",
    "requires_addition"
  ]
}
```

Human:
Here is the overview of my system:
{user_input2}

```
# Please follow the instructions below
regarding Chaos Engineering:
{ce_instructions}

# Steady states already defined are as
follows:
{predefined_steady_states}

Now, determine whether an additional steady
state needs to be defined.
```

Prompt 12: Agent # 1-5 for drafting failure injection

System:

```
You are a helpful AI assistant for Chaos
Engineering.
Given k8s manifests for a system, the steady
 states of the system, and user's
instructions for Chaos Engineering, you will
 define the most impactful fault injections
to reveal potential weaknesses of the system
, such as insufficient recovery functions,
resource allocation, redundancy, etc.
Always keep the following rules:
- First, assume a real-world event that may
be most impactful in the the system, such as
 promotion campaign, cyber attacks,
disasters, etc.
- Then, define the most impactful fault
injections to reveal potential weaknesses of
 the given system while simulating the
assumed real-world event.
- Prioritize fault injections that target
the system's weak resources related to the
steady states to verify whether those
resources can handle the faults and the
steady states can be maintained.
- The injected faults should be selected
from the following fault types of {
ce_tool_name}:
  - PodChaos: simulates Pod failures, such
  as Pod node restart, Pod's persistent
  unavailablility, and certain container
  failures in a specific Pod. The supported
  subtypes include 'pod-failure', 'pod-kill
  ', 'container-kill'.
  - NetworkChaos: simulates network failures
  , such as network latency, packet loss,
  packet disorder, and network partitions.
  - DNSChaos: simulates DNS failures, such
  as the parsing failure of DNS domain name
  and the wrong IP address returned.
  - HTTPChaos: simulates HTTP communication
  failures, such as HTTP communication
  latency.
  - StressChaos: simulates CPU race or
  memory race.
  - IOChaos: simulates the I/O failure of an
   application file, such as I/O delays,
  read and write failures.
  - TimeChaos: simulates the time jump
  exception.
- The output should be formatted as a JSON
instance that conforms to the JSON schema
below.

As an example, for the schema {\"properties
\": {\"foo\": {\"title\": \"Foo\", \"
description\": \"a list of strings\", \"type
\": \"array\", \"items\": {\"type\": \"
string\"}}}, \"required\": [\"foo\"]}\nthe
object {\"foo\": [\"bar\", \"baz\"]} is a
well-formatted instance of the schema. The
object {\"properties\": {\"foo\": [\"bar\",
\"baz\"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "event": {
      "title": "Event",
      "description": "Consider a real-world
      fault event that may be most impactful
       of the system, such as promotion
      campaign, cyber attacks, disasters,
      etc.",
      "type": "string"
    },
    "thought": {
      "title": "Thought",
      "description": "Write down your
      thought process to define a sequence
      of fault injections that exploit the
      system's weaknesses of while
      simulating the fault event: 1) how the
       system's weaknesses affect the steady
       state; 2) how each fault injection
      exploit the system's weaknesses; 3)
      how the sequence simulates the
      phenamena in the fault event (consider
       carefully the sequence order).
      Prioritize fault injections that
      directly attack the weaknessses of the
       system, such as insufficient recovery
       functions, resource allocation,
      redundancy, etc.",
      "type": "string"
    },
    "faults": {
      "title": "Faults",
      "description": "Define a sequence of
      fault injections that exploit the
      system's vulnerabilities to the
      fullest according to the above
      thoughts. In the inner list, a set of
      simultaneously injected faults are
      listed, while in the outer list, the
      sets are listed in the injection order
      . For example, [[fault_a], [fault_b,
      fault_c]] indicates that fault_a is
      injected, then fault_b and fault_c are
       injected simultaneously.",
      "type": "array",
      "items": {
        "type": "array",
        "items": {
          "\$ref": "#/definitions/Fault"
        }
      }
    }
  },
  "required": [
    "event",
    "thought",
    "faults"
  ],
  "definitions": {
    "Fault": {
      "title": "Fault",
      "type": "object",
      "properties": {
        "name": {
          "title": "Name",
          "description": "Select a fault
          type from [\"PodChaos\", \"
          NetworkChaos\", \"DNSChaos\", \"
          HTTPChaos\", \"StressChaos\", \"
          IOChaos\", \"TimeChaos\"]",
          "enum": [
            "PodChaos",
            "NetworkChaos",
            "DNSChaos",
            "HTTPChaos",
            "StressChaos",
            "IOChaos",
            "TimeChaos"
          ],
          "type": "string"
        },
        "name_id": {
          "title": "Name Id",
          "description": "An identifier to
          prevent name conflicts when the
```
```

```
          same Fault appears. Assign numbers
           starting from 0 in sequential
          order to prevent name conflicts.",
          "type": "integer"
        },
        "scope": {
          "title": "Scope",
          "description": "Specify only the
          fault injection scope (i.e., the
          target resource where the fault is
           injected) in advance here.",
          "type": "object",
          "additionalProperties": {
            "type": "string"
          }
        }
      },
      "required": [
        "name",
        "name_id",
        "scope"
      ]
    }
  }
}
```

Human:
Here is the overview of my system:
{user_input2}

Steady states of the network system defined by the manifests are the following:
{steady_states}

Please follow the instructions below regarding Chaos Engineering as necessary:
{ce_instructions}

Now, please define fault injections to reveal the system's vulnerabilities.

Example 14: **steady_states**

Steady states of the network system defined by the manifests are the following:
2 steady states are defined.

1st steady states:
- Name: example-pod-running-state
- Description: The first issue to address is the Pod's restart policy set to 'Never' in the 'nginx/pod.yaml' manifest. This is a critical issue because if the Pod fails, it will not restart automatically, leading to potential downtime. A steady state related to this issue would be to ensure that the Pod is running and available. This can be measured by checking the number of running Pods. Since there is only one Pod, the steady state is that the Pod should always be in a 'Running' state.
- Threshold for the steady state: The Pod should be in the 'Running' state at least 90% of the time during the observation period.; The steady state we are considering is the 'example-pod-running-state', which requires the Pod to be in a 'Running' state. The current state shows that the Pod was running 5 out of 5 seconds, which is 100% of the time. To account for some fluctuations and ensure the threshold is reasonable, we can set a threshold that allows for a small percentage of time where the Pod might not be in the 'Running' state due to transient issues. A reasonable threshold could be that the Pod should be in the 'Running' state at least 90% of the time during the observation period. This allows for some tolerance while still ensuring the Pod is mostly available.
- Whether the steady state meets the threshold is determined by the following Python script with K8s API:
```
import os
import time
import argparse
from kubernetes import client, config
from unittest_base import K8sAPIBase
class TestPodRunningState(K8sAPIBase):
    ...

if __name__ == '__main__':
    parser = argparse.ArgumentParser(
    description='Test the running state of a
     Pod.')
    parser.add_argument('--duration', type=
    int, default=5, help='Duration to check
    the Pod status in seconds.')
    args = parser.parse_args()
    # Create an instance of the test class
    and run the test
    test = TestPodRunningState(namespace='
    default', pod_name='example-pod',
    duration=args.duration)
    test.test_pod_running_state()
```

2nd steady states:
- Name: example-service-http-response-state
- Description: The next issue to address is the lack of redundancy due to the single Pod deployment in the 'nginx/pod.yaml' manifest. This is a significant issue because if the Pod fails, there is no automatic recovery or redundancy, which can lead to service unavailability. A steady state related to this issue would be to ensure that the Service is able to route traffic to the Pod. This can be measured by checking the Service's ability to respond to HTTP requests successfully. Since the Service is supposed to expose the Pod on port 80, the steady state is that the Service should respond with a successful HTTP status code (e.g., 200 OK) for a certain percentage of requests.
- Threshold for the steady state: 95% of HTTP requests should return a 200 OK status.; The steady state for the system is defined as the Service's ability to respond with a successful HTTP status code (200 OK) for a certain percentage of requests. The current state shows that 100% of the requests received a 200 OK status, which indicates a perfectly healthy system. However, to account for potential fluctuations and to ensure the threshold is reasonable, we should allow for some tolerance. A common practice is to set a threshold slightly below the current perfect state to accommodate minor, non-critical issues that might occur during normal operations. Therefore, setting the threshold at 95% ensures that the system is considered healthy as long as it maintains a high level of successful responses, while still allowing for some minor issues.
- Whether the steady state meets the threshold is determined by the following K6 Javascript:
```
import http from 'k6/http';\nimport { check
} from 'k6';

export const options = {
  vus: 5,
  duration: '5s',
  thresholds: {
    'http_req_failed': ['rate<0.05'],
  },
};

export default function () {
  const res = http.get('http:\/\/example-
  service.default.svc.cluster.local:80');
```

```
  check(res, {
    'is status 200': (r) => r.status ===
    200,
  });
}
```

Prompt 13: Agent # 1-6 for determining detailed failure parameters

System:
You are a helpful AI assistant for Chaos Engineering.
Given k8s manifests that define a network system, its steady states, and a fault type that may affect the steady states in the system, please detail the parameters of the fault.
Always keep the following rules:
- Pay attention to namespace specification. If the namespace is specified in the manifest, it is deployed with the namespace. If not, it is deployed with the 'default' namespace.
- The parameters follow the format of Chaos Mesh.

Human:
Here is the overview of my system:
{user_input2}

Steady states of my system:
{steady_states}

A fault scenario that may occur in my system and may affect the steady states:
{fault_scenario}

Please follow the instructions below regarding Chaos Engineering as necessary:
{ce_instructions}

Now, please detail the parameters of the fault "**{refined_fault_type}**".
{detailed_param_instructions}

In the verification loop, the prompts below will be stacked as history

AI:
{output}

Human:
Your current fault parameters cause errors when conducted.
The error message is as follows:
{error_message}

Please analyze the reason why the errors occur, then fix the errors.
Always keep the following rules:
- NEVER repeat the same fixes that have been made in the past.
- Fix only the parts related to the errors without changing the original intent.

Example 15: `fault_scenario`

```
An assumed fault scenario is as follows:
- Event: Cyber Attack Simulation
- Used Chaos Engineering tool: Chaos Mesh
- Faults to simulate the event: [[{'name': '
PodChaos', 'name_id': 0, 'scope': {'pod': '
example-pod'}}], [{'name': 'NetworkChaos', '
name_id': 0, 'scope': {'service': 'example-
service'}}]]
- Description: Given the system's weaknesses
, a cyber attack targeting the web server
could be highly impactful. The Pod's restart
 policy set to 'Never' and the single Pod
deployment without redundancy are critical
vulnerabilities. If the Pod fails, it will
not restart, leading to downtime, and the
lack of redundancy means there is no backup
to handle traffic. To simulate a cyber
attack, we can inject faults that exploit
these weaknesses. First, we will use
PodChaos to simulate a Pod failure, which
will test the system's ability to maintain
the 'example-pod-running-state'. Since the
Pod will not restart automatically, this
will directly impact the steady state. Next,
 we will use NetworkChaos to simulate
network latency, which will test the system'
s ability to maintain the 'example-service-
http-response-state'. This sequence
simulates a cyber attack where the Pod is
targeted first, followed by network
disruptions, revealing the system's
vulnerabilities in handling such events.
```

Example 16: `refined_fault_type`

```
NetworkChaos({'service': 'example-service'})
```

Dynamic instruction 1: `detailed_param_instructions` for `PodChaos`

```
The output should be formatted as a JSON
instance that conforms to the JSON schema
below.

As an example, for the schema {"properties":
 {"foo": {"title": "Foo", "description": "a
list of strings", "type": "array", "items":
{"type": "string"}}}, "required": ["foo"]}
the object {"foo": ["bar", "baz"]} is a well
-formatted instance of the schema. The
object {"properties": {"foo": ["bar", "baz
"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "action": {
      "title": "Action",
      "description": "Specifies the fault
      type from 'pod-failure', 'pod-kill',
      or 'container-kill'. Note that you may
       select 'pod-failure' only when the
      target Pod's container has
      livenessProbe and readinessProbe
      defined.",
      "example": "pod-kill",
      "enum": [
        "pod-failure",
        "pod-kill",
        "container-kill"
      ],
      "type": "string"
    },
    "mode": {
      "title": "Mode",
      "description": "Specifies the mode of
      the experiment. The mode options
      include 'one' (selecting a random Pod)
      , 'all' (selecting all eligible Pods),
       'fixed' (selecting a specified number
       of eligible Pods), 'fixed-percent' (
```
```

```
        selecting a specified percentage of
        Pods from the eligible Pods), and '
        random-max-percent' (selecting the
        maximum percentage of Pods from the
        eligible Pods)",
        "example": "one",
        "enum": [
          "one",
          "all",
          "fixed",
          "fixed-percent",
          "random-max-percent"
        ],
        "type": "string"
      },
      "value": {
        "title": "Value",
        "description": "Provides parameters
        for the mode configuration, depending
        on mode.For example, when mode is set
        to fixed-percent, value specifies the
        percentage of Pods.",
        "example": "1",
        "type": "string"
      },
      "selector": {
        "title": "Selector",
        "description": "Specifies the target
        Pod.",
        "example": null,
        "allOf": [
          {
            "$ref": "#/definitions/Selectors"
          }
        ]
      },
      "containerNames": {
        "title": "Containernames",
        "description": "When you configure
        action to container-kill, this
        configuration is mandatory to specify
        the target container name for
        injecting faults.",
        "example": [
          "prometheus"
        ],
        "type": "array",
        "items": {
          "type": "string"
        }
      }
    },
    "required": [
      "action",
      "mode",
      "selector"
    ],
    "definitions": {
      "SetBasedRequirements": {
        "title": "SetBasedRequirements",
        "type": "object",
        "properties": {
          "key": {
            "title": "Key",
            "description": "Label key",
            "type": "string"
          },
          "operator": {
            "title": "Operator",
            "description": "Select an operator
            .",
            "enum": [
              "In",
              "NotIn",
              "Exists",
              "DoesNotExist"
            ],
            "type": "string"
          },
          "values": {
            "title": "Values",
            "description": "Label values. The
            values set must be non-empty in
            the case of In and NotIn.",
            "type": "array",
            "items": {
              "type": "string"
            }
          }
        },
        "required": [
          "key",
          "operator",
          "values"
        ]
      },
      "Selectors": {
        "title": "Selectors",
        "type": "object",
        "properties": {
          "namespaces": {
            "title": "Namespaces",
            "description": "Specifies the
            namespace of the experiment's
            target Pod. If this selector is
            None, Chaos Mesh will set it to
            the namespace of the current Chaos
             experiment.",
            "type": "array",
            "items": {
              "type": "string"
            }
          },
          "labelSelectors": {
            "title": "Labelselectors",
            "description": "Specifies the
            label-key/value pairs that the
            experiment's target Pod must have.
             If multiple labels are specified,
             the experiment target must have
            all the labels specified by this
            selector.",
            "type": "object",
            "additionalProperties": {
              "type": "string"
            }
          },
          "expressionSelectors": {
            "title": "Expressionselectors",
            "description": "Specifies a set of
             expressions that define the label
            's rules to specifiy the
            experiment's target Pod.",
            "example": [
              {
                "key": "tier",
                "operator": "In",
                "values": [
                  "cache"
                ]
              },
              {
                "key": "environment",
                "operator": "NotIn",
                "values": [
                  "dev"
                ]
              }
            ],
            "type": "array",
            "items": {
              "$ref": "#/definitions/
              SetBasedRequirements"
            }
          },
          "annotationSelectors": {
            "title": "Annotationselectors",
            "description": "Specifies the
            annotation-key/value pairs that
            the experiment's target Pod must
            have. If multiple annotations are
            specified, the experiment target
            must have all annotations
            specified by this selector.",
            "type": "object",
            "additionalProperties": {
              "type": "string"
            }
          },
          "fieldSelectors": {
```

```
    "title": "Fieldselectors",
    "description": "Specifies the
    field-key/value pairs of the
    experiment's target Pod. If
    multiple fields are specified, the
     experiment target must have all
    fields set by this selector.",
    "example": {
      "metadata.name": "my-pod",
      "metadata.namespace": "dafault"
    },
    "type": "object",
    "additionalProperties": {
      "type": "string"
    }
  },
  "podPhaseSelectors": {
    "title": "Podphaseselectors",
    "description": "Specifies the
    phase of the experiment's target
    Pod. If this selector is None, the
     target Pod's phase is not limited
    .",
    "type": "array",
    "items": {
      "enum": [
        "Pending",
        "Running",
        "Succeeded",
        "Failed",
        "Unknown"
      ],
      "type": "string"
    }
  },
  "nodeSelectors": {
    "title": "Nodeselectors",
    "description": "Specifies the node
    -label-key/value pairs to which
    the experiment's target Pod
    belongs.",
    "type": "object",
    "additionalProperties": {
      "type": "string"
    }
  },
  "nodes": {
    "title": "Nodes",
    "description": "Specifies the node
     to which the experiment's target
    Pod belongs. The target Pod can
    only belong to one node in the
    configured node list. If multiple
    node labels are specified, the
    node to which the experiment's
    target Pod belongs must have all
    labels specified by this selector
    .",
    "type": "array",
    "items": {
      "type": "string"
    }
  },
  "pods": {
    "title": "Pods",
    "description": "Specifies the
    namespaces and list of the
    experiment's target Pods. If you
    have specified this selector,
    Chaos Mesh ignores other
    configured selectors.",
    "example": {
      "default": [
        "pod-0",
        "pod-2"
      ]
    },
    "type": "object",
    "additionalProperties": {
      "type": "array",
      "items": {
        "type": "string"
      }
    }
  }
```

Dynamic instruction 2: **detailed_param_instructions** for **NetworkChaos**

```
he output should be formatted as a JSON
instance that conforms to the JSON schema
below.

As an example, for the schema {"properties":
 {"foo": {"title": "Foo", "description": "a
list of strings", "type": "array", "items":
{"type": "string"}}}, "required": ["foo"]}
the object {"foo": ["bar", "baz"]} is a well
-formatted instance of the schema. The
object {"properties": {"foo": ["bar", "baz
"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "action": {
      "title": "Action",
      "description": "Indicates the specific
       fault type. Available types include:
      netem, delay (network delay), loss (
      packet loss), duplicate (packet
      duplicating), corrupt (packet corrupt)
      , partition (network partition), and
      bandwidth (network bandwidth limit).
      After you specify action field,
      specify action-related fields for
      other necessary field configuration.",
      "example": "Partition",
      "enum": [
        "netem",
        "delay",
        "loss",
        "duplicate",
        "corrupt",
        "partition",
        "bandwidth"
      ],
      "type": "string"
    },
    "direction": {
      "title": "Direction",
      "description": "Indicates the
      direction of target packets. Available
       vaules include from (the packets from
       target), to (the packets to target),
      and both (the packets from or to
      target). This parameter makes Chaos
      only take effect for a specific
      direction of packets.",
      "default": "to",
      "example": "both",
      "enum": [
        "from",
        "to",
        "both"
      ],
      "type": "string"
    },
    "target": {
      "title": "Target",
      "description": "Used in combination
      with direction, making Chaos only
      effective for some packets. 'from' and
       'both' direction cannot be used when
      targets is empty in netem action.",
      "allOf": [
        {
          "$ref": "#/definitions/Selector"
        }
      }
    }
  }
}
```
```

```
      ]
    },
    "mode": {
      "title": "Mode",
      "description": "Specifies the mode of
      the experiment. The mode options
      include one (selecting a random Pod),
      all (selecting all eligible Pods),
      fixed (selecting a specified number of
       eligible Pods), fixed-percent (
      selecting a specified percentage of
      Pods from the eligible Pods), and
      random-max-percent (selecting the
      maximum percentage of Pods from the
      eligible Pods)",
      "example": "one",
      "enum": [
        "one",
        "all",
        "fixed",
        "fixed-percent",
        "random-max-percent"
      ],
      "type": "string"
    },
    "value": {
      "title": "Value",
      "description": "Provides parameters
      for the mode configuration, depending
      on mode. For example, when mode is set
       to fixed-percent, value specifies the
       percentage of Pods.",
      "example": "1",
      "type": "string"
    },
    "selector": {
      "title": "Selector",
      "description": "Specifies the target
      Pod.",
      "allOf": [
        {
          "$ref": "#/definitions/Selectors"
        }
      ]
    },
    "externalTargets": {
      "title": "Externaltargets",
      "description": "Indicates the network
      targets except for Kubernetes, which
      can be IPv4 addresses or domains. This
       parameter only works with direction:
      to.",
      "example": [
        "1.1.1.1",
        "www.google.com"
      ],
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "device": {
      "title": "Device",
      "description": "Specifies the affected
       network interface",
      "example": "eth0",
      "type": "string"
    },
    "delay": {
      "title": "Delay",
      "description": "When setting action to
       delay means simulating network delay
      fault, you also need to configure this
       parameters.",
      "allOf": [
        {
          "$ref": "#/definitions/Deplay"
        }
      ]
    },
    "loss": {
      "title": "Loss",
      "description": "When setting action to
       loss means simulating packet loss
      fault, you can also configure this
      parameters.",
      "allOf": [
        {
          "$ref": "#/definitions/Loss"
        }
      ]
    },
    "duplicated": {
      "title": "Duplicated",
      "description": "When setting action to
       duplicate, meaning simulating package
       duplication, you can also set this
      parameters.",
      "allOf": [
        {
          "$ref": "#/definitions/Duplicate"
        }
      ]
    },
    "corrupt": {
      "title": "Corrupt",
      "description": "When setting action to
       corrupt means simulating package
      corruption fault, you can also
      configure the following parameters.",
      "allOf": [
        {
          "$ref": "#/definitions/Corrupt"
        }
      ]
    },
    "rate": {
      "title": "Rate",
      "description": "When setting action to
       rate means simulating bandwidth rate
      fault, you also need to configure this
       parameters. This action is similar to
       bandwidth/rate below, however, the
      key distinction is that this action
      can combine with other netem actions
      listed above. However, if you require
      more control over the bandwidth
      simulation such as limiting the buffer
       size, select the bandwidth action.",
      "allOf": [
        {
          "$ref": "#/definitions/Rate"
        }
      ]
    },
    "bandwidth": {
      "title": "Bandwidth",
      "description": "When setting 'action'
      to 'bandwidth' means simulating
      bandwidth limit fault, you also need
      to configure this parameters. This
      action is mutually exclusive with any
      netem action defined above. If you
      need to inject bandwidth rate along
      with other network failures such as
      corruption, use the rate action
      instead.",
      "allOf": [
        {
          "$ref": "#/definitions/Bandwidth"
        }
      ]
    }
  },
  "required": [
    "action",
    "mode",
    "selector"
  ],
  "definitions": {
    "SetBasedRequirements": {
      "title": "SetBasedRequirements",
      "type": "object",
      "properties": {
        "key": {
          "title": "Key",
          "description": "Label key",
          "type": "string"
        },
        "operator": {
```

```
        "title": "Operator",
        "description": "Select an operator
        .",
        "enum": [
          "In",
          "NotIn",
          "Exists",
          "DoesNotExist"
        ],
        "type": "string"
      },
      "values": {
        "title": "Values",
        "description": "Label values. The
        values set must be non-empty in
        the case of In and NotIn.",
        "type": "array",
        "items": {
          "type": "string"
        }
      }
    },
    "required": [
      "key",
      "operator",
      "values"
    ]
  },
  "Selectors": {
    "title": "Selectors",
    "type": "object",
    "properties": {
      "namespaces": {
        "title": "Namespaces",
        "description": "Specifies the
        namespace of the experiment's
        target Pod. If this selector is
        None, Chaos Mesh will set it to
        the namespace of the current Chaos
         experiment.",
        "type": "array",
        "items": {
          "type": "string"
        }
      },
      "labelSelectors": {
        "title": "Labelselectors",
        "description": "Specifies the
        label-key/value pairs that the
        experiment's target Pod must have.
         If multiple labels are specified,
         the experiment target must have
        all the labels specified by this
        selector.",
        "type": "object",
        "additionalProperties": {
          "type": "string"
        }
      },
      "expressionSelectors": {
        "title": "Expressionselectors",
        "description": "Specifies a set of
         expressions that define the label
        's rules to specifiy the
        experiment's target Pod.",
        "example": [
          {
            "key": "tier",
            "operator": "In",
            "values": [
              "cache"
            ]
          },
          {
            "key": "environment",
            "operator": "NotIn",
            "values": [
              "dev"
            ]
          }
        ],
        "type": "array",
        "items": {
          "$ref": "#/definitions/
          SetBasedRequirements"
        }
      },
      "annotationSelectors": {
        "title": "Annotationselectors",
        "description": "Specifies the
        annotation-key/value pairs that
        the experiment's target Pod must
        have. If multiple annotations are
        specified, the experiment target
        must have all annotations
        specified by this selector.",
        "type": "object",
        "additionalProperties": {
          "type": "string"
        }
      },
      "fieldSelectors": {
        "title": "Fieldselectors",
        "description": "Specifies the
        field-key/value pairs of the
        experiment's target Pod. If
        multiple fields are specified, the
         experiment target must have all
        fields set by this selector.",
        "example": {
          "metadata.name": "my-pod",
          "metadata.namespace": "dafault"
        },
        "type": "object",
        "additionalProperties": {
          "type": "string"
        }
      },
      "podPhaseSelectors": {
        "title": "Podphaseselectors",
        "description": "Specifies the
        phase of the experiment's target
        Pod. If this selector is None, the
         target Pod's phase is not limited
        .",
        "type": "array",
        "items": {
          "enum": [
            "Pending",
            "Running",
            "Succeeded",
            "Failed",
            "Unknown"
          ],
          "type": "string"
        }
      },
      "nodeSelectors": {
        "title": "Nodeselectors",
        "description": "Specifies the node
        -label-key/value pairs to which
        the experiment's target Pod
        belongs.",
        "type": "object",
        "additionalProperties": {
          "type": "string"
        }
      },
      "nodes": {
        "title": "Nodes",
        "description": "Specifies the node
         to which the experiment's target
        Pod belongs. The target Pod can
        only belong to one node in the
        configured node list. If multiple
        node labels are specified, the
        node to which the experiment's
        target Pod belongs must have all
        labels specified by this selector
        .",
        "type": "array",
        "items": {
          "type": "string"
        }
      },
      "pods": {
        "title": "Pods",
        "description": "Specifies the
        namespaces and list of the
        experiment's target Pods. If you
```

```
        have specified this selector,
        Chaos Mesh ignores other
        configured selectors.",
        "example": {
          "default": [
            "pod-0",
            "pod-2"
          ]
        },
        "type": "object",
        "additionalProperties": {
          "type": "array",
          "items": {
            "type": "string"
          }
        }
      }
    }
  },
  "Selector": {
    "title": "Selector",
    "type": "object",
    "properties": {
      "mode": {
        "title": "Mode",
        "description": "Specifies the mode
         of the experiment. The mode
        options include one (selecting a
        random Pod), all (selecting all
        eligible Pods), fixed (selecting a
         specified number of eligible Pods
        ), fixed-percent (selecting a
        specified percentage of Pods from
        the eligible Pods), and random-max
        -percent (selecting the maximum
        percentage of Pods from the
        eligible Pods)",
        "example": "one",
        "enum": [
          "one",
          "all",
          "fixed",
          "fixed-percent",
          "random-max-percent"
        ],
        "type": "string"
      },
      "selector": {
        "title": "Selector",
        "description": "Specifies the
        target Pod.",
        "example": null,
        "allOf": [
          {
            "$ref": "#/definitions/
            Selectors"
          }
        ]
      }
    },
    "required": [
      "mode",
      "selector"
    ]
  },
  "Reorder": {
    "title": "Reorder",
    "type": "object",
    "properties": {
      "reorder": {
        "title": "Reorder",
        "description": "Indicates the
        probability to reorder",
        "default": "0",
        "example": "0.5",
        "type": "string"
      },
      "correlation": {
        "title": "Correlation",
        "description": "Indicates the
        correlation between this time's
        length of delay time and the
        previous time's length of delay
        time. Range of value: [0, 100]",
        "default": "0",
        "example": "50",
        "type": "string"
      },
      "gap": {
        "title": "Gap",
        "description": "Indicates the gap
        before and after packet reordering
        ",
        "default": 0,
        "example": 5,
        "type": "integer"
      }
    }
  },
  "Deplay": {
    "title": "Deplay",
    "type": "object",
    "properties": {
      "latency": {
        "title": "Latency",
        "description": "Indicates the
        network latency",
        "example": "2ms",
        "type": "string"
      },
      "correlation": {
        "title": "Correlation",
        "description": "Indicates the
        correlation between the current
        latency and the previous one.
        Range of value: [0, 100]. Specify
        only the number. NEVER include any
         units.",
        "example": "50",
        "type": "string"
      },
      "jitter": {
        "title": "Jitter",
        "description": "Indicates the
        range of the network latency",
        "example": "1ms",
        "type": "string"
      },
      "reorder": {
        "title": "Reorder",
        "description": "Indicates the
        status of network packet
        reordering",
        "allOf": [
          {
            "$ref": "#/definitions/Reorder
            "
          }
        ]
      }
    }
  },
  "Loss": {
    "title": "Loss",
    "type": "object",
    "properties": {
      "loss": {
        "title": "Loss",
        "description": "Indicates the
        probability of packet loss. Range
        of value: [0, 100]. Specify only
        the number. NEVER include any
        units.",
        "default": "0",
        "example": "50",
        "type": "string"
      },
      "correlation": {
        "title": "Correlation",
        "description": "Indicates the
        correlation between the
        probability of current packet loss
         and the previous time's packet
        loss. Range of value: [0, 100].
        Specify only the number. NEVER
        include any units.",
        "default": "0",
        "example": "50",
        "type": "string"
      }
```

```
    }
  },
  "Duplicate": {
    "title": "Duplicate",
    "type": "object",
    "properties": {
      "duplicate": {
        "title": "Duplicate",
        "description": "Indicates the
        probability of packet duplicating.
         Range of value: [0, 100]. Specify
         only the number. NEVER include
        any units.",
        "default": "0",
        "example": "50",
        "type": "string"
      },
      "correlation": {
        "title": "Correlation",
        "description": "Indicates the
        correlation between the
        probability of current packet
        duplicating and the previous time'
        s packet duplicating. Range of
        value: [0, 100]. Specify only the
        number. NEVER include any units.",
        "default": "0",
        "example": "50",
        "type": "string"
      }
    }
  },
  "Corrupt": {
    "title": "Corrupt",
    "type": "object",
    "properties": {
      "corrupt": {
        "title": "Corrupt",
        "description": "Indicates the
        probability of packet corruption.
        Range of value: [0, 100]. Specify
        only the number. NEVER include any
         units.",
        "default": "0",
        "example": "50",
        "type": "string"
      },
      "correlation": {
        "title": "Correlation",
        "description": "Indicates the
        correlation between the
        probability of current packet
        corruption and the previous time's
         packet corruption. Range of value
        : [0, 100]. Specify only the
        number. NEVER include any units.",
        "default": "0",
        "example": "50",
        "type": "string"
      }
    }
  },
  "Rate": {
    "title": "Rate",
    "type": "object",
    "properties": {
      "rate": {
        "title": "Rate",
        "description": "Indicates the rate
         of bandwidth limit. Allows bit,
        kbit, mbit, gbit, tbit, bps, kbps,
         mbps, gbps, tbps unit. bps means
        bytes per second",
        "example": "1mbps",
        "type": "string"
      }
    }
  },
  "Bandwidth": {
    "title": "Bandwidth",
    "type": "object",
    "properties": {
      "rate": {
        "title": "Rate",
        "description": "Indicates the rate
         of bandwidth limit. Allows bit,
        kbit, mbit, gbit, tbit, bps, kbps,
         mbps, gbps, tbps unit. bps means
        bytes per second",
        "example": "1mbps",
        "type": "string"
      },
      "limit": {
        "title": "Limit",
        "description": "Indicates the
        number of bytes waiting in queue",
        "example": 1,
        "type": "integer"
      },
      "buffer": {
        "title": "Buffer",
        "description": "Indicates the
        maximum number of bytes that can
        be sent instantaneously",
        "example": 1,
        "type": "integer"
      },
      "peakrate": {
        "title": "Peakrate",
        "description": "Indicates the
        maximum consumption of bucket (
        usually not set)",
        "example": 1,
        "type": "integer"
      },
      "minburst": {
        "title": "Minburst",
        "description": "Indicates the size
         of peakrate bucket (usually not
        set)",
        "example": 1,
        "type": "integer"
      }
    }
  }
}
```

Dynamic instruction 3: **detailed_param_instructions** for **DNSChaos**

```
he output should be formatted as a JSON
instance that conforms to the JSON schema
below.

As an example, for the schema {"properties":
 {"foo": {"title": "Foo", "description": "a
list of strings", "type": "array", "items":
{"type": "string"}}}, "required": ["foo"]}
the object {"foo": ["bar", "baz"]} is a well
-formatted instance of the schema. The
object {"properties": {"foo": ["bar", "baz
"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "action": {
      "title": "Action",
      "description": "Defines the behavior
      of DNS fault from 'random' or 'error'.
       When the value is random, DNS service
       returns a random IP address; when the
       value is error, DNS service returns
      an error.",
      "example": "random",
      "enum": [
        "random",
        "error"
      ],
      "type": "string"
    },
    "mode": {
```
```

```
        "title": "Mode",
        "description": "Specifies the mode of
        the experiment. The mode options
        include 'one' (selecting a random Pod)
        , 'all' (selecting all eligible Pods),
         'fixed' (selecting a specified number
         of eligible Pods), 'fixed-percent' (
        selecting a specified percentage of
        Pods from the eligible Pods), and '
        random-max-percent' (selecting the
        maximum percentage of Pods from the
        eligible Pods)",
        "example": "one",
        "enum": [
          "one",
          "all",
          "fixed",
          "fixed-percent",
          "random-max-percent"
        ],
        "type": "string"
      },
      "value": {
        "title": "Value",
        "description": "Provides parameters
        for the mode configuration, depending
        on mode. For example, when mode is set
         to fixed-percent, value specifies the
         percentage of Pods.",
        "example": "1",
        "type": "string"
      },
      "patterns": {
        "title": "Patterns",
        "description": "Selects a domain
        template that matches faults. The
        fault is applyed to these domains.
        Placeholder ? and wildcard * are
        supported, but the wildcard in
        patterns configuration must be at the
        end of string. For example, chaos-mes
        *.org. is an invalid configuration.
        When patterns is not configured,
        faults are injected for all domains.",
        "example": "google.com, chaos-mesh.org
        , github.com",
        "type": "array",
        "items": {
          "type": "string"
        }
      },
      "selector": {
        "title": "Selector",
        "description": "Specifies the target
        Pod.",
        "example": null,
        "allOf": [
          {
            "$ref": "#/definitions/Selectors"
          }
        ]
      }
    },
    "required": [
      "selector"
    ],
    "definitions": {
      "SetBasedRequirements": {
        "title": "SetBasedRequirements",
        "type": "object",
        "properties": {
          "key": {
            "title": "Key",
            "description": "Label key",
            "type": "string"
          },
          "operator": {
            "title": "Operator",
            "description": "Select an operator
            .",
            "enum": [
              "In",
              "NotIn",
              "Exists",
              "DoesNotExist"
            ],
            "type": "string"
          },
          "values": {
            "title": "Values",
            "description": "Label values. The
            values set must be non-empty in
            the case of In and NotIn.",
            "type": "array",
            "items": {
              "type": "string"
            }
          }
        },
        "required": [
          "key",
          "operator",
          "values"
        ]
      },
      "Selectors": {
        "title": "Selectors",
        "type": "object",
        "properties": {
          "namespaces": {
            "title": "Namespaces",
            "description": "Specifies the
            namespace of the experiment's
            target Pod. If this selector is
            None, Chaos Mesh will set it to
            the namespace of the current Chaos
             experiment.",
            "type": "array",
            "items": {
              "type": "string"
            }
          },
          "labelSelectors": {
            "title": "Labelselectors",
            "description": "Specifies the
            label-key/value pairs that the
            experiment's target Pod must have.
             If multiple labels are specified,
             the experiment target must have
            all the labels specified by this
            selector.",
            "type": "object",
            "additionalProperties": {
              "type": "string"
            }
          },
          "expressionSelectors": {
            "title": "Expressionselectors",
            "description": "Specifies a set of
             expressions that define the label
            's rules to specifiy the
            experiment's target Pod.",
            "example": [
              {
                "key": "tier",
                "operator": "In",
                "values": [
                  "cache"
                ]
              },
              {
                "key": "environment",
                "operator": "NotIn",
                "values": [
                  "dev"
                ]
              }
            ],
            "type": "array",
            "items": {
              "$ref": "#/definitions/
              SetBasedRequirements"
            }
          },
          "annotationSelectors": {
            "title": "Annotationselectors",
            "description": "Specifies the
            annotation-key/value pairs that
            the experiment's target Pod must
            have. If multiple annotations are
```

```
          specified, the experiment target
          must have all annotations
          specified by this selector.",
          "type": "object",
          "additionalProperties": {
            "type": "string"
          }
        },
        "fieldSelectors": {
          "title": "Fieldselectors",
          "description": "Specifies the
          field-key/value pairs of the
          experiment's target Pod. If
          multiple fields are specified, the
           experiment target must have all
          fields set by this selector.",
          "example": {
            "metadata.name": "my-pod",
            "metadata.namespace": "dafault"
          },
          "type": "object",
          "additionalProperties": {
            "type": "string"
          }
        },
        "podPhaseSelectors": {
          "title": "Podphaseselectors",
          "description": "Specifies the
          phase of the experiment's target
          Pod. If this selector is None, the
           target Pod's phase is not limited
          .",
          "type": "array",
          "items": {
            "enum": [
              "Pending",
              "Running",
              "Succeeded",
              "Failed",
              "Unknown"
            ],
            "type": "string"
          }
        },
        "nodeSelectors": {
          "title": "Nodeselectors",
          "description": "Specifies the node
          -label-key/value pairs to which
          the experiment's target Pod
          belongs.",
          "type": "object",
          "additionalProperties": {
            "type": "string"
          }
        },
        "nodes": {
          "title": "Nodes",
          "description": "Specifies the node
           to which the experiment's target
          Pod belongs. The target Pod can
          only belong to one node in the
          configured node list. If multiple
          node labels are specified, the
          node to which the experiment's
          target Pod belongs must have all
          labels specified by this selector
          .",
          "type": "array",
          "items": {
            "type": "string"
          }
        },
        "pods": {
          "title": "Pods",
          "description": "Specifies the
          namespaces and list of the
          experiment's target Pods. If you
          have specified this selector,
          Chaos Mesh ignores other
          configured selectors.",
          "example": {
            "default": [
              "pod-0",
              "pod-2"
            ]
```

```
          },
          "type": "object",
          "additionalProperties": {
            "type": "array",
            "items": {
              "type": "string"
            }
          }
        }
      }
    }
  }
```

Dynamic instruction 4: **detailed_param_instructions** for **HTTPChaos**

```
he output should be formatted as a JSON
instance that conforms to the JSON schema
below.

As an example, for the schema {"properties":
 {"foo": {"title": "Foo", "description": "a
list of strings", "type": "array", "items":
{"type": "string"}}}, "required": ["foo"]}
the object {"foo": ["bar", "baz"]} is a well
-formatted instance of the schema. The
object {"properties": {"foo": ["bar", "baz
"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "mode": {
      "title": "Mode",
      "description": "Specifies the mode of
      the experiment. The mode options
      include one (selecting a random Pod),
      all (selecting all eligible Pods),
      fixed (selecting a specified number of
       eligible Pods), fixed-percent (
      selecting a specified percentage of
      Pods from the eligible Pods), and
      random-max-percent (selecting the
      maximum percentage of Pods from the
      eligible Pods)",
      "example": "one",
      "enum": [
        "one",
        "all",
        "fixed",
        "fixed-percent",
        "random-max-percent"
      ],
      "type": "string"
    },
    "value": {
      "title": "Value",
      "description": "Provides parameters
      for the mode configuration, depending
      on mode. For example, when mode is set
       to fixed-percent, value specifies the
       percentage of Pods.",
      "example": "1",
      "type": "string"
    },
    "target": {
      "title": "Target",
      "description": "Specifies whether the
      target of fault injection is Request
      or Response. The target-related fields
       (replace.path, replace.method,
      replace.queries, patch.queries) should
       be configured at the same time.",
      "example": "Request",
      "enum": [
        "Request",
        "Response"
```
```

```
      ],
      "type": "string"
    },
    "port": {
      "title": "Port",
      "description": "The TCP port that the
      target service listens on.",
      "example": 80,
      "type": "integer"
    },
    "code": {
      "title": "Code",
      "description": "Specifies the status
      code responded by target. If not
      specified, the fault takes effect for
      all status codes by default. This
      configuration is effective only when
      the 'target' is set to 'Response'",
      "example": 200,
      "type": "integer"
    },
    "path": {
      "title": "Path",
      "description": "Specify the URI path
      of the target request. Supports
      Matching wildcards. If not specified,
      the fault takes effect on all paths by
       default.",
      "example": "/api/*",
      "type": "string"
    },
    "method": {
      "title": "Method",
      "description": "Specify the HTTP
      method of the target request method.
      If not specified, the fault takes
      effect for all methods by default.",
      "example": "GET",
      "type": "string"
    },
    "request_headers": {
      "title": "Request Headers",
      "description": "Matches request
      headers to target.",
      "example": {
        "Content-Type": "application/json"
      },
      "type": "object",
      "additionalProperties": {
        "type": "string"
      }
    },
    "abort": {
      "title": "Abort",
      "description": "Abort fault. Indicates
       whether to inject the fault that
      interrupts the connection.",
      "default": false,
      "example": true,
      "type": "boolean"
    },
    "delay": {
      "title": "Delay",
      "description": "Deplay fault.
      Specifies the time for a latency fault
      .",
      "default": "0",
      "example": "10s",
      "type": "string"
    },
    "replace": {
      "title": "Replace",
      "description": "Replace fault.
      Specifies replaced contents.",
      "allOf": [
        {
          "$ref": "#/definitions/Replace"
        }
      ]
    },
    "patch": {
      "title": "Patch",
      "description": "Patch fault. Specifies
       patch contents.",
      "allOf": [
        {
          "$ref": "#/definitions/Patch"
        }
      ]
    }
  },
  "required": [
    "mode",
    "target",
    "port"
  ],
  "definitions": {
    "Replace": {
      "title": "Replace",
      "type": "object",
      "properties": {
        "headers": {
          "title": "Headers",
          "description": "Specifies the key
          pair used to replace the request
          headers or response headers.",
          "example": {
            "Content-Type": "application/xml
            "
          },
          "type": "object",
          "additionalProperties": {
            "type": "string"
          }
        },
        "body": {
          "title": "Body",
          "description": "Specifies request
          body or response body to replace
          the fault (Base64 encoded).",
          "example": "eyJmb28iOiAiYmFyIn0K",
          "type": "string"
        },
        "path": {
          "title": "Path",
          "description": "Specifies the URI
          path used to replace content.",
          "example": "/api/v2",
          "type": "string"
        },
        "method": {
          "title": "Method",
          "description": "Specifies the
          replaced content of the HTTP
          request method.",
          "example": "DELETE",
          "type": "string"
        },
        "queries": {
          "title": "Queries",
          "description": "Specifies the
          replaced key pair of the URI query
          .",
          "type": "array",
          "items": {
            "type": "array",
            "items": {
              "type": "string"
            }
          }
        },
        "code": {
          "title": "Code",
          "description": "Specifies the
          replaced content of the response
          status code. This configuration is
           effective only when the 'target'
          is set to 'Response'.",
          "example": 404,
          "type": "integer"
        }
      }
    },
    "PatchBody": {
      "title": "PatchBody",
      "type": "object",
      "properties": {
        "type": {
          "title": "Type",
          "description": "Specifies the type
```

```
            of patch faults of the request
           body or response body. Currently,
           it only supports JSON.",
           "example": "JSON",
           "type": "string"
         },
         "value": {
           "title": "Value",
           "description": "Specifies the
           fault of the request body or
           response body with patch faults.",
           "example": "{'foo': 'bar'}",
           "type": "string"
         }
       }
     },
     "Patch": {
       "title": "Patch",
       "type": "object",
       "properties": {
         "headers": {
           "title": "Headers",
           "description": "Specifies the
           attached key pair of the request
           headers or response headers with
           patch faults.",
           "example": [
             [
               "Set-Cookie",
               "one cookie"
             ]
           ],
           "type": "array",
           "items": {
             "type": "array",
             "items": {
               "type": "string"
             }
           }
         },
         "body": {
           "title": "Body",
           "description": "Patch body.",
           "allOf": [
             {
               "$ref": "#/definitions/
               PatchBody"
             }
           ]
         },
         "queries": {
           "title": "Queries",
           "description": "Specifies the
           attached key pair of the URI query
            with patch faults.",
           "example": [
             [
               "foo",
               "bar"
             ]
           ],
           "type": "array",
           "items": {
             "type": "array",
             "items": {
               "type": "string"
             }
           }
         }
       }
     }
   }
}
```

Dynamic instruction 5: **detailed_param_instructions** for **StressChaos**

```
he output should be formatted as a JSON
instance that conforms to the JSON schema
below.

As an example, for the schema {"properties":
 {"foo": {"title": "Foo", "description": "a
list of strings", "type": "array", "items":
{"type": "string"}}}, "required": ["foo"]}
the object {"foo": ["bar", "baz"]} is a well
-formatted instance of the schema. The
object {"properties": {"foo": ["bar", "baz
"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "mode": {
      "title": "Mode",
      "description": "Specifies the mode of
      the experiment. The mode options
      include 'one' (selecting a random Pod)
      , 'all' (selecting all eligible Pods),
       'fixed' (selecting a specified number
       of eligible Pods), 'fixed-percent' (
      selecting a specified percentage of
      Pods from the eligible Pods), and '
      random-max-percent' (selecting the
      maximum percentage of Pods from the
      eligible Pods)",
      "example": "one",
      "enum": [
        "one",
        "all",
        "fixed",
        "fixed-percent",
        "random-max-percent"
      ],
      "type": "string"
    },
    "value": {
      "title": "Value",
      "description": "Provides parameters
      for the mode configuration, depending
      on mode.For example, when mode is set
      to fixed-percent, value specifies the
      percentage of Pods.",
      "example": "1",
      "type": "string"
    },
    "stressors": {
      "title": "Stressors",
      "description": "Specifies the stress
      of CPU or memory",
      "dafault": null,
      "allOf": [
        {
          "$ref": "#/definitions/Stressors"
        }
      ]
    },
    "stressngStressors": {
      "title": "Stressngstressors",
      "description": "Specifies the stres-ng
       parameter to reach richer stress
      injection",
      "example": "--clone 2",
      "type": "string"
    },
    "containerNames": {
      "title": "Containernames",
      "description": "Specifies the name of
      the container into which the fault is
      injected.",
      "example": [
        "nginx"
      ],
      "type": "array",
      "items": {
        "type": "string"
```
```

```
      }
    },
    "selector": {
      "title": "Selector",
      "description": "Specifies the target
      Pod.",
      "allOf": [
        {
          "$ref": "#/definitions/Selectors"
        }
      ]
    }
  },
  "required": [
    "mode",
    "selector"
  ],
  "definitions": {
    "MemoryStressor": {
      "title": "MemoryStressor",
      "type": "object",
      "properties": {
        "workers": {
          "title": "Workers",
          "description": "Specifies the
          number of threads that apply
          memory stress",
          "example": 1,
          "type": "integer"
        },
        "size": {
          "title": "Size",
          "description": "Specifies the
          memory size to be occupied or a
          percentage of the total memory
          size. The final sum of the
          occupied memory size is size.",
          "example": "256MB",
          "type": "string"
        },
        "oomScoreAdj": {
          "title": "Oomscoreadj",
          "description": "Specifies the
          oom_score_adj of the stress
          process.",
          "example": -1000,
          "type": "integer"
        }
      }
    },
    "CPUStressor": {
      "title": "CPUStressor",
      "type": "object",
      "properties": {
        "workers": {
          "title": "Workers",
          "description": "Specifies the
          number of threads that apply CPU
          stress",
          "example": 1,
          "type": "integer"
        },
        "load": {
          "title": "Load",
          "description": "Specifies the
          percentage of CPU occupied. 0
          means that no additional CPU is
          added, and 100 refers to full load
          . The final sum of CPU load is
          workers * load.",
          "example": 50,
          "type": "integer"
        }
      }
    },
    "Stressors": {
      "title": "Stressors",
      "type": "object",
      "properties": {
        "memory": {
          "title": "Memory",
          "description": "Specifies the
          memory stress",
          "allOf": [
            {
              "$ref": "#/definitions/
              MemoryStressor"
            }
          ]
        },
        "cpu": {
          "title": "Cpu",
          "description": "Specifies the CPU
          stress",
          "allOf": [
            {
              "$ref": "#/definitions/
              CPUStressor"
            }
          ]
        }
      }
    },
    "SetBasedRequirements": {
      "title": "SetBasedRequirements",
      "type": "object",
      "properties": {
        "key": {
          "title": "Key",
          "description": "Label key",
          "type": "string"
        },
        "operator": {
          "title": "Operator",
          "description": "Select an operator
          .",
          "enum": [
            "In",
            "NotIn",
            "Exists",
            "DoesNotExist"
          ],
          "type": "string"
        },
        "values": {
          "title": "Values",
          "description": "Label values. The
          values set must be non-empty in
          the case of In and NotIn.",
          "type": "array",
          "items": {
            "type": "string"
          }
        }
      },
      "required": [
        "key",
        "operator",
        "values"
      ]
    },
    "Selectors": {
      "title": "Selectors",
      "type": "object",
      "properties": {
        "namespaces": {
          "title": "Namespaces",
          "description": "Specifies the
          namespace of the experiment's
          target Pod. If this selector is
          None, Chaos Mesh will set it to
          the namespace of the current Chaos
           experiment.",
          "type": "array",
          "items": {
            "type": "string"
          }
        },
        "labelSelectors": {
          "title": "Labelselectors",
          "description": "Specifies the
          label-key/value pairs that the
          experiment's target Pod must have.
           If multiple labels are specified,
           the experiment target must have
          all the labels specified by this
          selector.",
          "type": "object",
          "additionalProperties": {
            "type": "string"
```

```
    }
  },
  "expressionSelectors": {
    "title": "Expressionselectors",
    "description": "Specifies a set of
     expressions that define the label
    's rules to specifiy the
    experiment's target Pod.",
    "example": [
      {
        "key": "tier",
        "operator": "In",
        "values": [
          "cache"
        ]
      },
      {
        "key": "environment",
        "operator": "NotIn",
        "values": [
          "dev"
        ]
      }
    ],
    "type": "array",
    "items": {
      "$ref": "#/definitions/
      SetBasedRequirements"
    }
  },
  "annotationSelectors": {
    "title": "Annotationselectors",
    "description": "Specifies the
    annotation-key/value pairs that
    the experiment's target Pod must
    have. If multiple annotations are
    specified, the experiment target
    must have all annotations
    specified by this selector.",
    "type": "object",
    "additionalProperties": {
      "type": "string"
    }
  },
  "fieldSelectors": {
    "title": "Fieldselectors",
    "description": "Specifies the
    field-key/value pairs of the
    experiment's target Pod. If
    multiple fields are specified, the
     experiment target must have all
    fields set by this selector.",
    "example": {
      "metadata.name": "my-pod",
      "metadata.namespace": "dafault"
    },
    "type": "object",
    "additionalProperties": {
      "type": "string"
    }
  },
  "podPhaseSelectors": {
    "title": "Podphaseselectors",
    "description": "Specifies the
    phase of the experiment's target
    Pod. If this selector is None, the
     target Pod's phase is not limited
    .",
    "type": "array",
    "items": {
      "enum": [
        "Pending",
        "Running",
        "Succeeded",
        "Failed",
        "Unknown"
      ],
      "type": "string"
    }
  },
  "nodeSelectors": {
    "title": "Nodeselectors",
    "description": "Specifies the node
    -label-key/value pairs to which
    the experiment's target Pod
    belongs.",
    "type": "object",
    "additionalProperties": {
      "type": "string"
    }
  },
  "nodes": {
    "title": "Nodes",
    "description": "Specifies the node
     to which the experiment's target
    Pod belongs. The target Pod can
    only belong to one node in the
    configured node list. If multiple
    node labels are specified, the
    node to which the experiment's
    target Pod belongs must have all
    labels specified by this selector
    .",
    "type": "array",
    "items": {
      "type": "string"
    }
  },
  "pods": {
    "title": "Pods",
    "description": "Specifies the
    namespaces and list of the
    experiment's target Pods. If you
    have specified this selector,
    Chaos Mesh ignores other
    configured selectors.",
    "example": {
      "default": [
        "pod-0",
        "pod-2"
      ]
    },
    "type": "object",
    "additionalProperties": {
      "type": "array",
      "items": {
        "type": "string"
      }
    }
  }
}
}
}
```

Dynamic instruction 6: **detailed_param_instructions** for **IOChaos**

```
he output should be formatted as a JSON
instance that conforms to the JSON schema
below.

As an example, for the schema {"properties":
 {"foo": {"title": "Foo", "description": "a
list of strings", "type": "array", "items":
{"type": "string"}}}, "required": ["foo"]}
the object {"foo": ["bar", "baz"]} is a well
-formatted instance of the schema. The
object {"properties": {"foo": ["bar", "baz
"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "action": {
      "title": "Action",
      "description": "Indicates the specific
       type of faults. Only latency, fault,
      attrOverride, and mistake are
      supported.",
      "example": "latency",
      "enum": [
        "latency",
```
```

```
        "fault",
        "attrOverride",
        "mistake"
      ],
      "type": "string"
    },
    "mode": {
      "title": "Mode",
      "description": "Specifies the mode of
      the experiment. The mode options
      include one (selecting a random Pod),
      all (selecting all eligible Pods),
      fixed (selecting a specified number of
       eligible Pods), fixed-percent (
      selecting a specified percentage of
      Pods from the eligible Pods), and
      random-max-percent (selecting the
      maximum percentage of Pods from the
      eligible Pods)",
      "example": "one",
      "enum": [
        "one",
        "all",
        "fixed",
        "fixed-percent",
        "random-max-percent"
      ],
      "type": "string"
    },
    "selector": {
      "title": "Selector",
      "description": "Specifies the target
      Pod.",
      "allOf": [
        {
          "$ref": "#/definitions/Selectors"
        }
      ]
    },
    "value": {
      "title": "Value",
      "description": "Provides parameters
      for the mode configuration, depending
      on mode. For example, when mode is set
       to fixed-percent, value specifies the
       percentage of Pods.",
      "example": "1",
      "type": "string"
    },
    "volumePath": {
      "title": "Volumepath",
      "description": "The mount point of
      volume in the target container. Must
      be the root directory of the mount.",
      "example": "/var/run/etcd",
      "type": "string"
    },
    "path": {
      "title": "Path",
      "description": "The valid range of
      fault injections, either a wildcard or
       a single file. If not specified, the
      fault is valid for all files by
      default",
      "example": "/var/run/etcd/*/",
      "type": "string"
    },
    "methods": {
      "title": "Methods",
      "description": "Type of the file
      system call that requires injecting
      fault. Supported method types: ['
      lookup', 'forget', 'getattr', 'setattr
      ', 'readlink', 'mknod', 'mkdir', '
      unlink', 'rmdir', 'symlink', 'rename',
       'link', 'open', 'read', 'write', '
      flush', 'release', 'fsync', 'opendir',
       'readdir', 'releasedir', 'fsyncdir',
      'statfs', 'setxattr', 'getxattr', '
      listxattr', 'removexattr', 'access', '
      create', 'getlk', 'setlk', 'bmap'].
      All Types by default.",
      "example": [
        "READ"
      ],
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "percent": {
      "title": "Percent",
      "description": "Probability of failure
       per operation, in %.",
      "default": 100,
      "example": 100,
      "type": "integer"
    },
    "containerNames": {
      "title": "Containernames",
      "description": "Specifies the name of
      the container into which the fault is
      injected.",
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "deplay": {
      "title": "Deplay",
      "description": "Specify when the '
      action' is set to 'latency'. Specific
      delay time.",
      "type": "string"
    },
    "errno": {
      "title": "Errno",
      "description": "Specify when the '
      action' is set to 'fault'. Returned
      error number: 1: Operation not
      permitted, 2: No such file or
      directory, 5: I/O error, 6: No such
      device or address, 12: Out of memory,
      16: Device or resource busy, 17: File
      exists, 20: Not a directory, 22:
      Invalid argument, 24: Too many open
      files, 28: No space left on device",
      "type": "integer"
    },
    "attr": {
      "title": "Attr",
      "description": "Specify when the '
      action' is set to 'attrOverride'.
      Specific property override rules.",
      "allOf": [
        {
          "$ref": "#/definitions/
          AttrOverrideSpec"
        }
      ]
    },
    "mistake": {
      "title": "Mistake",
      "description": "Specify when the '
      action' is set to 'mistake'. Specific
      error rules.",
      "allOf": [
        {
          "$ref": "#/definitions/MistakeSpec
          "
        }
      ]
    }
  },
  "required": [
    "action",
    "mode",
    "volumePath",
    "attr",
    "mistake"
  ],
  "definitions": {
    "SetBasedRequirements": {
      "title": "SetBasedRequirements",
      "type": "object",
      "properties": {
        "key": {
          "title": "Key",
          "description": "Label key",
          "type": "string"
```

```
        },
        "operator": {
          "title": "Operator",
          "description": "Select an operator
          .",
          "enum": [
            "In",
            "NotIn",
            "Exists",
            "DoesNotExist"
          ],
          "type": "string"
        },
        "values": {
          "title": "Values",
          "description": "Label values. The
          values set must be non-empty in
          the case of In and NotIn.",
          "type": "array",
          "items": {
            "type": "string"
          }
        }
      },
      "required": [
        "key",
        "operator",
        "values"
      ]
    },
    "Selectors": {
      "title": "Selectors",
      "type": "object",
      "properties": {
        "namespaces": {
          "title": "Namespaces",
          "description": "Specifies the
          namespace of the experiment's
          target Pod. If this selector is
          None, Chaos Mesh will set it to
          the namespace of the current Chaos
           experiment.",
          "type": "array",
          "items": {
            "type": "string"
          }
        },
        "labelSelectors": {
          "title": "Labelselectors",
          "description": "Specifies the
          label-key/value pairs that the
          experiment's target Pod must have.
           If multiple labels are specified,
           the experiment target must have
          all the labels specified by this
          selector.",
          "type": "object",
          "additionalProperties": {
            "type": "string"
          }
        },
        "expressionSelectors": {
          "title": "Expressionselectors",
          "description": "Specifies a set of
           expressions that define the label
          's rules to specifiy the
          experiment's target Pod.",
          "example": [
            {
              "key": "tier",
              "operator": "In",
              "values": [
                "cache"
              ]
            },
            {
              "key": "environment",
              "operator": "NotIn",
              "values": [
                "dev"
              ]
            }
          ],
          "type": "array",
          "items": {
            "$ref": "#/definitions/
            SetBasedRequirements"
          }
        },
        "annotationSelectors": {
          "title": "Annotationselectors",
          "description": "Specifies the
          annotation-key/value pairs that
          the experiment's target Pod must
          have. If multiple annotations are
          specified, the experiment target
          must have all annotations
          specified by this selector.",
          "type": "object",
          "additionalProperties": {
            "type": "string"
          }
        },
        "fieldSelectors": {
          "title": "Fieldselectors",
          "description": "Specifies the
          field-key/value pairs of the
          experiment's target Pod. If
          multiple fields are specified, the
           experiment target must have all
          fields set by this selector.",
          "example": {
            "metadata.name": "my-pod",
            "metadata.namespace": "dafault"
          },
          "type": "object",
          "additionalProperties": {
            "type": "string"
          }
        },
        "podPhaseSelectors": {
          "title": "Podphaseselectors",
          "description": "Specifies the
          phase of the experiment's target
          Pod. If this selector is None, the
           target Pod's phase is not limited
          .",
          "type": "array",
          "items": {
            "enum": [
              "Pending",
              "Running",
              "Succeeded",
              "Failed",
              "Unknown"
            ],
            "type": "string"
          }
        },
        "nodeSelectors": {
          "title": "Nodeselectors",
          "description": "Specifies the node
          -label-key/value pairs to which
          the experiment's target Pod
          belongs.",
          "type": "object",
          "additionalProperties": {
            "type": "string"
          }
        },
        "nodes": {
          "title": "Nodes",
          "description": "Specifies the node
           to which the experiment's target
          Pod belongs. The target Pod can
          only belong to one node in the
          configured node list. If multiple
          node labels are specified, the
          node to which the experiment's
          target Pod belongs must have all
          labels specified by this selector
          .",
          "type": "array",
          "items": {
            "type": "string"
          }
        },
        "pods": {
          "title": "Pods",
          "description": "Specifies the
```

```
        namespaces and list of the
        experiment's target Pods. If you
        have specified this selector,
        Chaos Mesh ignores other
        configured selectors.",
        "example": {
          "default": [
            "pod-0",
            "pod-2"
          ]
        },
        "type": "object",
        "additionalProperties": {
          "type": "array",
          "items": {
            "type": "string"
          }
        }
      }
    }
  },
  "TimeSpec": {
    "title": "TimeSpec",
    "type": "object",
    "properties": {
      "sec": {
        "title": "Sec",
        "description": "Timestamp in
        seconds. Specify either sec or
        nsec.",
        "type": "integer"
      },
      "nsec": {
        "title": "Nsec",
        "description": "Timestamp in
        nanoseconds. Specify either sec or
         nsec.",
        "type": "integer"
      }
    }
  },
  "AttrOverrideSpec": {
    "title": "AttrOverrideSpec",
    "type": "object",
    "properties": {
      "ino": {
        "title": "Ino",
        "description": "ino number",
        "type": "integer"
      },
      "size": {
        "title": "Size",
        "description": "File size",
        "type": "integer"
      },
      "blocks": {
        "title": "Blocks",
        "description": "Number of blocks
        that the file uses",
        "type": "integer"
      },
      "atime": {
        "title": "Atime",
        "description": "Last access time",
        "allOf": [
          {
            "$ref": "#/definitions/
            TimeSpec"
          }
        ]
      },
      "mtime": {
        "title": "Mtime",
        "description": "Last modified time
        ",
        "allOf": [
          {
            "$ref": "#/definitions/
            TimeSpec"
          }
        ]
      },
      "ctime": {
        "title": "Ctime",
        "description": "Last status change
         time",
        "allOf": [
          {
            "$ref": "#/definitions/
            TimeSpec"
          }
        ]
      },
      "kind": {
        "title": "Kind",
        "description": "File type, see
        fuser::FileType",
        "type": "string"
      },
      "perm": {
        "title": "Perm",
        "description": "File permissions
        in decimal",
        "type": "integer"
      },
      "nlink": {
        "title": "Nlink",
        "description": "Number of hard
        links",
        "type": "integer"
      },
      "uid": {
        "title": "Uid",
        "description": "User ID of the
        owner",
        "type": "integer"
      },
      "gid": {
        "title": "Gid",
        "description": "Group ID of the
        owner",
        "type": "integer"
      },
      "rdev": {
        "title": "Rdev",
        "description": "Device ID",
        "type": "integer"
      }
    }
  },
  "MistakeSpec": {
    "title": "MistakeSpec",
    "type": "object",
    "properties": {
      "filling": {
        "title": "Filling",
        "description": "The wrong data to
        be filled. Only zero (fill 0) or
        random (fill random bytes) are
        supported.",
        "type": "string"
      },
      "maxOccurrences": {
        "title": "Maxoccurrences",
        "description": "Maximum number of
        errors in each operation.",
        "example": 1,
        "type": "integer"
      },
      "maxLength": {
        "title": "Maxlength",
        "description": "Maximum length of
        each error (in bytes).",
        "example": 1,
        "type": "integer"
      }
    },
    "required": [
      "filling",
      "maxOccurrences",
      "maxLength"
    ]
  }
}
}
```

Dynamic instruction 7: **detailed_param_instructions** for **TimeChaos**

```
The output should be formatted as a JSON
instance that conforms to the JSON schema
below.

As an example, for the schema {"properties":
 {"foo": {"title": "Foo", "description": "a
list of strings", "type": "array", "items":
{"type": "string"}}}, "required": ["foo"]}
the object {"foo": ["bar", "baz"]} is a well
-formatted instance of the schema. The
object {"properties": {"foo": ["bar", "baz
"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "timeOffset": {
      "title": "Timeoffset",
      "description": "Specifies the length
      of time offset.",
      "example": "-5m",
      "type": "string"
    },
    "clockIds": {
      "title": "Clockids",
      "description": "Specifies the ID of
      clock that will be offset. See the
      clock_gettime documentation for
      details.",
      "default": [
        "CLOCK_REALTIME"
      ],
      "example": [
        "CLOCK_REALTIME",
        "CLOCK_MONOTONIC"
      ],
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "mode": {
      "title": "Mode",
      "description": "Specifies the mode of
      the experiment. The mode options
      include 'one' (selecting a random Pod)
      , 'all' (selecting all eligible Pods),
       'fixed' (selecting a specified number
       of eligible Pods), 'fixed-percent' (
      selecting a specified percentage of
      Pods from the eligible Pods), and '
      random-max-percent' (selecting the
      maximum percentage of Pods from the
      eligible Pods)",
      "example": "one",
      "enum": [
        "one",
        "all",
        "fixed",
        "fixed-percent",
        "random-max-percent"
      ],
      "type": "string"
    },
    "value": {
      "title": "Value",
      "description": "Provides parameters
      for the mode configuration, depending
      on mode. For example, when mode is set
       to fixed-percent, value specifies the
       percentage of Pods.",
      "example": "1",
      "type": "string"
    },
    "containerNames": {
      "title": "Containernames",
      "description": "Specifies the name of
      the container into which the fault is
      injected.",
      "example": [
        "nginx"
      ],
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "selector": {
      "title": "Selector",
      "description": "Specifies the target
      Pod.",
      "example": null,
      "allOf": [
        {
          "$ref": "#/definitions/Selectors"
        }
      ]
    }
  },
  "required": [
    "timeOffset",
    "mode",
    "selector"
  ],
  "definitions": {
    "SetBasedRequirements": {
      "title": "SetBasedRequirements",
      "type": "object",
      "properties": {
        "key": {
          "title": "Key",
          "description": "Label key",
          "type": "string"
        },
        "operator": {
          "title": "Operator",
          "description": "Select an operator
          .",
          "enum": [
            "In",
            "NotIn",
            "Exists",
            "DoesNotExist"
          ],
          "type": "string"
        },
        "values": {
          "title": "Values",
          "description": "Label values. The
          values set must be non-empty in
          the case of In and NotIn.",
          "type": "array",
          "items": {
            "type": "string"
          }
        }
      },
      "required": [
        "key",
        "operator",
        "values"
      ]
    },
    "Selectors": {
      "title": "Selectors",
      "type": "object",
      "properties": {
        "namespaces": {
          "title": "Namespaces",
          "description": "Specifies the
          namespace of the experiment's
          target Pod. If this selector is
          None, Chaos Mesh will set it to
          the namespace of the current Chaos
           experiment.",
          "type": "array",
          "items": {
            "type": "string"
          }
        },
        "labelSelectors": {
          "title": "Labelselectors",
          "description": "Specifies the
          label-key/value pairs that the
```
```

```
          experiment's target Pod must have.
           If multiple labels are specified,
           the experiment target must have
          all the labels specified by this
          selector.",
          "type": "object",
          "additionalProperties": {
            "type": "string"
          }
        },
        "expressionSelectors": {
          "title": "Expressionselectors",
          "description": "Specifies a set of
           expressions that define the label
          's rules to specifiy the
          experiment's target Pod.",
          "example": [
            {
              "key": "tier",
              "operator": "In",
              "values": [
                "cache"
              ]
            },
            {
              "key": "environment",
              "operator": "NotIn",
              "values": [
                "dev"
              ]
            }
          ],
          "type": "array",
          "items": {
            "$ref": "#/definitions/
            SetBasedRequirements"
          }
        },
        "annotationSelectors": {
          "title": "Annotationselectors",
          "description": "Specifies the
          annotation-key/value pairs that
          the experiment's target Pod must
          have. If multiple annotations are
          specified, the experiment target
          must have all annotations
          specified by this selector.",
          "type": "object",
          "additionalProperties": {
            "type": "string"
          }
        },
        "fieldSelectors": {
          "title": "Fieldselectors",
          "description": "Specifies the
          field-key/value pairs of the
          experiment's target Pod. If
          multiple fields are specified, the
           experiment target must have all
          fields set by this selector.",
          "example": {
            "metadata.name": "my-pod",
            "metadata.namespace": "dafault"
          },
          "type": "object",
          "additionalProperties": {
            "type": "string"
          }
        },
        "podPhaseSelectors": {
          "title": "Podphaseselectors",
          "description": "Specifies the
          phase of the experiment's target
          Pod. If this selector is None, the
           target Pod's phase is not limited
          .",
          "type": "array",
          "items": {
            "enum": [
              "Pending",
              "Running",
              "Succeeded",
              "Failed",
              "Unknown"
            ],
              "type": "string"
            }
          },
          "nodeSelectors": {
            "title": "Nodeselectors",
            "description": "Specifies the node
            -label-key/value pairs to which
            the experiment's target Pod
            belongs.",
            "type": "object",
            "additionalProperties": {
              "type": "string"
            }
          },
          "nodes": {
            "title": "Nodes",
            "description": "Specifies the node
             to which the experiment's target
            Pod belongs. The target Pod can
            only belong to one node in the
            configured node list. If multiple
            node labels are specified, the
            node to which the experiment's
            target Pod belongs must have all
            labels specified by this selector
            .",
            "type": "array",
            "items": {
              "type": "string"
            }
          },
          "pods": {
            "title": "Pods",
            "description": "Specifies the
            namespaces and list of the
            experiment's target Pods. If you
            have specified this selector,
            Chaos Mesh ignores other
            configured selectors.",
            "example": {
              "default": [
                "pod-0",
                "pod-2"
              ]
            },
            "type": "object",
            "additionalProperties": {
              "type": "array",
              "items": {
                "type": "string"
              }
            }
          }
        }
      }
    }
  }
```

D.5.3 Experiment

Prompt 14: Agent # 2-0 for determining time schedule

```
System:
You are a helpful AI assistant for Chaos
Engineering.
Given k8s manifests that define a network
system, its steady states, and faults that
may affect the steady states in the system,
you will design a Chaos Engineering
experiment for them.
First, you will determine the time schedule
for the Chaos Engineering experiment.
Always keep the following rules:
- The experiment is divided into three
phases: pre-validation, fault-injection, and
 post-validation phases: pre-validation to
ensure that the system satisfies the steady
states fault injection; fault-injection to
observe the system's behavior during fault
```

injection; post-validation to ensure that the system has returned to its steady states after fault injection.
- The output should be formatted as a JSON instance that conforms to the JSON schema below.

As an example, for the schema {\"properties\": {\"foo\": {\"title\": \"Foo\", \"description\": \"a list of strings\", \"type\": \"array\", \"items\": {\"type\": \"string\"}}}, \"required\": [\"foo\"]}\nthe object {\"foo\": [\"bar\", \"baz\"]} is a well-formatted instance of the schema. The object {\"properties\": {\"foo\": [\"bar\", \"baz\"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "thought": {
      "title": "Thought",
      "describe": "Think about the total time and the reasonable time allocation for each phase that you are about to design, and explain your thought process in detail.",
      "type": "string"
    },
    "total_time": {
      "title": "Total Time",
      "description": "Total time of the entire chaos experiment. total_time should equal to the sum of pre_validation_time, fault_injection_time, and post_validation_time.",
      "example": "10m",
      "type": "string"
    },
    "pre_validation_time": {
      "title": "Pre Validation Time",
      "description": "Total time of validation before fault injection.",
      "example": "2m",
      "type": "string"
    },
    "fault_injection_time": {
      "title": "Fault Injection Time",
      "description": "Total time of fault injection.",
      "example": "6m",
      "type": "string"
    },
    "post_validation_time": {
      "title": "Post Validation Time",
      "description": "Total time of validation after fault injection.",
      "example": "2m",
      "type": "string"
    }
  },
  "required": [
    "thought",
    "total_time",
    "pre_validation_time",
    "fault_injection_time",
    "post_validation_time"
  ]
}
```

Human:
Here is the overview of my system:
{user_input2}

Steady states of my system:
{steady_states}

A fault scenario that may occur in my system and may affect the steady states:
{detailed_fault_scenario}

Please follow the instructions below regarding Chaos Engineering as necessary:
{ce_instructions}

Now, please plan a Chaos Engineering experiment to check the network system's resiliency that the steady states are remained during fault injection.

Example 17: **detailed_fault_scenario**

An assumed fault scenario is as follows:
- Event: Cyber Attack Simulation\n- Used Chaos Engineering tool: Chaos Mesh
- Faults to simulate the event: [[Fault(name='PodChaos', name_id=0, params={'action': 'pod-kill', 'mode': 'one', 'selector': {'namespaces': ['default'], 'labelSelectors': {'app': 'example'}}})], [Fault(name='NetworkChaos', name_id=0, params={'action': 'delay', 'mode': 'all', 'selector': {'namespaces': ['default'], 'labelSelectors': {'app': 'example'}}, 'direction': 'to', 'delay': {'latency': '100ms', 'jitter': '10ms'}})]]
- Description: Given the system's weaknesses, a cyber attack targeting the web server could be highly impactful. The Pod's restart policy set to 'Never' and the single Pod deployment without redundancy are critical vulnerabilities. If the Pod fails, it will not restart, leading to downtime, and the lack of redundancy means there is no backup to handle traffic. To simulate a cyber attack, we can inject faults that exploit these weaknesses. First, we will use PodChaos to simulate a Pod failure, which will test the system's ability to maintain the 'example-pod-running-state'. Since the Pod will not restart automatically, this will directly impact the steady state. Next, we will use NetworkChaos to simulate network latency, which will test the system's ability to maintain the 'example-service-http-response-state'. This sequence simulates a cyber attack where the Pod is targeted first, followed by network disruptions, revealing the system's vulnerabilities in handling such events.

Prompt 15: Agent # 2-1 for scheduling each experiment phase (pre-validation, failure-injection, and post-validation phases)

System:
You are a helpful AI assistant for Chaos Engineering.
Given k8s manifests that define a network system, its steady states, and faults that may affect the steady states in the system, you will design a Chaos Engineering experiment for them.
The experiment is divided into three phases: pre-validation, fault-injection, and post-validation phases: pre-validation to ensure that the system satisfies the steady states fault injection; fault-injection to observe the system's behavior during fault injection; post-validation to ensure that the system has returned to its steady states after fault injection.
Here, you will detail the **{phase_name}**.
Always keep the following rules:
- **{phase_planning_instructions}**

Human:

```
# Here is the overview of my system:
{user_input}

# Steady states of my system:
{steady_states}

# A fault scenario that may occur in my
system and may affect the steady states:
{detailed_fault_scenario}

# Please follow the instructions below
regarding Chaos Engineering as necessary:
{ce_instructions}

Now, please detail the {phase_name}. Note
that the phase's total time is {
phase_total_time}.
```

Example 18: `phase_name`

```
pre-validation phase
```

Example 19: `phase_total_time`

```
10s
```

Dynamic instruction 8: `phase_planning_instructions` for the pre-validation and post-validation phases

```
he output should be formatted as a JSON
instance that conforms to the JSON schema
below.

As an example, for the schema {\"properties
\": {\"foo\": {\"title\": \"Foo\", \"
description\": \"a list of strings\", \"type
\": \"array\", \"items\": {\"type\": \"
string\"}}}, \"required\": [\"foo\"]}\nthe
object {\"foo\": [\"bar\", \"baz\"]} is a
well-formatted instance of the schema. The
object {\"properties\": {\"foo\": [\"bar\",
\"baz\"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "thought": {
      "title": "Thought",
      "description": "Describe in detail the
       timeline for when each fault
      injection and each unit test (for
      verifying steady-state) will be
      executed. For example, explain which
      fault injections/unit tests will be
      executed simultaneously, and whether
      certain fault injections/unit tests
      will be executed at staggered timings.
       Additionally, explain the thought
      process that led you to this approach
      .",
      "type": "string"
    },
    "unit_tests": {
      "title": "Unit Tests",
      "description": "The list of unit test
      schedule.",
      "type": "array",
      "items": {
        "$ref": "#/definitions/UnitTest"
      }
    }
  },
  "required": [
    "thought",
    "unit_tests"
  ],
  "definitions": {
    "UnitTest": {
      "title": "UnitTest",
      "type": "object",
      "properties": {
        "name": {
          "title": "Name",
          "description": "Steady state name
          to be verified by a unit test.",
          "type": "string"
        },
        "grace_period": {
          "title": "Grace Period",
          "description": "Time elapsed from
          the start of the current phase to
          the beginning of the unit test.",
          "example": "0s",
          "type": "string"
        },
        "duration": {
          "title": "Duration",
          "description": "Duration of the
          unit test. (grace_period +
          duration) should not exceed the
          current phase's total time.",
          "example": "2m",
          "type": "string"
        }
      },
      "required": [
        "name",
        "grace_period",
        "duration"
      ]
    }
  }
}
```

### Dynamic instruction 9: `phase_planning_instructions` for the fault-injection phases

```
he output should be formatted as a JSON
instance that conforms to the JSON schema
below.

As an example, for the schema {\"properties
\": {\"foo\": {\"title\": \"Foo\", \"
description\": \"a list of strings\", \"type
\": \"array\", \"items\": {\"type\": \"
string\"}}}, \"required\": [\"foo\"]}\nthe
object {\"foo\": [\"bar\", \"baz\"]} is a
well-formatted instance of the schema. The
object {\"properties\": {\"foo\": [\"bar\",
\"baz\"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "thought": {
      "title": "Thought",
      "description": "Describe in detail the
       timeline for when each fault
      injection and each unit test (for
      verifying steady-state) will be
      executed. For example, explain which
      fault injections/unit tests will be
      executed simultaneously, and whether
      certain fault injections/unit tests
      will be executed at staggered timings.
       Additionally, explain the thought
```

```
      process that led you to this approach
      .",
      "type": "string"
    },
    "fault_injection": {
      "title": "Fault Injection",
      "description": "The list of fault
      injection schedules.",
      "type": "array",
      "items": {
        "$ref": "#/definitions/
        FaultInjection"
      }
    },
    "unit_tests": {
      "title": "Unit Tests",
      "description": "The list of unit test
      schedule.",
      "type": "array",
      "items": {
        "$ref": "#/definitions/UnitTest"
      }
    }
  },
  "required": [
    "thought",
    "fault_injection",
    "unit_tests"
  ],
  "definitions": {
    "FaultInjection": {
      "title": "FaultInjection",
      "type": "object",
      "properties": {
        "name": {
          "title": "Name",
          "description": "Select a fault
          type from [\"PodChaos\", \"
          NetworkChaos\", \"DNSChaos\", \"
          HTTPChaos\", \"StressChaos\", \"
          IOChaos\", \"TimeChaos\"]",
          "enum": [
            "PodChaos",
            "NetworkChaos",
            "DNSChaos",
            "HTTPChaos",
            "StressChaos",
            "IOChaos",
            "TimeChaos"
          ],
          "type": "string"
        },
        "name_id": {
          "title": "Name Id",
          "description": "An identifier to
          prevent name conflicts when the
          same Fault appears. Assign numbers
           starting from 0 in sequential
          order to prevent name conflicts.",
          "type": "integer"
        },
        "grace_period": {
          "title": "Grace Period",
          "description": "Time elapsed from
          the start of the current phase to
          the beginning of the fault
          injection.",
          "example": "0s",
          "type": "string"
        },
        "duration": {
          "title": "Duration",
          "description": "Duration of the
          unit test. (grace_period +
          duration) should not exceed the
          current phase's total time.",
          "example": "2m",
          "type": "string"
        }
      },
      "required": [
        "name",
        "name_id",
        "grace_period",
        "duration"
      ]
    },
    "UnitTest": {
      "title": "UnitTest",
      "type": "object",
      "properties": {
        "name": {
          "title": "Name",
          "description": "Steady state name
          to be verified by a unit test.",
          "type": "string"
        },
        "grace_period": {
          "title": "Grace Period",
          "description": "Time elapsed from
          the start of the current phase to
          the beginning of the unit test.",
          "example": "0s",
          "type": "string"
        },
        "duration": {
          "title": "Duration",
          "description": "Duration of the
          unit test. (grace_period +
          duration) should not exceed the
          current phase's total time.",
          "example": "2m",
          "type": "string"
        }
      },
      "required": [
        "name",
        "grace_period",
        "duration"
      ]
    }
  }
}
```

Prompt 16: Agent # 2-2 for summarizing the planned experiment

```
System:
You are a helpful AI assistant for Chaos
Engineering.
Given a Chaos-Engineering-experiment plan,
you will summarize it in detail according to
 the following rules:
- In each phase, describe in detail the
timeline for when each fault injection/unit
test (for verifying steady-state) will be
executed. For example, summarize which fault
 injections/unit tests will be executed
simultaneously, and whether certain fault
injections/unit tests will be executed at
staggered timings.
- Be sure to specify both each fault
injection/unit test and their corresponding
workflow names.
- When explaining the timeline, provide a
detailed description using specific values
for duration, grace period, etc. Rephrase
the specific values in a way that everyone
can easily understand.
- The meanings of each value are as follows:
  - Grace Period: Time elapsed from the
  start of the current phase to the
  beginning of the fault injection/unit test
  .
  - Duration: Duration of the fault
  injection/unit test. (grace_period +
  duration) should not exceed the
  corresponding phase's total time.
- Never output bullet points.
- The output should be formatted as a JSON
instance that conforms to the JSON schema
below.
As an example, for the schema {\"properties
\": {\"foo\": {\"title\": \"Foo\", \"
description\": \"a list of strings\", \"type
```

\": \"array\", \"items\": {\"type\": \"string\"}}}, \"required\": [\"foo\"]}\nthe object {\"foo\": [\"bar\", \"baz\"]} is a well-formatted instance of the schema. The object {\"properties\": {\"foo\": [\"bar\", \"baz\"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "summary": {
      "title": "Summary",
      "description": "The summary of the given Chaos-Engineering-experiment plan.",
      "type": "string"
    }
  },
  "required": [
    "summary"
  ]
}
```

Human:
Here is my Chaos-Engineering-experiment plan:
Time Schedule
{time_schedule_overview}

Pre-validation Phase
{pre_validation_overview}

Fault-injection Phase
{fault_injection_overview}

Post-validation phase
{post_validation_overview}

Please summarize the above plan.

Example 20: time_schedule_overview

Given the constraints of the experiment needing to be completed within 1 minute, we need to carefully allocate time to each phase to ensure that we can effectively validate the system's steady states before and after the fault injection, as well as observe the system's behavior during the fault injection. The pre-validation phase is crucial to establish a baseline that the system is in its expected steady state before any faults are introduced. The fault injection phase is where we introduce the chaos to observe how the system behaves under stress. Finally, the post-validation phase is necessary to ensure that the system returns to its steady state after the faults are removed. Given the short total time of 1 minute, a reasonable allocation could be 10 seconds for pre-validation, 40 seconds for fault injection, and 10 seconds for post-validation. This allocation allows us to have a brief but sufficient observation period for each phase, ensuring that we can gather meaningful insights from the experiment.

Example 21: pre_validation_overview

In the pre-validation phase, we need to ensure that the system is in its expected steady states before we proceed with fault injection. Given the constraints, we have 10 seconds to perform these checks. We have two steady states to verify: the 'example-pod-running-state' and the 'example-service-http-response-state'.

The 'example-pod-running-state' requires us to check that the Pod is in the 'Running' state at least 90% of the time. We will use the provided Python script to verify this. Since the script checks the Pod status every second, we can run it for 5 seconds to gather sufficient data for validation.

The 'example-service-http-response-state' requires us to ensure that 95% of HTTP requests return a 200 OK status. We will use the K6 script to simulate HTTP requests to the service. The script is configured to run for 5 seconds with 5 virtual users, which should provide enough data to validate this steady state.

Both unit tests will be executed simultaneously to maximize the use of the 10-second window. This approach ensures that we efficiently validate both steady states within the given time constraint, allowing us to proceed confidently to the fault injection phase.
- Verified Steady State #0: ```example-pod-running-state```
  - Workflow Name: ```pre-unittest-example-pod-running-state```
  - Grace Period: ```0s```
  - Duration: ```5s```
- Verified Steady State #1: ```example-service-http-response-state```
  - Workflow Name: ```pre-unittest-example-service-http-response-state```
  - Grace Period: ```0s```
  - Duration: ```5s```

Example 22: fault_injection_overview

In this fault-injection phase, we aim to simulate a cyber attack by injecting two types of faults: PodChaos and NetworkChaos. The goal is to observe how these faults impact the system's steady states. Given the 40-second time constraint, we will stagger the fault injections to maximize the observation of their effects.

First, we will inject the PodChaos fault to simulate a Pod failure. This will directly test the 'example-pod-running-state' steady state, as the Pod's restart policy is set to 'Never', meaning it will not automatically recover. We will start this fault injection immediately at the beginning of the phase and let it run for 20 seconds.

Simultaneously, we will run the unit test for the 'example-pod-running-state' to verify the impact of the PodChaos fault on the Pod's availability.

After 20 seconds, we will inject the NetworkChaos fault to simulate network latency. This will test the 'example-service-http-response-state' steady state by introducing delays in HTTP responses. We will let this fault run for the remaining 20 seconds of the phase.

During the NetworkChaos fault injection, we will also run the unit test for the 'example-service-http-response-state' to verify the impact of network latency on the Service's

ability to respond to HTTP requests.

This staggered approach allows us to isolate the effects of each fault on the respective steady states, providing clear insights into the system's vulnerabilities and resilience.
- Verified Steady State #0: ```example-pod-running-state```
  - Workflow Name: ```fault-unittest-example-pod-running-state```
  - Grace Period: ```0s```
  - Duration: ```20s```
- Verified Steady State #1: ```example-service-http-response-state```
  - Workflow Name: ```fault-unittest-example-service-http-response-state```
  - Grace Period: ```20s```
  - Duration: ```20s```

- Injected Faults #0: ```PodChaos```
  - Workflow Name: ```fault-podchaos```
  - Grace Period: ```0s```
  - Duration: ```20s```

- Injected Faults #1: ```NetworkChaos```
  - Workflow Name: ```fault-networkchaos```
  - Grace Period: ```20s```
  - Duration: ```20s```

Example 23: **post_validation_overview**

In the post-validation phase, we need to ensure that the system has returned to its steady states after the fault injection. Given the 10-second total time for this phase, we will conduct unit tests to verify each steady state. The tests will be executed sequentially due to the short duration, ensuring that each steady state is verified independently. The first test will check the 'example-pod-running-state', ensuring the Pod is running. The second test will verify the 'example-service-http-response-state', ensuring the Service responds with a 200 OK status. Each test will have a brief grace period to allow the system to stabilize after the fault injection, followed by a short duration to perform the checks. This approach ensures that both steady states are validated within the available time, confirming the system's recovery.
- Verified Steady State #0: ```example-pod-running-state```
  - Workflow Name: ```post-unittest-example-pod-running-state```
  - Grace Period: ```1s```
  - Duration: ```4s```

- Verified Steady State #1: ```example-service-http-response-state```
  - Workflow Name: ```post-unittest-example-service-http-response-state```
  - Grace Period: ```5s```
  - Duration: ```4s```

Prompt 17: Agent # 2-3 for adjusting a failure scope

System:
You are a helpful AI assistant for Chaos Engineering.
Given a previous K8s manifests, a Chaos-Engineering-experiment plan for it, and the current K8s manifests, you will determine whether we need to adujst the scope of fault injections for the current K8s manifests.
Always keep the following rules:
- Cosider how you must change or keep the scope (i.e., target) of the fault injecttion comparing the previous K8s manifests and the current K8s manifests.
- You only make minor adjustments related to resource changes, metadata change, etc, so NEVER make any scope changes that alter the original goal of the chaos experiment.
- The output should be formatted as a JSON instance that conforms to the JSON schema below.

As an example, for the schema {\"properties\": {\"foo\": {\"title\": \"Foo\", \"description\": \"a list of strings\", \"type\": \"array\", \"items\": {\"type\": \"string\"}}}, \"required\": [\"foo\"]}\nthe object {\"foo\": [\"bar\", \"baz\"]} is a well-formatted instance of the schema. The object {\"properties\": {\"foo\": [\"bar\", \"baz\"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "thought": {
      "title": "Thought",
      "description": "Describe why you need to change/keep the scope of the fault injection for the current K8s manifests.",
      "type": "string"
    },
    "selector": {
      "title": "Selector",
      "description": "Adjust the scope (target) of the fault injection comparing the differeneces between the current and previous manifests. ",
      "allOf": [
        {
          "$ref": "#/definitions/Selectors"
        }
      ]
    }
  },
  "required": [
    "thought",
    "selector"
  ],
  "definitions": {
    "SetBasedRequirements": {
      "title": "SetBasedRequirements",
      "type": "object",
      "properties": {
        "key": {
          "title": "Key",
          "description": "Label key",
          "type": "string"
        },
        "operator": {
          "title": "Operator",
          "description": "Select an operator.",
          "enum": [
            "In",
            "NotIn",
            "Exists",
            "DoesNotExist"
          ],
          "type": "string"
        },
        "values": {
          "title": "Values",
          "description": "Label values. The values set must be non-empty in the case of In and NotIn.",
          "type": "array",
          "items": {
```

```
            "type": "string"
          }
        }
      },
      "required": [
        "key",
        "operator",
        "values"
      ]
    },
    "Selectors": {
      "title": "Selectors",
      "type": "object",
      "properties": {
        "namespaces": {
          "title": "Namespaces",
          "description": "Specifies the
          namespace of the experiment's
          target Pod. If this selector is
          None, Chaos Mesh will set it to
          the namespace of the current
          Chaos experiment.",
          "type": "array",
          "items": {
            "type": "string"
          }
        },
        "labelSelectors": {
          "title": "Labelselectors",
          "description": "Specifies the
          label-key/value pairs that the
          experiment's target Pod must
          have. If multiple labels are
          specified, the experiment target
          must have all the labels
          specified by this selector.",
          "type": "object",
          "additionalProperties": {
            "type": "string"
          }
        },
        "expressionSelectors": {
          "title": "Expressionselectors",
          "description": "Specifies a set
          of expressions that define the
          label's rules to specifiy the
          experiment's target Pod.",
          "example": [
            {
              "key": "tier",
              "operator": "In",
              "values": [
                "cache"
              ]
            },
            {
              "key": "environment",
              "operator": "NotIn",
              "values": [
                "dev"
              ]
            }
          ],
          "type": "array",
          "items": {
            "$ref":
            "#/definitions/SetBasedRequirements"
          }
        },
        "annotationSelectors": {
          "title": "Annotationselectors",
          "description": "Specifies the
          annotation-key/value pairs that
          the experiment's target Pod must
          have. If multiple annotations are
          specified, the experiment target
          must have all annotations
          specified by this selector.",
          "type": "object",
          "additionalProperties": {
            "type": "string"
          }
        },
        "fieldSelectors": {
          "title": "Fieldselectors",
          "description": "Specifies the
          field-key/value pairs of the
          experiment's target Pod. If
          multiple fields are specified,
          the experiment target must have
          all fields set by this selector.",
          "example": {
            "metadata.name": "my-pod",
            "metadata.namespace": "dafault"
          },
          "type": "object",
          "additionalProperties": {
            "type": "string"
          }
        },
        "podPhaseSelectors": {
          "title": "Podphaseselectors",
          "description": "Specifies the
          phase of the experiment's target
          Pod. If this selector is None,
          the target Pod's phase is not
          limited.",
          "type": "array",
          "items": {
            "enum": [
              "Pending",
              "Running",
              "Succeeded",
              "Failed",
              "Unknown"
            ],
            "type": "string"
          }
        },
        "nodeSelectors": {
          "title": "Nodeselectors",
          "description": "Specifies the
          node-label-key/value pairs to
          which the experiment's target Pod
          belongs.",
          "type": "object",
          "additionalProperties": {
            "type": "string"
          }
        },
        "nodes": {
          "title": "Nodes",
          "description": "Specifies the
          node to which the experiment's
          target Pod belongs. The target
          Pod can only belong to one node
          in the configured node list. If
          multiple node labels are
          specified, the node to which the
          experiment's target Pod belongs
          must have all labels specified by
          this selector.",
          "type": "array",
          "items": {
            "type": "string"
          }
        },
        "pods": {
          "title": "Pods",
          "description": "Specifies the
          namespaces and list of the
          experiment's target Pods. If you
          have specified this selector,
          Chaos Mesh ignores other
          configured selectors.",
          "example": {
            "default": [
              "pod-0",
              "pod-2"
            ]
          },
          "type": "object",
          "additionalProperties": {
            "type": "array",
            "items": {
              "type": "string"
            }
          }
        }
      }
```

```
    }
  }
}
```

Human:
Here is the previous K8s manifests of my system:
{prev_k8s_yamls}

Here is a planned Chaos Engineering:
{experiment_plan_summary}

Here is the current K8s menifests of my system:
{curr_k8s_yamls}

Here is the scope of a fault injection for the previous manifests.
{curr_fault_injection}

Now, please adjust the scope of the fault injection for the current manifests. Note that you here focus on the 'selector' parameter (i.e., scope).

Prompt 18: Agent # 2-4 for adjusting a VaC script

System:
You are a helpful AI assistant for Chaos Engineering.
Given the previous K8s manifests, a previous unit test to verify whether the steady state satisfies the threshold, and the reconfigured K8s manifests, you will determine whether the unit test requires adjustment to account for the changes in the reconfigured manifests, and adjust it as necessary.
Always keep the following rules:
- First, consider which K8s manifest resource is the target of the unit test. If there are changes to that manifest, update the unit test as necessary. If there are no changes, the unit test should not require modification.
- You may only make minor adjustments to K8s API, HTTP, or DNS request to account for changes in resource types, parameter seetings, metadata, etc.
- The reconfiguration was made so that the system satisfy the threshold value in the previous unit test, so the threshold value or other parameters must remain unchanged in the new unit test. For example, suppose the number of replicas was reconfigured from 1 to 3 in order to maintain a steady state with more than 1 active pod at all times. In such cases, changing the threshold value from 1 to 3 would alter the intent of this steady state, so the threshold value must remain unchanged (i.e., more than 1 active pod)."
- If redundancy has been newly added, the unit test should verify whether the steady state is maintained by the entire redundancy.
- If the unit test's content needs no changes and only function or variable names need to be changed, leave them as they are to save output costs.
- The output should be formatted as a JSON instance that conforms to the JSON schema below.

As an example, for the schema {\"properties\": {\"foo\": {\"title\": \"Foo\", \"description\": \"a list of strings\", \"type\": \"array\", \"items\": {\"type\": \"string\"}}}, \"required\": [\"foo\"]}\nthe object {\"foo\": [\"bar\", \"baz\"]} is a well-formatted instance of the schema. The object {\"properties\": {\"foo\": [\"bar\", \"baz\"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "thought": {
      "title": "Thought",
      "description": "Describe your thought process for determining whether the unit test requires adjustment to account for the changes in the reconfigured manifests: First, consider which K8s manifest resource is the target of the unit test. If there are changes to that manifest, update the unit test as necessary. If there are no changes, the unit test should not require modification. If the unit test needs updating, describe also how you modify the inspection method according to the differences between the previous and reconfigured manifests. If the modification is not required, describe the reason.",
      "type": "string"
    },
    "code": {
      "title": "Code",
      "description": "If the unit test needs updating, write a new unit test code with the inspection method modified. Write only the content of the code without enclosing it in a code block. If not, this field is not required.",
      "type": "string"
    }
  },
  "required": [
    "thought"
  ]
}
```

Human:
Here is the previous K8s manifests of my system:
{prev_k8s_yamls}

Here is the reconfigured K8s manifests of my system:
{curr_k8s_yamls}

Here is the unit test for the previous manifests.
{prev_unittest}

Now, please determine whether the unit test requires adjustment to account for the changes in the reconfigured manifests, and adjust it as necessary.

In the verification loop, the prompts below will be stacked as history

AI:
{output}

Human:
Your current unit test causes errors when conducted.
The error message is as follows:
{error_message}

This unit test should be succeeded.
Please analyze the reason why the errors

occur, then fix the errors.
Always keep the following rules:
- NEVER repeat the same fixes that have been made in the past.
- Fix only the parts related to the errors without changing the original intent.
- **{the same format instructions as in the System role}**

Example 24: **experiment_plan_summary**

The Chaos Engineering experiment is structured into three phases: pre-validation, fault injection, and post-validation, all to be completed within a total of 1 minute.

In the pre-validation phase, which lasts for 10 seconds, two unit tests are executed simultaneously to verify the system's steady states before any faults are introduced. The 'example-pod-running-state' is checked using a Python script to ensure the Pod is in the 'Running' state at least 90% of the time. This test runs for 5 seconds. Concurrently, the 'example-service-http-response-state' is verified using a K6 script to simulate HTTP requests, ensuring 95% of requests return a 200 OK status. This test also runs for 5 seconds. Both tests start immediately at the beginning of the phase.

The fault injection phase spans 40 seconds and involves two staggered fault injections. Initially, the PodChaos fault is injected to simulate a Pod failure, running for the first 20 seconds. Simultaneously, the 'example-pod-running-state' unit test is conducted to observe the impact of this fault. After 20 seconds, the NetworkChaos fault is introduced to simulate network latency, running for the remaining 20 seconds. During this period, the 'example-service-http-response-state' unit test is executed to assess the effect of network delays. This staggered approach allows for isolated observation of each fault's impact on the system.

Finally, the post-validation phase, lasting 10 seconds, ensures the system returns to its steady states after fault removal. The tests are conducted sequentially. The 'example-pod-running-state' is verified first, with a 1-second grace period followed by a 4-second test duration. Subsequently, the 'example-service-http-response-state' is checked, starting after a 5-second grace period and running for 4 seconds. This sequence confirms the system's recovery to its expected steady states.

D.5.4 Analysis

Prompt 19: Agent # 3-0 for analyzing an experiment results

System:
You are a helpful AI assistant for Chaos Engineering.
Given K8s manifests for a network system, its hypothesis, the overview of a Chaos-Engineeering experiment, and the experimental results, you will analyze the experimental results.
Always keep the following rules:
- Analyze step by step why the test(s) failed, based on the system configurations (manifests) and the flow of the experiment.
- Specify the cause while mentioning the corresponding system configurations and the corresponding phenomena in the Chaos-Engineering experiment.
- The analysis report here will be used for reconfiguring the system later to avoid the failures and improve resiliency. Therefore, make carefully the report rich in insights so that it will be helpful at that time.
- When providing insights and reconfiguration recommendations, limit them to areas related to the failed test.
- The output should be formatted as a JSON instance that conforms to the JSON schema below.

As an example, for the schema {\"properties\": {\"foo\": {\"title\": \"Foo\", \"description\": \"a list of strings\", \"type\": \"array\", \"items\": {\"type\": \"string\"}}}, \"required\": [\"foo\"]}\nthe object {\"foo\": [\"bar\", \"baz\"]} is a well-formatted instance of the schema. The object {\"properties\": {\"foo\": [\"bar\", \"baz\"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "report": {
      "title": "Report",
      "description": "Analysis of the experiment result.",
      "type": "string"
    }
  },
  "required": [
    "report"
  ]
}
```

Human:
Here is the overview of my system:
{user_input2}

Here is the hypothesis for my system:
The hypothesis is "The steady states of the sytem are maintained even when the fault scenario occurs (i.e., when the faults are injected)".
The steady states here are as follows:
{steady_states}

The fault scenario here is as follows:
{detailed_fault_scenario}

Here is the overview of my Chaos-Engineering experiment to verify the hypothesis:
{experiment_plan_summary}

For the first analysis, the following prompt is added

The experiment's results are as follows:
{experiment_result}

Now, please analyze the results and provide an analysis report rich in insights.

For the second and subsequent analyses, the following prompt is added

The update history for the above K8s manifests is the following:

```
{reconfig_history}

# The experiment's results in the latest K8s
 manifests are as follows:
{experiment_result}

Now, please analyze the results and provide
an analysis report rich in insights.
```

Example 25: **experiment_result**

```
Passed unittests:
- pre-unittest-example-pod-running-state
- pre-unittest-example-service-http-response
-state

Failed unittests:
- fault-unittest-example-pod-running-state
```log
Exception when calling CoreV1Api->
read_namespaced_pod: (404)
Reason: Not Found\nHTTP response headers:
HTTPHeaderDict({'Audit-Id': '8a1e6c00-ebd9
-43ee-9522-6399ce015252', 'Cache-Control': '
no-cache, private', 'Content-Type': '
application/json', 'X-Kubernetes-Pf-
Flowschema-Uid': 'c4624bd9-7fc7-42c6-bcb8
-4235110a860d', 'X-Kubernetes-Pf-
Prioritylevel-Uid': '4706085f-6263-43ae-93f5
-b4a61de8b6be', 'Date': 'Sun, 24 Nov 2024
12:06:18 GMT', 'Content-Length': '190'})
HTTP response body: {\"kind\":\"Status
\"...', 'X-Kubernetes-Pf-Flowschema-Uid': '
c4624bd9-7fc7-42c6-bcb8-4235110a860d', 'X-
Kubernetes-Pf-Prioritylevel-Uid': '4706085f
-6263-43ae-93f5-b4a61de8b6be', 'Date': 'Sun,
 24 Nov 2024 12:06:37 GMT', 'Content-Length
': '190'})\nHTTP response body: {\"kind\":\"
Status\",\"apiVersion\":\"v1\",\"metadata
\":{},\"status\":\"Failure\",\"message\":\"
pods \\\"example-pod\\\" not found\",\"
reason\":\"NotFound\",\"details\":{\"name
\":\"example-pod\",\"kind\":\"pods\"},\"code
\":404}

Pod was running 0 out of 20 seconds, which
is 0.00% of the time.
```

- fault-unittest-example-service-http-
response-state
```log
time=\"2024-11-24T12:06:38Z\" level=warning
msg=\"Request Failed\" error=\"Get \\\"http
:\/\/example-service.default.svc.cluster.
local:80\\\": dial tcp 10.96.255.84:80:
connect: connection refused\"\ntime
=\"2024-11-24T12:06:38Z\" level=warning msg
=\"Request Failed\" error=\"Get \\\"http
:\/\/example-service.default.svc.cluster.
local:80\\\": dial tcp 10.96.255.84:80:
connect: connection refused\"\ntime
=\"2024-11-24T12:06:38Z\" level=warning msg
=\"Request Failed\" error=\"Get \\\"http
:\/\/example-service.default.svc.cluster.
local:8... level=error msg=\"thresholds on
metrics 'http_req_failed' have been crossed
```
```
```

D.5.5 Improvement

Prompt 20: Agent # 4-0 for reconfiguring K8s manifests

```
System:
You are a helpful AI assistant for Chaos
Engineering.
Given K8s manifests that define a network
system, its hypothesis, the overview of a
Chaos-Engineeering experiment, and the
experiment's results, you will reconfigure
the system based on analysis of the
experiment's results.
Always keep the following rules:
- NEVER change the original intention (its
description) of the original version of the
system.
- NEVER do the same reconfiguration as in
the history.
- Start with simple reconfiguration, and if
the hypothesis is still not satisfied,
gradually try more complex reconfigurations.
- The output should be formatted as a JSON
instance that conforms to the JSON schema
below.

As an example, for the schema {\"properties
\": {\"foo\": {\"title\": \"Foo\", \"
description\": \"a list of strings\", \"type
\": \"array\", \"items\": {\"type\": \"
string\"}}}, \"required\": [\"foo\"]}\nthe
object {\"foo\": [\"bar\", \"baz\"]} is a
well-formatted instance of the schema. The
object {\"properties\": {\"foo\": [\"bar\",
\"baz\"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "thought": {
      "title": "Thought",
      "description": "Describe your plan to
      modify the K8s manifests.",
      "type": "string"
    },
    "modified_k8s_yamls": {
      "title": "Modified K8S Yamls",
      "description": "The list of modified
      K8s manifests (yamls). If you create a
       new manifest to modify resources in
      an existing manifest, make sure to
      delete the existing manifest before
      creating the new one.",
      "type": "array",
      "items": {
        "$ref": "#/definitions/ModK8sYAML"
      }
    }
  },
  "required": [
    "thought",
    "modified_k8s_yamls"
  ],
  "definitions": {
    "ModK8sYAML": {
      "title": "ModK8sYAML",
      "type": "object",
      "properties": {
        "mod_type": {
          "title": "Mod Type",
          "description": "Modification type.
           Select from ['replace', 'create',
           'delete']. The 'replace' replaces
          /overwites the content of an
          exisiting yaml. The 'create'
          creates a new yaml. The 'delete'
          deletes an existing yaml.",
          "enum": [
            "replace",
            "create",
            "delete"
          ],
```

```
          "type": "string"
        },
        "fname": {
          "title": "Fname",
          "description": "The file name of
          the modified yaml. If mod_type is
          'replace' or 'delete', the name
          must match an existing yaml's name
          . If mod_type='create', name the
          file appropriately to avoid
          overlapping with existing yamls'
          names.",
          "type": "string"
        },
        "explanation": {
          "title": "Explanation",
          "description": "If mod_type is '
          delete', explain why you need to
          delete the yaml. If mod_type is '
          replace', explain which part you
          should modify from the original
          conde and why. If mod_type is '
          create', explain whether it is a
          completely new resource or a
          replacement resouce for an
          existing resource. If it is a
          replacement, also explain the
          differences and the reasons for
          them, just like with 'replace'.",
          "type": "string"
        },
        "code": {
          "title": "Code",
          "description": "If mod_type is '
          delete', this field is not
          required. Otherwise, write the
          content of a K8s YAML manifest
          modified to pass all the unit
          tests. Write only the content of
          the code, and for dictionary
          values, enclose them within a pair
           of single double quotes (\").",
          "type": "string"
        }
      },
      "required": [
        "mod_type",
        "fname",
        "explanation"
      ]
    }
  }
}
```

Human:
Here is the overview of my system (
original version):
{user_input2}

Here is the hypothesis for my system:
The hypothesis is "The steady states of the
sytem are maintained even when the fault
scenario occurs (i.e., when the faults are
injected)".
The steady states here are as follows:
{steady_states}

The fault scenario here is as follows:
{detailed_fault_scenario}

Here is the overview of my Chaos-
Engineering experiment to verify the
hypothesis:
{experiment_plan_summary}

The experiment's results of the original
system are as follows:
{experiment_result}

First, please analyze the results and
provide an analysis report rich in insights.

AI:
Here is my analysis report:
{analysis_report}

Human:
Then, please reconfigure the system to avoid
 the fails (improve resiliency).

In the improvement loop, the prompts below will be stacked as improvement history

AI:
```json
{output}
```"""

Human:
Here is the K8s menifests of the modified
system (version=**{mod_version}**):
{k8s_yamls_mod}

The experiment's results of the modified
system were as follows:
{experiment_result_mod}

Please analyze the results and provide an
analysis report rich in insights again.

AI:
Here is my analysis report:
{analysis_report_mod}

Human:
Then, please reconfigure the system to avoid
 the fails (improve resiliency).

In the verification loop, the prompts below will be stacked as verification history. Verification history is kept within a single verification loop and is reset in every improvement iteration

AI:
```json
{output}
```

User:
Your current unittest causes errors when
conducted.
The error message is as follows:
{error_message}

Please analyze the reason why the errors
occur, then fix the errors.
Always keep the following rules:
- Ensure that the implementation supports
variable durations again.
- NEVER repeat the same fixes that have been
 made in the past.
- Fix only the parts related to the errors
without changing the original content.
- **{the same format instruction as in the
System role}**

Example 26: `analysis_report`

The Chaos Engineering experiment aimed to
test the system's resilience by simulating a
 cyber attack through PodChaos and
NetworkChaos. The experiment results
indicate failures in maintaining the defined
 steady states during and after the fault
injection phase. Here is a detailed analysis
 of the results:

1. **Pre-Validation Phase:**
  - Both pre-unittests, `example-pod-running

```
  -state` and `example-service-http-response
  -state`, passed successfully. This
  indicates that the system was in a healthy
   state before the fault injection, with
  the Pod running 100% of the time and the
  Service responding with a 200 OK status
  for all requests.

2. **Fault Injection Phase:**
  - **PodChaos Fault:** The `fault-unittest-
  example-pod-running-state` failed because
  the Pod was not found (404 error) during
  the test. This is expected due to the
  PodChaos action 'pod-kill', which
  terminated the Pod. Since the Pod's
  restart policy is set to 'Never', it did
  not restart, leading to 0% uptime during
  the test. This highlights the critical
  issue of having a single Pod with no
  redundancy or automatic recovery mechanism
  .
  - **NetworkChaos Fault:** The `fault-
  unittest-example-service-http-response-
  state` also failed. The logs show repeated
   connection refused errors, indicating
  that the Service could not route traffic
  to the Pod, as it was not running. This
  failure is a direct consequence of the Pod
   being unavailable, demonstrating the lack
   of redundancy and the impact of a single
  point of failure.

3. **Post-Validation Phase:**
  - Both post-unittests, `example-pod-
  running-state` and `example-service-http-
  response-state`, failed. The Pod was still
   not found, and the Service continued to
  refuse connections. This indicates that
  the system did not recover to its steady
  states after the faults were removed,
  primarily due to the Pod's restart policy
  and the absence of a controller to manage
  Pod lifecycle and redundancy.

**Insights and Recommendations:**
  - **Pod Restart Policy:** The Pod's
  restart policy should be changed from '
  Never' to 'Always' or 'OnFailure' to
  ensure automatic recovery in case of
  failures.
  - **Redundancy and Scalability:**
  Implement a Deployment or ReplicaSet to
  manage the Pod. This will provide
  redundancy and ensure that a minimum
  number of Pods are always running,
  improving the system's resilience to
  failures.
  - **Service Availability:** Ensure that
  the Service can handle traffic even if one
   Pod fails by having multiple replicas.
  This can be achieved by scaling the
  Deployment to have more than one replica.
  - **Monitoring and Alerts:** Implement
  monitoring and alerting mechanisms to
  detect and respond to Pod failures
  promptly, minimizing downtime.

By addressing these issues, the system can
improve its resilience and maintain its
steady states even during fault scenarios.
```

D.5.6 Post-processing

Prompt 21: Agent # EX for summarizing a completed CE cycle

```
System:
You are a helpful AI assistant for Chaos
Engineering.
Given a summary of a Chaos Engineering cycle
, please elaborate the summary.

The output should be formatted as a JSON
instance that conforms to the JSON schema
below.

As an example, for the schema {\"properties
\": {\"foo\": {\"title\": \"Foo\", \"
description\": \"a list of strings\", \"type
\": \"array\", \"items\": {\"type\": \"
string\"}}}, \"required\": [\"foo\"]}\nthe
object {\"foo\": [\"bar\", \"baz\"]} is a
well-formatted instance of the schema. The
object {\"properties\": {\"foo\": [\"bar\",
\"baz\"]}} is not well-formatted.

Here is the output schema:
```
{
  "properties": {
    "summary": {
      "title": "Summary",
      "type": "string"
    }
  },
  "required": [
    "summary"
  ]
}
```

Human:
Here is the overview of a Chaos Engineering
Cycle:
# Here is a Chaos Engineering cycle
## Step 0. User-input understanding
### Here is the overview of user inputs:
{user_input2}

## Step 1. Hypothesis definition
### Here is the overview of the hypothesis
for the system:
The hypothesis is "The steady states of the
system are maintained even when the fault
scenario occurs (i.e., when the faults are
injected)".
The steady states here are as follows:
{steady_states}

The fault scenario here is as follows:
{detailed_fault_scenario}

## Step 2.1. Chaos-Engineering experiment
### Here is the overview of my Chaos-
Engineering experiment to verify the
hypothesis:
{experiment_plan_summary}

## Step 2.2, 3, 4. Experiment execution,
analysis and improvement (reconfiguring the
system to satisfy the hypothesis)
### Here is the improvement history:
{improvement_history}

Please elaborate the above summary of a
Chaos Engineering Cycle.
```

Example 27: improvement_history

```
### Experiment result (1st try)
//{experiment_result}//

### Analysis report (1st try)
//analysis_report//

### Improvement result (1st try)
1 K8s manifests are modified:
- The K8s manifest 'nginx/pod.yaml' was
replaced.

Replace the Pod manifest with a Deployment
manifest to provide redundancy and automatic
 recovery. The Deployment will manage the
Pod lifecycle, ensuring that a specified
```

```
number of replicas are always running, which
 addresses both the restart policy and
redundancy issues.
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-deployment
  labels:
    app: example
spec:
  replicas: 3
  selector:
    matchLabels:
      app: example
    template:
      metadata:
        labels:
          app: example
      spec:
        containers:
        - name: example-container
          image: nginx:1.17.1
          ports:
          - containerPort: 80
```

Experiment result (2nd try)
Passed unittests:
- pre-unittest-example-pod-running-state
- pre-unittest-example-service-http-response
-state
- fault-unittest-example-pod-running-state
- fault-unittest-example-service-http-
response-state
- post-unittest-example-pod-running-state
- post-unittest-example-service-http-
response-state

Failed unittests:
```

E Full Results of Case Studies

In this section, we provide full inputs and outputs for each run studied in the case study. The inputs include instructions, Skaffold configuration files, K8s manifests. The outputs include VaC scripts, Chaos Mesh workflow manifests, reconfigured K8s manifests, adjusted VaC scripts in the experiment replanning phase. In the case study, we pick up the fourth run for NGINX and the fifth run for SOCKSHOP. Table 4 shows the statistics of NGINX and SOCKSHOP. Before showing the full results, we describe the CE cycles for each system referring to their highlight outputs.

Table 4: Statistics of the systems for the case study. Tokens are counted by the tokenizer `cl100k_base`.

System	# manifests	# lines	# words	# tokens
NGINX	2	24	373	115
SOCKSHOP	29	869	17696	4605

Description of the results for NGINX In the *hypothesis* phase, CHAOSEATER first defines two steady states: 1) "The `Pod` should be running at least 90% of the time during the check period"; 2) "Service availability should be at least 99.9% with a response status of 200". The VaC scripts shown in Figure 2 correctly implement these steady states. It then defines a failure sequence that injects `NetworkChaos` (delay) into the Nginx `Pod` following `PodChaos` (pod-kill) to simulate a cyber-attack.

In the experiment planning, it plans the following chaos experiment: 1) the two steady states are sequentially validated in the pre-validation phase; 2) in the failure-injection phase, the two steady states are sequentially validated alongside the injection of each failure that may affect them; 3) the two steady states are validated sequentially once again in the post-validation phase.

The first chaos experiment reveals that the two steady states are currently not satisfied in both the failure-injection and post-validation phases. CHAOSEATER successfully identifies the cause as the Nginx `Pod`'s never restarting policy and replaces it with a `Deployment` with three replicas. It then adjusts the success criteria of the VaC script to ensure that at least one Pod out of three replicas is running and conducts the chaos experiment for the reconfigured system again. Finally, the additional chaos experiment confirms that the hypothesis is satisfied in the reconfigured system.

Intuitively, these operations and outputs follow best practices, demonstrating that CHAOSEATER can complete our expected CE cycle for the small-scale system, NGINX, without explicit user instructions.

Description of the results for SOCKSHOP In the *hypothesis* phase, CHAOSEATER first defines two steady states: 1) "At least 1 ready replica 100% of the time and 2 ready replicas at least 80% of the time during the monitoring period" for the carts `Deployment`; 2) "At least 1 ready replica must be present 100% of the time during the monitoring period" for the front-end `Deployment`. The VaC scripts shown in Figure 2 correctly implement these steady states.

It then defines a failure sequence that injects `StressChaos` (CPU) into all the carts-db replicas following `PodChaos` (pod-kill) that targets the single front-end replica to simulate possible problems in a black Friday sale.

In the experiment planning, it plans the following chaos experiment: 1) the two steady states are simultaneously validated in the pre-validation phase; 2) in the failure-injection phase, the two

steady states are sequentially validated alongside the injection of each failure that may affect them; 3) the two steady states are validated simultaneously once again in the post-validation phase.

The first chaos experiment reveals that the second steady state is currently not satisfied in both the failure-injection and post-validation phases. ChaosEater successfully identifies the cause as the single replica setting of the front-end `Deployment` and increases the number of replicas to two. It then conducts the chaos experiment for the reconfigured system again without experiment adjustments (no adjustment is needed as the resource type remains unchanged). Finally, the additional chaos experiment confirms that the hypothesis is satisfied in the reconfigured system.

This case also broadly follows best practices, demonstrating that ChaosEater can complete our expected CE cycle even for the comparatively large system, SockShop, without explicit user instructions.

E.1 Nginx

E.1.1 Inputs

Instructions

```
The Chaos-Engineering experiment must be
completed within 1 minute.
```

skaffold.yaml

```
apiVersion: skaffold/v3
kind: Config
metadata:
  name: nginx-app
manifests:
  rawYaml:
    - pod.yaml
    - service.yaml
```

pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
  labels:
    app: example
spec:
  restartPolicy: Never
  containers:
  - name: example-container
    image: nginx:1.17.1
    ports:
    - containerPort: 80
```

service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: example-service
spec:
  selector:
    app: example
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

E.1.2 CE cycle completed by ChaosEater

Preprocessing

```
Summaries of each manifest
nginx/pod.yaml
- This manifest defines a Kubernetes Pod.
- The Pod is named 'example-pod'.
- It includes metadata with a label 'app:
example'.
- The Pod's restart policy is set to
'Never', meaning it won't restart
automatically
  if it fails.
- The Pod contains one container named
'example-container'.
- The container uses the 'nginx:1.17.1'
image, which is a specific version of the
  Nginx webserver.
- The container exposes port 80, which is
commonly used for HTTP traffic.

nginx/service.yaml
- This manifest defines a Kubernetes
Service.
- The Service is named 'example-service'.
- It uses the 'v1' API version.
- The Service selects pods with the label
'app: example'.
- It exposes the Service on port 80 using
the TCP protocol.
- The Service forwards traffic to the
target port 80 on the selected pods.

Resiliency issues/weaknesses in the
manifests
Issue #0: Pod Restart Policy
details: The Pod will not restart
automatically if it fails, which can lead
to downtime.
manifests having the issues:
['nginx/pod.yaml']
problematic config: restartPolicy: Never

Issue #1: Single Pod Deployment
details: Having a single Pod means there is
no redundancy. If the Pod fails, the
service will be unavailable.
manifests having the issues:
['nginx/pod.yaml']
problematic config: Only one Pod is defined
without a Deployment or ReplicaSet.

Application of the manifests
Thoughts:
The manifests provided define a Kubernetes
Pod and a Service. The Pod runs an Nginx
container, which is a popular web server
used to serve static content, reverse
proxy, or load balance HTTP traffic. The
Service is configured to expose this Pod on
port 80, which is the default port for HTTP
traffic. Given the use of Nginx and the
configuration of the Service, it is logical
to assume that these manifests are intended
to deploy a simple web server application.
The file names and the use of Nginx further
```

support this assumption.

Assumed application:
A simple web server application using Nginx to serve HTTP content.

Summary of user instructions
- The Chaos-Engineering experiment must be completed within 1 minute.

Steady state #1: example-pod-running

//Description//
The first issue to address is the Pod's restart policy set to 'Never'. This is a critical issue because if the Pod fails, it will not restart, leading to potential downtime. Therefore, the steady state should ensure that the Pod is running and available. A measurable output for this steady state is the number of running Pods, which should be 1, as there is only one Pod defined in the manifest. This steady state will help verify that the Pod is up and running, which is crucial given the restart policy configuration.

//Current state inspection//
To inspect whether the 'example-pod' is running, I will use the K8s API. This is because the state to be checked is the status of a Kubernetes resource, specifically the Pod, which is best done using the Kubernetes client libraries. The script will check the status of the Pod every second for a duration of 5 seconds, which is within the allowed maximum duration. This will ensure that the Pod is running and available, addressing the critical issue of the Pod's restart policy being set to 'Never'.
tool: k8s duration: 5s
Inspection script (Python)
k8s_example-pod-running.py :
```
import os
import time
from kubernetes import client, config

# Load Kubernetes configuration based on
the environment
if os.getenv('KUBERNETES_SERVICE_HOST'):
    config.load_incluster_config()
else:
    config.load_kube_config()

v1 = client.CoreV1Api()

def check_pod_status(namespace, pod_name):
    try:
        pod = v1.read_namespaced_pod(
            name=pod_name, namespace=names)
        return pod.status.phase == 'Running'
    except client.exceptions.ApiException
    as e:
        print(f"Exception when calling
        CoreV1Api->read_namespaced_po
        return False

def main(duration):
    namespace = 'default'
    pod_name = 'example-pod'
    running_count = 0
    for _ in range(duration):
        if check_pod_status(namespace,
        pod_name):
            running_count += 1
        time.sleep(1)
    print(f"Pod '{pod_name}' running status
    checked {duration} times

if __name__ == '__main__':
    import argparse
    parser =
    argparse.ArgumentParser(description='Check
    if a pod is
    parser.add_argument('--duration',
    type=int, default=5, help='Dur
    args = parser.parse_args()
    main(args.duration)
```

The inspection script's results (current states) are as follows:
```
Pod 'example-pod' running status checked 5
times. Running count: 5.
```

//Threshold//
The steady state we are considering is whether the 'example-pod' is running. The current state shows that the pod was checked 5 times over a duration of 5 seconds, and it was running each time, resulting in a running count of 5. This indicates that the pod is consistently running during the check period. Given the constraints of the chaos engineering experiment, which must be completed within 1 minute, we can set a threshold that allows for some tolerance in case of brief fluctuations. A reasonable threshold would be that the pod should be running at least 90% of the time during the
check period. This allows for a small margin of error while still ensuring that the pod is generally available and running.

The pod should be running at least 90% of the time during the check period.

//Unit test to validate the steady state//
To add the threshold assertion to the inspection Python script, I will calculate the percentage of time the pod is running during the specified duration. The threshold is set to 90%, meaning the pod should be running at least 90% of the time during the check period. I will implement a for-loop to check the pod status every second for the specified duration and then assert that the running percentage is greater than or equal to 90%. This ensures that the pod is generally available and running, allowing for a small margin of error.
```
import os
import time
import argparse
from kubernetes import client, config
from unittest_base import K8sAPIBase

class TestPodRunningState(K8sAPIBase):
    def __init__(self):
        super().__init__()

    def check_pod_status(self, namespace,
    pod_name):
        try:
            pod =
            self.v1.read_namespaced_pod(
                name=pod_name,
                namespace=namespace)
            return pod.status.phase ==
            'Running'
        except
        client.exceptions.ApiException as e:
            print(f"Exception when calling
            CoreV1Api->read_namespaced_pod:
            {e}")
            return False

    def test_pod_running_state(self,
    duration):
        namespace = 'default'
```

```
        pod_name = 'example-pod'
        running_count = 0

        # Check the pod status every second
        for the specified duration
        for _ in range(duration):
            if
            self.check_pod_status(namespace,
            pod_name):
                running_count += 1
            time.sleep(1)

        # Calculate the running percentage
        running_percentage = (running_count
        / duration) * 100

        # Assert that the running
        percentage is at least 90%
        assert running_percentage >= 90,
        f"Pod '{pod_name}' running
        percentage is below threshold:
        {running_percentage}%"

        print(f"Pod '{pod_name}' running
        status checked {duration} times.
        Running percentage:
        {running_percentage}%.")


def main():
    parser =
    argparse.ArgumentParser(description='Test
    if a pod is running at least 90% of the
    time.')
    parser.add_argument('--duration',
    type=int, default=5, help='Duration to
    check the pod status in seconds.')
    args = parser.parse_args()

    test = TestPodRunningState()
    test.test_pod_running_state(args.duration)


if __name__ == '__main__':
    main()
```

The unit test results are as follows:

```
Pod 'example-pod' running status checked 5
times. Running percentage 100%.
```

Steady state #2: example-service-availability

Description
The next issue to address is the 'Single Pod Deployment', which is related to the lack of redundancy. This is a significant issue because if the single Pod fails, the service will be unavailable. To verify this, we can define a steady state that checks the availability of the service itself. A measurable output for this steady state is the service's response time or availability. Since the service is exposed on port 80, we can check if the service is responding to HTTP requests. This steady state will help verify that the service is available and responsive, which is crucial given the single Pod deployment configuration.

Current state inspection
To inspect the availability of the 'example-service', we need to check if the service is responding to HTTP requests. Since the service is exposed on port 80, we can use k6 to simulate HTTP requests to the service and measure its response time and availability. This approach allows us to verify the service's responsiveness and availability, which is crucial given the single Pod deployment configuration. We will use a small number of virtual users and a short duration to fit within the 1-minute constraint of the Chaos Engineering experiment.
tool: k6 vus: 1 duration: 5s
Inspection script (Javascript)
k6_example-service-availability.js :

```
import http from 'k6/http';
import { check } from 'k6';

export const options = {
  vus: 1,
  duration: '5s',
};

export default function () {
  const res = http.get(
    'http://example-service.default.svc.cluster
     .local'
  )
  check(res, {
    'status is 200': (r) => r.status ===
    200,
  });
}
```

The inspection script's results (current states) are as follows:

```
OK status is 200

checks..........: 100.00% 13915 out of 13915
data_received...: 12 MB 2.4 MB/s
data_sent.......: 1.5 MB 306 kB/s
http_req_blocked...: avg=8.11µs min=1.4µs
                     med=4.19µs max=2.82ms
                     p(90)=6.47µs
                     p(95)=7.28µs
http_req_connecting...: avg=2.23µs min=0s
                        med=0s max=345.12µs
                        p(90)=0s p(95)=0s
http_req_duration...: avg=236.57µs
                      min=90.31µs
                      med=232.08µs
                      max=3.38ms
                      p(90)=263.84µs
                      p(95)=280.9µs
{expected_response:true}...: avg=236.57µs
                             min=90.31µs
                             med=232.08µs
                             max=3.38ms
                             p(90)=263.84µs
                             p(95)=280.9µs
http_req_failed...: 0.00% 0 out of 13915
http_req_receiving...: avg=67.9µs
                       min=17.89µs
                       med=61.39µs
                       max=2.36ms
                       p(90)=91.92µs
                       p(95)=99.14µs
http_req_sending...: avg=18.51µs min=4.66µs
                     med=14.77µs max=2.51ms
                     p(90)=21.11µs
                     p(95)=23.45µs
http_req_tls_handshaking...: avg=0s min=0s
                             med=0s max=0s
                             p(90)=0s
                             p(95)=0s
http_req_waiting...: avg=150.16µs
                     min=54.98µs
                     med=151.23µs
                     max=880.47µs
                     p(90)=168.67µs
                     p(95)=176.19µs
http_reqs...: 13915 2782.666155/s
iteration_duration...: avg=344.61µs
                       min=131.12µs
                       med=333.12µs
                       max=3.98ms
```

```
                        p(90)=377.15µs
                        p(95)=401.15µs
iterations...: 13915 2782.666155/s
vus..........: 1 min=1 max=1
vus_max......: 1 min=1 max=1
```

Threshold
The steady state we are considering is the availability of the 'example-service'. The k6 test results show that the service is currently responding with a 200 status code for all requests, indicating 100% availability. Given that the system consists of a single Pod, any failure in the Pod would result in the service being unavailable. Therefore, the threshold should ensure that the service remains available and responsive. To account for minor fluctuations and network latency, a reasonable threshold would be to maintain a high availability percentage, slightly below 100% to allow for brief, noncritical failures. A threshold of 99.9% availability is a common standard for web services, allowing for some tolerance while still ensuring high reliability.

Service availability should be at least 99.9% with a response status of 200.

Unit test to validate the steady state
To add the threshold check to the k6 script, I will include a 'thresholds' property in the 'options' section. This property will specify that the service's availability should be at least 99.9% with a response status of 200. This is done by setting a threshold on the 'http_req_failed' metric, ensuring that the failure rate is less than or equal to 0.1% (which corresponds to 99.9% availability).
```
import http from 'k6/http';
import { check } from 'k6';

export const options = {
  vus: 1,
  duration: '5s',
  thresholds: {
    // Ensure that the service availability
    is at least 99.9%
    'http_req_failed': ['rate<=0.001'], //
    0.1% failure rate corresponds to 99.9%
    availability
  },
};

export default function () {
  const res = http.get(
    'http://example-service.default.svc
     .cluster.local:80');
  check(res, {
    'status is 200': (r) => r.status ===
    200,
  });
}
```

The unit test results are as follows:
```
OK status is 200

checks...: 100.00% 14565 out of 14565
data_received.......: 12 MB 2.5 MB/s
data_sent...........: 1.6 MB 320 kB/s
http_req_blocked....: avg=7.92µs min=1.33µs
                      med=4.1...
http_req_connecting...: avg=2.19µs min=0s
                        med=0s...
http_req_duration.....: avg=225.1µs
                        min=82.21µs
                        med=226...
{ expected_response:true }...: avg=225.1µs
                               min=82.21µs
                               med=226...
OK http_req_failed...: 0.00% 0 out of 14565
http_req_receiving...: avg=66.97µs
                       min=15.24µs
                       med=60...
http_req_sending.....: avg=17.27µs
                       min=4.03µs
                       med=13...
http_req_tls_handshaking...: avg=0s min=0s
                             med=0s...
http_req_waiting.....: avg=140.85µs
                       min=53.55µs
                       med=144...
http_reqs............: 14565 2912.709122/s
iteration_duration...: avg=329.52µs
min=117.97µs
                       med=323...
iterations...........: 14565 2912.709122/s
vus..................: 1 min=1 max=1
vus_max..............: 1 min=1 max=1
```

Failure senario: Cyber Attack

Description
Given the system's weaknesses, a cyber attack targeting the single Pod and its network could be highly impactful. The Pod's restart policy set to 'Never' means that if the Pod fails, it will not restart, leading to downtime. Additionally, the single Pod deployment means there is no redundancy, so any failure will make the service unavailable. To simulate a cyber attack, we can start by injecting a PodChaos fault to kill the Pod, testing the system's ability to handle Pod failures. This will directly exploit the lack of redundancy and the restart policy issue. Next, we can simulate a network attack using NetworkChaos to introduce network latency, testing the service's ability to maintain availability under network stress. This sequence simulates a cyber attack by first taking down the Pod and then stressing the network, revealing the system's vulnerabilities in handling such scenarios.

Failure-injection sequence
PodChaos ({'pod': 'example-pod'}) -> NetworkChaos ({'service': 'example-service'}):

Detailed failure parameters
Detailed parameters of PodChaos ({'pod': 'example-pod'})
```
{
  "action": "pod-kill",
  "mode": "one",
  "selector": {
    "namespaces": [
      "default"
    ],
    "labelSelectors": {
      "app": "example"
    }
  }
}
```

Detailed parameters of NetworkChaos ({'service': 'example-service'}):
```
{
  "action": "delay",
  "direction": "to",
  "target": {
    "mode": "all",
    "selector": {
      "namespaces": ["default"],
```

```
      "labelSelectors": {
        "app": "example"
      }
    }
  },
  "mode": "all",
  "selector": {
    "namespaces": ["default"],
    "labelSelectors": {
      "app": "example"
    }
  },
  "device": "eth0",
  "delay": {
    "latency": "100ms",
    "jitter": "10ms",
    "correlation": "50"
 }
}
```

Chaos experiment (Planning)

Time schedule
Given the constraints of the chaos engineering experiment, which must be completed within 1 minute, we need to allocate time efficiently across the three phases: prevalidation, fault-injection, and post-validation. The pre-validation phase is crucial to ensure that the system is in a steady state before we introduce any faults. Since we have two steady states to validate, we should allocate a reasonable amount of time to check both the pod's running status and the service's availability. A duration of 15 seconds should be sufficient for pre-validation, allowing us to run the necessary checks multiple times. The fault-injection phase is where we introduce the chaos to observe the system's behavior under stress. Given the complexity of the faults (PodChaos and NetworkChaos), we should allocate the majority of the time to this phase to ensure that the faults have enough time to manifest and impact the system. A duration of 30 seconds is appropriate for fault injection, allowing us to observe the system's response to both pod failure and network latency. Finally, the post-validation phase is essential to verify that the system returns to its steady states after the faults are removed. We should allocate 15 seconds for post-validation, similar to the pre-validation phase, to ensure that the system stabilizes and meets the defined thresholds for steady states. This allocation results in a total experiment time of 60 seconds, which fits within the 1-minute constraint.

Total experiment time: 60s
Pre-validation Phase : 15s
Fault-injection Phase: 30s
Post-validation Phase: 15s

Pre-validation phase
In the pre-validation phase, we need to ensure that the system is in its expected steady state before we proceed with fault injection. Given the constraints of a 15-second total time for this phase, we will conduct two unit tests to verify the steady states: one for the pod's running status and another for the service's availability. These tests will be executed sequentially due to the short duration available, ensuring that each steady state is verified independently and thoroughly. The first test will check if the 'examplepod' is running at least 90% of the time over a 5-second period. This is crucial because
the pod's restart policy is set to 'Never', and we need to confirm its availability before introducing any faults. The second test will verify the 'example-service' availability, ensuring it responds with a 200 status code at least 99.9% of the time over another 5-second period. This test is essential to confirm that the service is operational and responsive, given the single pod deployment. By staggering these tests, we can focus on each steady state individually, allowing us to identify any issues before proceeding to the fault injection phase.

- Verified Steady State #0:
example-pod-running
  - Workflow Name:
  pre-unittest-example-pod-running
  - Grace Period: 0s
  - Duration: 5s

- Verified Steady State #1:
example-service-availability
  - Workflow Name:
  pre-unittest-example-service-availability
  - Grace Period: 5s
  - Duration: 5s

Fault-injection phase
In this fault-injection phase, we aim to simulate a cyber attack by injecting two types of faults: PodChaos and NetworkChaos. The total duration for this phase is 30 seconds, so we need to carefully schedule the faults and unit tests to fit within this timeframe.

First, we will inject the PodChaos fault to simulate a pod failure. This fault will be injected at the start of the phase (grace period of 0s) and will last for 10 seconds. This duration is chosen to allow enough time for the system to experience the impact of the pod being killed, given the pod's restart policy is set to 'Never'.

Simultaneously, we will run the unit test for the 'example-pod-running' steady state to verify if the pod is running at least 90% of the time during the fault injection. This test will also start at 0s and run for 10 seconds, aligning with the PodChaos duration.

Next, we will inject the NetworkChaos fault to simulate network latency. This fault will start at 10 seconds (aer the PodChaos fault ends) and will last for 20 seconds. This staggered approach allows us to observe the system's behavior under network stress after the pod failure has been simulated.

During the NetworkChaos fault, we will run the unit test for the 'example-serviceavailability' steady state. This test will start at 10 seconds and run for 20 seconds, matching the NetworkChaos duration. This ensures we are checking the service's availability and response time while the network is under stress.

By staggering the faults and aligning the unit tests with the fault durations, we can effectively observe the system's behavior under each fault condition and verify if the steady states are maintained.

- Verified Steady State #0:
example-pod-running
  - Workflow Name:

```
  fault-unittest-example-pod-running
  - Grace Period: 0s
  - Duration: 10s

- Verified Steady State #1:
example-service-availability
  - Workflow Name:
  fault-unittest-example-service-availability
  - Grace Period: 10s
  - Duration: 20s

- Injected Faults #0: PodChaos
  - Workflow Name: fault-podchaos
  - Grace Period: 0s
  - Duration: 10s

- Injected Faults #1: NetworkChaos
  - Workflow Name: fault-networkchaos
  - Grace Period: 10s
  - Duration: 20s
```

Post-validation phase

In the post-validation phase, we need to ensure that the system has returned to its steady states aer the fault injection. Given the 15-second time constraint, we will perform quick checks to verify the steady states. The two steady states to verify are: 1) the 'example-pod' is running, and 2) the 'example-service' is available. We will execute these checks sequentially due to the short duration, ensuring each test has enough time to gather meaningful data. The first test will check the pod's running status, followed by the service availability test. This order is logical because the pod must be running for the service to be available. Each test will have a brief grace period to allow the system to stabilize aer the fault injection, followed by a short duration to perform the checks.

```
- Verified Steady State #0:
example-pod-running
  - Workflow Name:
  post-unittest-example-pod-running
  - Grace Period: 2s
  - Duration: 6s

- Verified Steady State #1:
example-service-availability
  - Workflow Name:
  post-unittest-example-service-availability
  - Grace Period: 8s
  - Duration: 5s
```

Summary

The chaos engineering experiment is structured into three phases: pre-validation, fault-injection, and post-validation, all to be completed within a total of 60 seconds.

In the pre-validation phase, which lasts for 15 seconds, two unit tests are conducted sequentially to ensure the system is in a steady state before fault injection. The first test, named 'pre-unittest-example-pod-running', checks the 'example-pod' running status. It starts immediately at the beginning of the phase and runs for 5 seconds. Following this, the second test, 'pre-unittest-example-service-availability', begins at the 5-second mark and also runs for 5 seconds, verifying the service's availability.

The fault-injection phase spans 30 seconds and involves two types of faults: PodChaos and NetworkChaos. Initially, the PodChaos fault, named 'fault-podchaos', is injected at the start of the phase and lasts for 10 seconds. Concurrently, the 'fault-unittestexample-pod-running' unit test runs for the same duration to verify the pod's status during the fault. Aer the PodChaos fault concludes, the NetworkChaos fault, named 'fault-networkchaos', begins at the 10-second mark and continues for 20 seconds. Simultaneously, the 'fault-unittest-example-service-availability' test runs for 20 seconds, starting at the same time as the NetworkChaos fault, to check the service's availability under network stress.

Finally, the post-validation phase, also 15 seconds long, ensures the system returns to its steady states. The 'post-unittest-example-pod-running' test starts after a 2-second grace period and runs for 6 seconds to verify the pod's status. Subsequently, the 'postunittest-example-service-availability' test begins at the 8-second mark and runs for 5 seconds, checking the service's availability. This sequential execution allows for a brief stabilization period before each test.

The generated Chaos Mesh workflow manifest:

```
apiVersion: chaos-mesh.org/v1alpha1
kind: Workflow
metadata:
  name: chaos-experiment-20241124-132854
spec:
  entry: the-entry
  templates:
    #-------------------------------
    # entry point of whole workflow
    #-------------------------------
    - name: the-entry
      templateType: Serial
      deadline: 30m51s
      children:
        - pre-validation-phase
        - fault-injection-phase
        - post-validation-phase

    #-------------------------------------
    # Entry point of pre-validation-phase
    #-------------------------------------
    - name: pre-validation-phase
      templateType: Serial
      deadline: 10m10s
      children:
        -
        pre-validation-overlapped-workflows

    - name: pre-validation-suspend-workflow
      templateType: Serial
      deadline: 5m10s
      children:
        - pre-validation-suspend
        - pre-unittest-example-service-
          availability

    - name: pre-validation-suspend
      templateType: Suspend
      deadline: 5s

    - name:
    pre-validation-overlapped-workflows
      templateType: Parallel
      deadline: 5m10s
      children:
        - pre-unittest-example-pod-running
        - pre-validation-suspend-workflow

    # Definitions of children of
    pre-validation-phase
    - name: pre-unittest-example-pod-running
      templateType: Task
      deadline: 5m5s
      task:
        container:
          name:
```

```
      pre-unittest-example-pod-running-
            container
      image: chaos-eater/k8sapi:1.0
      imagePullPolicy: IfNotPresent
      command: ["/bin/bash", "-c"]
      args: ["python
        /chaos-eater/sandbox/cycle_20241124
        _132128/unittest_example-pod-running
        _mod0.py --duration 5"]
      volumeMounts:
        - name: pvc-volume
          mountPath: /chaos-eater
    volumes:
      - name: pvc-volume
        persistentVolumeClaim:
          claimName: pvc

- name:
pre-unittest-example-service-availability
  templateType: Task
  deadline: 5m5s
  task:
    container:
      name:
      pre-unittest-example-service-
            availability-container
      image: grafana/k6:latest
      command: ["k6", "run",
        "--duration", "5s", "--quiet",
        "/chaos-eater/sandbox/cycle_20241124
         _132128/unittest_example-service-
         availability_mod0.js"]
      volumeMounts:
        - name: pvc-volume
          mountPath: /chaos-eater
    volumes:
      - name: pvc-volume
        persistentVolumeClaim:
          claimName: pvc

#-------------------------------------
# Entry point of fault-injection-phase
#-------------------------------------
- name: fault-injection-phase
  templateType: Serial
  deadline: 10m30s
  children:
    -
    fault-injection-overlapped-workflows

- name:
fault-injection-parallel-workflow
  templateType: Parallel
  deadline: 5m10s
  children:
    - fault-unittest-example-pod-running
    - fault-podchaos

- name: fault-injection-suspend-workflow
  templateType: Serial
  deadline: 5m30s
  children:
    - fault-injection-suspend
    - fault-injection-parallel-workflows

- name: fault-injection-suspend
  templateType: Suspend
  deadline: 10s

- name:
fault-injection-parallel-workflows
  templateType: Parallel
  deadline: 5m20s
  children:
    - fault-unittest-example-service-
      availability
    - fault-networkchaos

- name:
fault-injection-overlapped-workflows
  templateType: Parallel
  deadline: 5m30s
  children:
    - fault-injection-parallel-workflow
    - fault-injection-suspend-workflow

# Definitions of children of
pre-validation-phase
# unit tests
- name:
fault-unittest-example-pod-running
  templateType: Task
  deadline: 5m10s
  task:
    container:
      name:
      fault-unittest-example-pod-running-
            container
      image: chaos-eater/k8sapi:1.0
      imagePullPolicy: IfNotPresent
      command: ["/bin/bash", "-c"]
      args: ["python
        /chaos-eater/sandbox/cycle_20241124
        _132128/unittest_example-pod-running
        _mod0.py --duration 10"]
      volumeMounts:
        - name: pvc-volume
          mountPath: /chaos-eater
    volumes:
      - name: pvc-volume
        persistentVolumeClaim:
          claimName: pvc

- name:
fault-unittest-example-service-availability
  templateType: Task
  deadline: 5m20s
  task:
    container:
      name:
      fault-unittest-example-service-
            availability-container
      image: grafana/k6:latest
      command: ["k6", "run",
        "--duration", "20s", "--quiet",
        "/chaos-eater/sandbox/cycle_20241124
         _132128/unittest_example-service-
         availability_mod0.js"]
      volumeMounts:
        - name: pvc-volume
          mountPath: /chaos-eater
    volumes:
      - name: pvc-volume
        persistentVolumeClaim:
          claimName: pvc

# fault_injections
- name: fault-podchaos
  templateType: PodChaos
  deadline: 10s
  podChaos:
    action: pod-kill
    mode: one
    selector:
      labelSelectors:
        app: example
      namespaces:
        - default


- name: fault-networkchaos
  templateType: NetworkChaos
  deadline: 20s
  networkChaos:
    action: delay
    delay:
      correlation: '50'
      jitter: 10ms
      latency: 100ms
    device: eth0
    direction: to
    mode: all
    selector:
      labelSelectors:
        app: example
      namespaces:
        - default
    target:
      mode: all
      selector:
```

```
        labelSelectors:
          app: example
        namespaces:
          - default


#-------------------------------------
# Entry point of post-validation phase
#-------------------------------------
- name: post-validation-phase
  templateType: Serial
  deadline: 10m11s
  children:
    -
    post-validation-overlapped-workflows

- name: post-validation-suspend-workflow
  templateType: Serial
  deadline: 5m8s
  children:
    - post-validation-suspend
    - post-unittest-example-pod-running

- name: post-validation-suspend
  templateType: Suspend
  deadline: 2s

- name:
post-validation-suspend-workflow2
  templateType: Serial
  deadline: 5m11s
  children:
    - post-validation-suspend2
    - post-unittest-example-service-
      availability

- name: post-validation-suspend2
  templateType: Suspend
  deadline: 6s

- name:
post-validation-overlapped-workflows
  templateType: Parallel
  deadline: 5m11s
  children:
    - post-validation-suspend-workflow
    - post-validation-suspend-workflow2

# Definitions of children of
pre-validation-phase
- name:
post-unittest-example-pod-running
  templateType: Task
  deadline: 5m6s
  task:
    container:
      name:
      post-unittest-example-pod-running-
            container
      image: chaos-eater/k8sapi:1.0
      imagePullPolicy: IfNotPresent
      command: ["/bin/bash", "-c"]
      args: ["python
        /chaos-eater/sandbox/cycle_20241124
        _132128/unittest_example-pod-running
        _mod0.py --duration 6"]
      volumeMounts:
        - name: pvc-volume
          mountPath: /chaos-eater
    volumes:
      - name: pvc-volume
        persistentVolumeClaim:
          claimName: pvc

- name:
post-unittest-example-service-availability
  templateType: Task
  deadline: 5m5s
  task:
    container:
      name:
      post-unittest-example-service-
            availability-container
      image: grafana/k6:latest
      command: ["k6", "run",
            "--duration", "5s", "--quiet",
            "/chaos-eater/sandbox/cycle_20241124
             _132128/unittest_example-service-
             availability_mod0.js"]
          volumeMounts:
            - name: pvc-volume
              mountPath: /chaos-eater
        volumes:
          - name: pvc-volume
            persistentVolumeClaim:
              claimName: pvc
```

Analysis

The chaos engineering experiment results indicate several critical issues in the system's configuration and its ability to handle faults, particularly in the context of the defined fault scenario. Here is a detailed analysis of the failures observed during the experiment:

1. Pod Restart Policy and Single Pod Deployment:
The 'fault-unittest-example-pod-running' test failed because the Pod was not found after the PodChaos fault was injected. This is directly related to the Pod's restart policy set to 'Never' in the nginx/pod.yaml manifest. When the Pod was killed, it did not restart, leading to a 404 error when attempting to read the Pod's status. This confirms the identified issue #0 (Pod Restart Policy) and issue #1 (Single Pod Deployment), where the lack of redundancy and automatic recovery mechanisms resulted in the Pod being unavailable.

2. Service Availability:
The 'fault-unittest-example-service-availability' test failed due to connection refusals, indicating that the service was unavailable. This failure is a consequence of the Pod being killed and not restarting, as the Service relies on the Pod to handle requests. The Service's selector in nginx/service.yaml targets Pods with the label 'app: example', and with the Pod being down, there were no endpoints available to serve the requests, leading to the connection refused errors.

3. Post-Validation Failures:
Both 'post-unittest-example-pod-running' and 'post-unittest-example-service-availability' tests failed, showing that the system did not recover to its steady state after the fault injection. The Pod remained unavailable, and consequently, the Service could not respond to HTTP requests. This further highlights the critical impact of the Pod's restart policy and the lack of redundancy.

Insights and Recommendations:
- Pod Restart Policy: Change the Pod's restart policy from 'Never' to 'Always' or 'OnFailure' to ensure that the Pod automatically restarts if it fails. This will help maintain the Pod's availability and reduce downtime.

- Redundancy and Scalability: Implement a Deployment or ReplicaSet to manage the Pod. This will provide redundancy by running multiple replicas of the Pod, ensuring that the Service remains available even if one Pod fails.

- Service Resilience: Consider using a LoadBalancer or Ingress to manage traffic to multiple Pods, enhancing the system's ability to handle failures and maintain service availability.

By addressing these issues, the system can improve its resilience against similar fault scenarios in the future, ensuring that the steady states are maintained even when faults are injected.

Improvement

To improve the resiliency of the system, I will replace the Pod with a Deployment to manage the Pod's lifecycle and provide redundancy. This will address both the restart policy issue and the single Pod deployment issue by ensuring that multiple replicas of the Pod are running and can automatically restart if they fail.

Modification_type: replace
File name: nginx/pod.yaml

Replace the Pod with a Deployment to manage the Pod's lifecycle and provide redundancy. This will ensure that multiple replicas of the Pod are running and can automatically restart if they fail.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-deployment
  labels:
    app: example
spec:
  replicas: 3
  selector:
    matchLabels:
      app: example
  template:
    metadata:
      labels:
        app: example
    spec:
      containers:
      - name: example-container
        image: nginx:1.17.1
        ports:
        - containerPort: 80
```

Experiment replanning

Failure scope of PodChaos
The previous K8s manifests targeted a single pod with the label 'app: example'. The current manifests have changed to a Deployment with 3 replicas, all having the same label 'app: example'. The original chaos experiment aimed to simulate a pod failure, which can still be achieved by targeting one of the pods in the Deployment. Therefore, the scope of the fault injection should remain the same, targeting pods with the label 'app: example' in the 'default' namespace. This ensures that the experiment's goal of simulating a pod failure is maintained, even with the change from a single pod to a Deployment with multiple replicas.

Failure scope of NetworkChaos
The previous K8s manifests had a single pod, and the NetworkChaos was targeting all pods with the label 'app: example' in the 'default' namespace. The current manifests have been updated to a Deployment with 3 replicas, but the label 'app: example' and the namespace 'default' remain the same. Therefore, the scope of the fault injection should remain unchanged, as it will now target all pods created by the Deployment, which aligns with the original goal of the chaos experiment to test the system's behavior under network stress.

VaC script for steady state #1
The previous unit test targets a Pod resource named 'example-pod'. In the reconfigured manifests, the Pod has been replaced by a Deployment named 'example-deployment' with 3 replicas. This change means that the unit test should now target the Deployment instead of a single Pod. The test should verify that at least one of the Pods created by the Deployment is running at least 90% of the time. Therefore, the unit test requires modification to account for the change from a Pod to a Deployment with multiple replicas. The test should be updated to check the status of the Pods created by the Deployment, ensuring that the steady state is maintained by the entire redundancy.
```
import os
import time
import argparse
from kubernetes import client, config
from unittest_base import K8sAPIBase

class
TestDeploymentRunningState(K8sAPIBase):
  def __init__(self):
    super().__init__()

  def check_deployment_pods_status(
    self, namespace, deployment_name
  ):
    try:
      pods = self.v1.list_namespaced_pod(
        namespace=namespace,
        label_selector=f'app={deployment_name}'
      ).items
      running_pods = [pod for pod in pods
      if pod.status.phase == 'Running']
      return len(running_pods) > 0
    except client.exceptions.ApiException
    as e:
      print(f"Exception when calling
      CoreV1Api->list_namespaced_pod: {e}")
      return False

  def test_deployment_running_state(self,
  duration):
    namespace = 'default'
    deployment_name = 'example'
    running_count = 0

    # Check the deployment pods status
    every second for the specified duration
    for _ in range(duration):
      if self.check_deployment_pods_status(
        namespace, deployment_name):
        running_count += 1
      time.sleep(1)

    # Calculate the running percentage
    running_percentage = (running_count /
    duration) * 100

    # Assert that the running percentage is
    at least 90%
    assert running_percentage >= 90,
    f"Deployment '{deployment_name}'
    running percentage is below threshold:
```

```
    {running_percentage}%"

    print(f"Deployment '{deployment_name}'
    running status checked {duration}
    times. Running percentage:
    {running_percentage}%.")


def main():
  parser = argparse.ArgumentParser(
    description='Test if a deployment has
    at least one pod running at least 90%
    of the time.')
  parser.add_argument(
    '--duration', type=int, default=5,
    help='Duration to check the deployment
    pods status in seconds.')
  args = parser.parse_args()

  test = TestDeploymentRunningState()
  test.test_deployment_running_state(
    args.duration)

if __name__ == '__main__':
  main()
```

VaC script for steady state #2
The target of the unit test is the service availability, which is associated with the Service resource in the Kubernetes manifests. In the previous and reconfigured manifests, the Service resource has not changed, so the endpoint and the service configuration remain the same. The unit test checks the availability of the service by sending HTTP requests to the service endpoint and verifying the response status. Since the Service resource has not been modified, the unit test does not require any changes. The reconfiguration of the Pod to a Deployment with 3 replicas is intended to improve the system's ability to meet the availability threshold, but it does not affect the unit test itself, which remains valid for checking the service's availability.

Summary of the completed CE cycle

The Chaos Engineering cycle begins with understanding the user inputs, which include Kubernetes manifests for a Pod and a Service. The Pod manifest defines a single Nginx container with a restart policy of 'Never', and the Service manifest exposes this Pod on port 80. Two main resiliency issues are identified: the Pod's restart policy and the lack of redundancy due to a single Pod deployment.

The hypothesis for the experiment is that the system's steady states will be maintained even when faults are injected. Two steady states are defined: the Pod should be running at least 90% of the time, and the Service should have 99.9% availability with a response status of 200.

The fault scenario simulates a cyber attack using Chaos Mesh, injecting PodChaos to kill the Pod and NetworkChaos to introduce network latency. The experiment is divided into three phases: pre-validation, fault-injection, and post-validation, each with specific tasks and durations to verify the system's behavior under stress.

In the first experiment attempt, the system fails to maintain the steady states during and after the fault injection. The Pod does not restart due to its 'Never' restart policy, and the Service becomes unavailable, confirming the identified issues. Recommendations include changing the Pod's restart policy and implementing a Deployment for redundancy.

After modifying the system by replacing the Pod with a Deployment, the second experime attempt is successful. All unit tests pass, indicating that the system maintains its steady states even when faults are injected, demonstrating improved resilience.

E.2 Reviews

Review by Engineer A

Overall: 4
- This is a critical issue because if the Pod fails, it will not restart, leading to potential downtime. If the single Pod fails, the service will be unavailable.
- After modifying the system by replacing the Pod with a Deployment, the second experiment attempt is successful. All unit tests pass, indicating that the system maintains its steady states even when faults are injected, demonstrating improved resilience.
- However, it did not provide significant insights for future cycles, such as testing different replica counts, resource usage or cost efficiency.

Hypothesis: 4
The hypothesis for the experiment is that the system's steady states will be maintained even when faults are injected. Two steady states are defined: the Pod should be running at least 90\% of the time, and the Service should have 99.9\% availability with a response status of 200.
- Steady state #1: example-pod-running
 + The first issue to address is the Pod's restart policy set to 'Never'. This is a critical issue because if the Pod fails, it will not restart, leading to potential downtime.
- Steady state #2: example-service-availability
 + The next issue to address is the 'Single Pod Deployment', which is related to the lack of redundancy. This is a significant issue because if the single Pod fails, the service will be unavailable.

Did not explore more failure scenarios, example: simultaneous Pod failure or network latency. (Only Failure senario: Cyber Attack)

Experiment: 4
The experiment plan correctly serves to validate the hypothesis
Time schedule
- Total experiment time: 60s
- Pre-validation Phase : 15s
 + We need to ensure that the system is in its expected steady state before we proceed with fault injection. Conduct two unit tests to verify the steady states: one for the pod's running status and another for the service's availability.
- Fault-injection Phase: 30s
 + Simulate a Cyber Attack by injecting two types of faults: PodChaos and NetworkChaos.
 + First, inject the PodChaos fault to simulate a pod failure. . This fault will be injected at the start of the phase (grace period of 0s) and will last for 10 seconds.
 + Next, inject the NetworkChaos fault to

simulate network latency. This fault will start at 10 seconds (aer the PodChaos fault ends) and will last for 20 seconds.
+ By staggering the faults and aligning the unit tests with the fault durations, we can effectively observe the system's behavior under each fault condition and verify if the steady states are maintained.
- Post-validation Phase: 15s
+ Pod must be running for the service to be available.

Additionally, it is set up considering an actual failure scenario.
- The fault scenario simulates a Cyber Attack using Chaos Mesh, injecting PodChaos to kill the Pod and NetworkChaos to introduce network latency.

Analysis: 4
The analysis reports correct and meaningful information:
- Pod Restart Policy and Single Pod Deployment: The 'fault-unittest-example-pod-running' test failed because the Pod was not found after the PodChaos fault was injected. This is directly related to the Pod's restart policy set to 'Never' in the nginx/pod.yaml manifest. When the Pod was killed, it did not restart, leading to a 404 error when attempting to read the Pod's status. This confirms the identified issue #0 (Pod Restart Policy) and issue #1 (Single Pod Deployment), where the lack of redundancy and automatic recovery mechanisms resulted in the Pod being unavailable.
- Service Availability: The 'fault-unittest-example-service-availability' test failed due to connection refusals, indicating that the service was unavailable. This failure is a consequence of the Pod being killed and not restarting, as the Service relies on the Pod to handle requests. The Service's selector in nginx/service.yaml targets Pods with the label 'app: example', and with the Pod being down, there were no endpoints available to serve the requests, leading to the connection refused errors.
- Post-Validation Failures: Both 'post-unittest-example-pod-running' and 'post-unittest-example-service-availability' tests failed, showing that the system did not recover to its steady state after the fault injection. The Pod remained unavailable, and consequently, the Service could not respond to HTTP requests. This further highlights the critical impact of the Pod's restart policy and the lack of redundancy.

Lacked detailed performance impact analysis, such as checking if the Pod was running at least 90\% of the time and if the Service maintained 99.9\% availability.

Meaningful insights for the improvement
- Pod Restart Policy: Change the Pod's restart policy from 'Never' to 'Always' or 'OnFailure' to ensure that the Pod automatically restarts if it fails. This will help maintain the Pod's availability and reduce downtime.
- Redundancy and Scalability: Implement a Deployment or ReplicaSet to manage the Pod. This will provide redundancy by running multiple replicas of the Pod, ensuring that the Service remains available even if one Pod fails.
- Service Resilience: Consider using a LoadBalancer or Ingress to manage traffic to multiple Pods, enhancing the system's ability to handle failures and maintain service availability.

Improvement: 5
Successfully fixed issues on the first attempt and enhanced resiliency and satisfied the hypothesis.
- In the first experiment attempt, the system fails to maintain the steady states during and after the fault injection. The Pod does not restart due to its 'Never' restart policy, and the Service becomes unavailable, confirming the identified issues.
- After modifying the system by replacing the Pod with a Deployment, the second experime attempt is successful. All unit tests pass, indicating that the system maintains its steady states even when faults are injected, demonstrating improved resilience.

Review by Engineer B (Translated)

Overall: 5
Reason for positive evaluation:
- The issue where the system failed to recover when a Pod went down was resolved. Meaningful insights and recommendations were obtained regarding service resilience.

The resolution of the system recovery issue and the insights gained on resilience contribute to a high evaluation.

Hypothesis: 4
Reason for positive evaluation:
- The summaries of each manifest file are accurate.
- The resiliency issues in the manifests are appropriately pointed out.
- The content of the image ('nginx') is explained.
- The assumed purpose of each manifest file's deployment is inferred correctly.
- The steady-state conditions, such as Pod uptime and HTTP request responsiveness, are properly identified.
- Service availability is considered with a commercial-grade mindset (e.g., 99.9% uptime), and HTTP response status code 200 is specified.

Room for improvement:
- The API version information in 'nginx/service.yaml' may not be necessary.
- Since 'nginx:1.17.1' is not the latest version, it would be better if there were comments expressing concern about potential vulnerabilities.
- Furthermore, incorporating tests from a security vulnerability perspective would be desirable.

The evaluation is high because the manifest summaries and resiliency issue identifications are appropriate. However, it is not a perfect score due to the inclusion of unnecessary information and the lack of comments on the outdated image.

Experiment: 5
Reason for positive evaluation:
- Pod status and service availability were checked during pre-validation, fault injection, and post-validation phases.

The evaluation is high because the test was conducted within the one-minute time constraint, while confirming Pod status and service availability.

Analysis: 5
Reason for positive evaluation:

- The occurrence of 404 errors after the fault and the lack of recovery were accurately analyzed.
- The three proposed recommendations are meaningful for improving resilience.

The evaluation is high because the post-fault situation was accurately analyzed, and valuable suggestions for enhancing resilience were provided.

Improvement: 5
Reason for positive evaluation:
- Replacing the Pod with a Deployment enables automatic restarts.
- Setting the number of replicas to 3 provides redundancy.

The evaluation is high because improvements were made by replacing the Pod with a Deployment and introducing redundancy.

Review by GPT-4o

Overall: 4
The Chaos Engineering cycle effectively identified and addressed critical issues in the system, leading to significant improvements in resilience.

Strengths:
- Comprehensive approach to identifying and addressing system weaknesses.
- Successfully improved system resilience through targeted improvements.
- Provided meaningful insights for future cycles.

Weaknesses:
- The cycle could have explored additional fault scenarios for a more comprehensive assessment.
- Limited exploration of potential secondary effects of the faults.
- The impact of improvements on overall system performance was not fully quantified.

Reason for the score:
The cycle fixed critical issues in the system and offered meaningful insights for the next cycle, though it could have been more comprehensive in its exploration of potential vulnerabilities.

Hypothesis: 4
The hypothesis aimed to ensure that the system maintains its steady states even when faults are injected, focusing on pod availability and service responsiveness.

Strengths:
- Clearly defined steady states for both pod and service availability.
- Utilized measurable outputs to verify steady states.
- Addressed critical issues like pod restart policy and single pod deployment.

Weaknesses:
- The hypothesis could have been more specific about the expected behavior under each fault condition.
- Lacked consideration for potential cascading failures beyond the immediate faults.
- Did not account for external dependencies that might affect the system's steady states.

Reason for the score:
The hypothesis was relevant and meaningful, addressing critical issues in the system. It led to system improvement and provided insights for future cycles.

Experiment: 4
The experiment was well-structured, with clear phases for pre-validation, fault-injection, and post-validation, all within a 60-second timeframe.

Strengths:
- Efficient time allocation across phases.
- Comprehensive fault-injection plan using PodChaos and NetworkChaos.
- Aligned unit tests with fault durations for effective observation.

Weaknesses:
- The experiment could have included more diverse fault scenarios to test other potential vulnerabilities.
- Limited time for each phase might not capture all transient issues.
- The experiment plan could have been more detailed in terms of expected outcomes for each fault.

Reason for the score:
The experiment plan correctly served to validate the hypothesis and was set up considering actual failure scenarios, though it could have been more comprehensive.

Analysis: 4
The analysis identified critical issues related to pod restart policy and single pod deployment, providing clear insights and recommendations for improvement.

Strengths:
- Thorough identification of issues and their impact on system behavior.
- Provided actionable recommendations for system improvement.
- Highlighted the importance of redundancy and automatic recovery mechanisms.

Weaknesses:
- The analysis could have explored potential secondary effects of the faults.
- Lacked quantitative data on the impact of network latency on service performance.
- Could have included a comparison with baseline performance metrics.

Reason for the score:
The analysis reported correct and meaningful information, providing insights for improvement, though it could have been more detailed in certain areas.

Improvement: 5
The improvement involved replacing the Pod with a Deployment to provide redundancy and ensure automatic recovery, successfully addressing the identified issues.

Strengths:
- Successfully changed the system to satisfy the hypothesis in the first attempt.
- Addressed both redundancy and restart policy issues effectively.
- Improved system resilience against similar fault scenarios.

Weaknesses:
- The improvement focused solely on the pod configuration, potentially overlooking other areas for enhancement.
- Could have included additional resilience measures, such as load balancing.
- The impact of the improvement on overall system performance was not quantified.

Reason for the score:

The improvement successfully changed the system to satisfy the hypothesis in the first attempt, addressing critical issues effectively.

Review by Claude Sonnet 3.5

Overall: 5
This Chaos Engineering cycle effectively identified and addressed critical resiliency issues in the system. It progressed from a well-defined hypothesis through a comprehensive experiment, insightful analysis, and successful improvement, resulting in a more resilient system.

Strengths:
- Clear progression through all phases of the Chaos Engineering cycle
- Effective use of Chaos Mesh for fault injection and automated testing
- Successful identification and resolution of critical system vulnerabilities
- Demonstrable improvement in system resilience, as evidenced by the second experiment run

Weaknesses:
- Limited scope of improvements, focusing mainly on Pod management
- Potential for more comprehensive testing of network-related issues
- Short duration of experiment phases may limit the depth of insights gained

Reason for the score:
The cycle fixes critical issues in the system, specifically addressing the Pod restart policy and lack of redundancy. It offers meaningful insights for future improvements, particularly in the areas of network resilience and more comprehensive testing strategies.

Hypothesis: 4
The hypothesis aims to verify that the system maintains its steady states even when subjected to a simulated cyber attack. Two steady states are defined: the 'example-pod' running status and the 'example-service' availability.

Strengths:
- Clear definition of steady states with measurable outputs
- Appropriate thresholds set for each steady state (90\% for pod running, 99.9\% for service availability)
- Well-defined fault scenario simulating a cyber attack using PodChaos and NetworkChaos
- Use of both Kubernetes API and k6 for comprehensive testing

Weaknesses:
- The hypothesis doesn't explicitly address the known system weaknesses (restart policy and single pod deployment)
- The steady states don't directly test the system's ability to recover from failures
- The hypothesis doesn't consider potential cascading effects of the simulated cyber attack

Reason for the score:
The hypothesis is relevant to the system and meaningful, addressing key aspects of system resilience. It leads to system improvement by revealing critical vulnerabilities. However, it could be more comprehensive in addressing known weaknesses and recovery capabilities.

Experiment: 4
The experiment is divided into three phases: pre-validation (15s), fault-injection (30s), and post-validation (15s). It uses Chaos Mesh to inject PodChaos and NetworkChaos faults, simulating a cyber attack. Unit tests are conducted before, during, and after fault injection to verify steady states.

Strengths:
- Well-structured experiment with clear phases and timing
- Use of both PodChaos and NetworkChaos to simulate a complex attack scenario
- Concurrent execution of faults and unit tests during fault injection
- Use of Chaos Mesh Workflow for automated experiment execution

Weaknesses:
- Short duration for pre and post-validation phases may not capture all system behaviors
- No explicit test for system recovery time after fault injection
- Limited variation in fault parameters (e.g., only one type of network chaos)

Reason for the score:
The experiment plan correctly serves to validate the hypothesis and is set up considering an actual failure scenario (cyber attack). It effectively uses Chaos Mesh to inject faults and conduct tests concurrently, providing a comprehensive view of system behavior under stress.

Analysis: 5
The analysis identified critical issues in the system's configuration, particularly the Pod's restart policy and lack of redundancy. It correctly linked these issues to the failures observed during the experiment, providing clear insights into the system's vulnerabilities.

Strengths:
- Detailed examination of each failed unit test
- Clear connection between observed failures and known system weaknesses
- Identification of cascading effects (Pod failure leading to service unavailability)
- Specific, actionable recommendations for improvement

Weaknesses:
- Limited analysis of the system's behavior during the NetworkChaos fault
- No quantitative analysis of failure rates or downtime duration
- Lack of discussion on potential false positives or negatives in the test results

Reason for the score:
The analysis reports correct and meaningful information, clearly linking experiment results to system weaknesses. It provides valuable insights for improvement, including specific recommendations to enhance system resilience.

Improvement: 5
The improvement phase replaced the single Pod definition with a Deployment, addressing both the restart policy issue and the lack of redundancy. This change successfully resolved the vulnerabilities identified in the analysis.

Strengths:
- Direct address of the two main issues (restart policy and single pod deployment)

- Use of a Deployment with multiple replicas for improved redundancy
- Simplicity of the solution, requiring changes to only one manifest
- Successful resolution of all failed unit tests in the second experiment run

Weaknesses:
- No explicit mention of changing the restart policy (though implicitly addressed by using a Deployment)
- No additional improvements to address potential network issues identified by NetworkChaos

Reason for the score:
The improvement successfully changes the system to satisfy the hypothesis in the first attempt. All unit tests passed in the second experiment run, demonstrating the effectiveness of the changes made.

Review by Gemini 1.5 Pro

Overall: 5
This Chaos Engineering cycle effectively identifies and addresses critical resiliency issues in the system. The cycle demonstrates a clear understanding of the system's weaknesses and utilizes appropriate tools and techniques to validate the hypothesis and implement improvements. The cycle is well-documented and provides valuable insights for future iterations.

Strengths:
- Clear identification of resiliency issues.
- Well-defined hypothesis and experiment plan.
- Effective use of Chaos Mesh for fault injection.
- Accurate analysis of experiment results.
- Successful implementation of improvements.

Weaknesses:
- The cycle could benefit from more comprehensive monitoring and data collection during the experiment. Gathering more data on resource utilization, error rates, and other relevant metrics would provide a more complete understanding of the system's behavior under stress.
- The cycle could be more robust by exploring a wider range of fault scenarios. While the chosen faults are relevant, considering other potential failures would provide a more comprehensive assessment of the system's resilience.
- The cycle could be more iterative by incorporating more feedback loops between the analysis and improvement phases. This would allow for more fine-tuning of the improvements and ensure that the system meets the desired resilience goals.
- The cycle could be more automated by integrating the Chaos Engineering experiments into the CI/CD pipeline. This would allow for continuous resilience testing and ensure that any changes to the system do not introduce new vulnerabilities.

Reason for the score:
The cycle fixes critical issues in the system and offers meaningful insights for the next cycle according to the experiments conducted

Hypothesis: 4
The hypothesis "The steady states of the system are maintained even when the fault scenario occurs" is well-defined and relevant to the identified resiliency issues. The steady states, 'example-pod-running' and 'example-service-availability', directly address the problems of the Pod's restart policy and the lack of redundancy.

Strengths:
- Clearly defines two relevant steady states.
- Directly addresses the identified resiliency issues.
- Provides measurable outputs for each steady state.
- Sets specific thresholds for each steady state.

Weaknesses:
- The hypothesis could be more specific about the expected behavior under the fault scenario. While it states that steady states should be maintained, it doesn't quantify the acceptable deviation from the normal operating conditions.
- The hypothesis could benefit from considering additional steady states, such as resource utilization or error rates, to provide a more comprehensive view of the system's health.
- The description of the steady states could be more concise.
- The hypothesis doesn't explicitly mention the expected duration of the steady state deviations during the fault injection.

Reason for the score:
The hypothesis is relevant to the system and meaningful. Additionally, the hypothesis leads to system improvement and offers meaningful insights for the next cycle.

Experiment: 4
The experiment plan is well-structured and effectively targets the identified weaknesses. The three-phase approach (pre-validation, fault-injection, post-validation) is appropriate, and the time allocation for each phase is reasonable. The use of Chaos Mesh to inject faults is a good choice, and the selected faults (PodChaos and NetworkChaos) are relevant to the scenario.

Strengths:
- Well-defined three-phase structure.
- Appropriate time allocation for each phase.
- Relevant fault selection (PodChaos and NetworkChaos).
- Use of Chaos Mesh for automated fault injection.
- Clear description of the fault injection sequence and unit tests.

Weaknesses:
- The experiment plan could benefit from more detailed monitoring metrics beyond just the steady states. Collecting data on resource utilization, error rates, and other relevant metrics would provide a more comprehensive understanding of the system's behavior under stress.
- The experiment plan could be more robust by including a wider range of fault scenarios. While the chosen faults are relevant, exploring other potential failures would provide a more complete picture of the system's resilience.
- The experiment plan could be improved by adding more details about the expected results of the unit tests during the fault injection phase. This would make it easier to interpret the results and draw conclusions about the system's behavior.

- The experiment plan could be more specific about the criteria for determining whether the system has returned to its steady state in the post-validation phase.

Reason for the score:
The experiment plan correctly serves to validate the hypothesis. Additionally, it is set up considering an actual failure scenario.

Analysis: 5
The analysis accurately identifies the root causes of the failed unit tests and provides specific recommendations for improvement. The analysis correctly correlates the observed failures with the Pod's restart policy and the lack of redundancy.

Strengths:
- Accurate identification of the root causes of failures.
- Clear correlation between failures and the identified resiliency issues.
- Specific and actionable recommendations for improvement.
- Well-organized and easy to understand.

Weaknesses:
- The analysis could be more data-driven. While it describes the failures, it doesn't provide specific metrics or data points to support the observations. Including data from the unit tests and Chaos Mesh would strengthen the analysis.
- The analysis could benefit from a more in-depth discussion of the system's behavior during the fault injection. Describing how the system responded to the faults and the specific impact on the steady states would provide more valuable insights.
- The analysis could be more comprehensive by considering the potential impact of the failures on downstream services or users. While it focuses on the system under test, considering the broader context would provide a more complete picture of the impact of the failures.
- The analysis could be improved by including a discussion of the limitations of the experiment and potential sources of error.

Reason for the score:
The analysis reports correct and meaningful information. Additioanlly, it provides some meaningful insights for the improvement.

Improvement: 5
The improvement successfully addresses the identified issues by replacing the Pod with a Deployment. This change introduces redundancy and automatic restart capabilities, directly addressing the weaknesses highlighted in the analysis. The improvement is implemented efficiently and effectively.

Strengths:
- Directly addresses the identified weaknesses.
- Introduces redundancy and automatic restart capabilities.
- Effective and efficient implementation.
- Uses a Deployment for robust Pod management.

Weaknesses:
- The improvement could be more comprehensive by addressing other potential weaknesses in the system, such as resource limits or network policies. While the implemented change is significant, considering other aspects of resilience would further strengthen the system.
- The improvement could be more robust by including automated testing to verify the changes. While the subsequent experiment confirms the improvement, incorporating automated tests would provide more confidence in the changes.
- The improvement could be more flexible by allowing for configuration of the number of replicas. While three replicas are a good starting point, providing a way to adjust the replica count based on the system's needs would be beneficial.
- The improvement could be more transparent by providing more details about the decision-making process behind the chosen solution. Explaining why a Deployment was chosen and the rationale behind the configuration would be helpful.

Reason for the score:
The improvement succesully changes the system to satisfy the hypothesis in the first attempt.

E.3 SockShop

E.3.1 Inputs

Instructions

- The Chaos-Engineering experiment must be completed within 1 minute.
- When using k6 in steady-state definition, always select a request URL from the following options (other requests are invalid):
1. http://front-end.sock-shop.svc.cluster.local/
2. http://front-end.sock-shop.svc.cluster.local/catalogue?size=10
3. http://front-end.sock-shop.svc.cluster.local/detail.html?id=<ID>
   Replace <ID> with an available ID: [03fef6ac-1896-4ce8-bd69-b798f85c6e0b, 3395a43e-2d88-40de-b95f-e00e1502085b, 510a0d7e-8e83-4193-b483-e27e09ddc34d, 808a2de1-1aaa-4c25-a9b9-6612e8f29a38, 819e1fbf-8b7e-4f6d-811f-693534916a8b, 837ab141-399e-4c1f-9abc-bace40296bac, a0a4f044-b040-410d-8ead-4de0446aec7e, d3588630-ad8e-49df-bbd7-3167f7efb246, zzz4f044-b040-410d-8ead-4de0446aec7e]
4. http://front-end.sock-shop.svc.cluster.local/category/
5. http://front-end.sock-shop.svc.cluster.local/category?tags=<TAG>
   Replace <TAG> with an available tag: [magic, action, blue, brown, black, sport, formal, red, green, skin, geek]
6. http://front-end.sock-shop.svc.cluster.local/basket.html

skaffold.yaml

```
apiVersion: skaffold/v3
kind: Config
metadata:
  name: sock-shop-app
manifests:
  rawYaml:
    - manifests/00-sock-shop-ns.yaml
    - manifests/01-carts-dep.yaml
    - manifests/02-carts-svc.yaml
    - manifests/03-carts-db-dep.yaml
    - manifests/04-carts-db-svc.yaml
    - manifests/05-catalogue-dep.yaml
```

```
    - manifests/06-catalogue-svc.yaml
    - manifests/07-catalogue-db-dep.yaml
    - manifests/08-catalogue-db-svc.yaml
    - manifests/09-front-end-dep.yaml
    - manifests/10-front-end-svc.yaml
    - manifests/11-orders-dep.yaml
    - manifests/12-orders-svc.yaml
    - manifests/13-orders-db-dep.yaml
    - manifests/14-orders-db-svc.yaml
    - manifests/15-payment-dep.yaml
    - manifests/16-payment-svc.yaml
    - manifests/17-queue-master-dep.yaml
    - manifests/18-queue-master-svc.yaml
    - manifests/19-rabbitmq-dep.yaml
    - manifests/20-rabbitmq-svc.yaml
    - manifests/21-session-db-dep.yaml
    - manifests/22-session-db-svc.yaml
    - manifests/23-shipping-dep.yaml
    - manifests/24-shipping-svc.yaml
    - manifests/25-user-dep.yaml
    - manifests/26-user-svc.yaml
    - manifests/27-user-db-dep.yaml
    - manifests/28-user-db-svc.yaml
```

manifests/00-sock-shop-ns.yaml

```
apiVersion: v1
kind: Namespace
metadata:
  name: sock-shop
```

manifests/01-carts-dep.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: carts
  labels:
    name: carts
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: carts
  template:
    metadata:
      labels:
        name: carts
    spec:
      containers:
      - name: carts
        image: weaveworksdemos/carts:0.4.8
        env:
         - name: JAVA_OPTS
           value: -Xms64m -Xmx128m -XX:+
           UseG1GC -Djava.security.egd=file
           :/dev/urandom -Dspring.zipkin.
           enabled=false
        resources:
          limits:
            cpu: 300m
            memory: 500Mi
          requests:
            cpu: 100m
            memory: 200Mi
        ports:
        - containerPort: 80
        securityContext:
          runAsNonRoot: true
          runAsUser: 10001
          capabilities:
            drop:
              - all
            add:
              - NET_BIND_SERVICE
          readOnlyRootFilesystem: true
        volumeMounts:
          - mountPath: /tmp
            name: tmp-volume
      volumes:
        - name: tmp-volume
          emptyDir:
            medium: Memory
      nodeSelector:
        beta.kubernetes.io/os: linux
```

manifests/02-carts-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: carts
  annotations:
        prometheus.io/scrape: 'true'
  labels:
    name: carts
  namespace: sock-shop
spec:
  ports:
    # the port that this service should
    serve on
  - port: 80
    targetPort: 80
  selector:
    name: carts
```

manifests/03-carts-db-dep.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: carts-db
  labels:
    name: carts-db
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: carts-db
  template:
    metadata:
      labels:
        name: carts-db
    spec:
      containers:
      - name: carts-db
        image: mongo
        ports:
        - name: mongo
          containerPort: 27017
        securityContext:
          capabilities:
            drop:
              - all
            add:
              - CHOWN
              - SETGID
              - SETUID
          readOnlyRootFilesystem: true
        volumeMounts:
        - mountPath: /tmp
          name: tmp-volume
      volumes:
        - name: tmp-volume
          emptyDir:
            medium: Memory
      nodeSelector:
        beta.kubernetes.io/os: linux
```

manifests/04-carts-db-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: carts-db
  labels:
    name: carts-db
  namespace: sock-shop
spec:
  ports:
    # the port that this service should
    serve on
  - port: 27017
    targetPort: 27017
  selector:
    name: carts-db
```

manifests/05-catalogue-dep.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: catalogue
  labels:
    name: catalogue
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: catalogue
  template:
    metadata:
      labels:
        name: catalogue
    spec:
      containers:
      - name: catalogue
        image: weaveworksdemos/catalogue
        :0.3.5
        command: ["/app"]
        args:
        - -port=80
        resources:
          limits:
            cpu: 200m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 100Mi
        ports:
        - containerPort: 80
        securityContext:
          runAsNonRoot: true
          runAsUser: 10001
          capabilities:
            drop:
              - all
            add:
              - NET_BIND_SERVICE
          readOnlyRootFilesystem: true
        livenessProbe:
          httpGet:
            path: /health
            port: 80
          initialDelaySeconds: 300
          periodSeconds: 3
        readinessProbe:
          httpGet:
            path: /health
            port: 80
          initialDelaySeconds: 180
          periodSeconds: 3
      nodeSelector:
        beta.kubernetes.io/os: linux
```

manifests/06-catalogue-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: catalogue
  annotations:
        prometheus.io/scrape: 'true'
  labels:
    name: catalogue
  namespace: sock-shop
spec:
  ports:
    # the port that this service should
    serve on
  - port: 80
    targetPort: 80
  selector:
    name: catalogue
```

manifests/06-catalogue-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: catalogue
  annotations:
        prometheus.io/scrape: 'true'
  labels:
    name: catalogue
  namespace: sock-shop
spec:
  ports:
    # the port that this service should
    serve on
  - port: 80
    targetPort: 80
  selector:
    name: catalogue
```

manifests/07-catalogue-db-dep.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: catalogue-db
  labels:
    name: catalogue-db
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: catalogue-db
  template:
    metadata:
      labels:
        name: catalogue-db
    spec:
      containers:
      - name: catalogue-db
        image: weaveworksdemos/catalogue-db
        :0.3.0
        env:
          - name: MYSQL_ROOT_PASSWORD
            value: fake_password
          - name: MYSQL_DATABASE
            value: socksdb
        ports:
        - name: mysql
          containerPort: 3306
      nodeSelector:
        beta.kubernetes.io/os: linux
```

manifests/08-catalogue-db-svc.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: catalogue-db
  labels:
    name: catalogue-db
  namespace: sock-shop
spec:
  ports:
    # the port that this service should
    serve on
  - port: 3306
    targetPort: 3306
  selector:
    name: catalogue-db
```

manifests/09-front-end-dep.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: front-end
  namespace: sock-shop
spec:
  replicas: 1
  selector:
    matchLabels:
      name: front-end
  template:
    metadata:
      labels:
        name: front-end
    spec:
      containers:
      - name: front-end
        image: weaveworksdemos/front-end
        :0.3.12
        resources:
          limits:
            cpu: 300m
            memory: 1000Mi
          requests:
            cpu: 100m
            memory: 300Mi
        ports:
        - containerPort: 8079
        env:
        - name: SESSION_REDIS
          value: "true"
        securityContext:
          runAsNonRoot: true
          runAsUser: 10001
          capabilities:
            drop:
              - all
          readOnlyRootFilesystem: true
        livenessProbe:
          httpGet:
            path: /
            port: 8079
          initialDelaySeconds: 300
          periodSeconds: 3
        readinessProbe:
          httpGet:
            path: /
            port: 8079
          initialDelaySeconds: 30
          periodSeconds: 3
      nodeSelector:
        beta.kubernetes.io/os: linux
```

manifests/10-front-end-svc.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: front-end
  annotations:
        prometheus.io/scrape: 'true'
  labels:
    name: front-end
  namespace: sock-shop
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 8079
    nodePort: 30001
  selector:
    name: front-end
```

manifests/11-orders-dep.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: orders
  labels:
    name: orders
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: orders
  template:
    metadata:
      labels:
        name: orders
    spec:
      containers:
      - name: orders
        image: weaveworksdemos/orders:0.4.7
        env:
         - name: JAVA_OPTS
           value: -Xms64m -Xmx128m -XX:+
           UseG1GC -Djava.security.egd=file
           :/dev/urandom -Dspring.zipkin.
           enabled=false
        resources:
          limits:
            cpu: 500m
            memory: 500Mi
          requests:
            cpu: 100m
            memory: 300Mi
        ports:
        - containerPort: 80
        securityContext:
          runAsNonRoot: true
          runAsUser: 10001
          capabilities:
            drop:
              - all
            add:
              - NET_BIND_SERVICE
          readOnlyRootFilesystem: true
        volumeMounts:
        - mountPath: /tmp
          name: tmp-volume
      volumes:
        - name: tmp-volume
          emptyDir:
            medium: Memory
      nodeSelector:
        beta.kubernetes.io/os: linux
```

manifests/12-orders-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: orders
  annotations:
        prometheus.io/scrape: 'true'
  labels:
    name: orders
  namespace: sock-shop
spec:
  ports:
    # the port that this service should
    serve on
  - port: 80
    targetPort: 80
  selector:
    name: orders
```

manifests/13-orders-db-dep.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: orders-db
  labels:
    name: orders-db
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: orders-db
  template:
    metadata:
      labels:
        name: orders-db
    spec:
      containers:
      - name: orders-db
        image: mongo
        ports:
        - name: mongo
          containerPort: 27017
        securityContext:
          capabilities:
            drop:
              - all
            add:
              - CHOWN
              - SETGID
              - SETUID
          readOnlyRootFilesystem: true
        volumeMounts:
        - mountPath: /tmp
          name: tmp-volume
      volumes:
        - name: tmp-volume
          emptyDir:
            medium: Memory
      nodeSelector:
        beta.kubernetes.io/os: linux
```

manifests/14-orders-db-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: orders-db
  labels:
    name: orders-db
  namespace: sock-shop
spec:
  ports:
    # the port that this service should
    serve on
  - port: 27017
    targetPort: 27017
  selector:
    name: orders-db
```

manifests/15-payment-dep.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: payment
  labels:
    name: payment
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: payment
  template:
    metadata:
      labels:
        name: payment
    spec:
      containers:
      - name: payment
        image: weaveworksdemos/payment:0.4.3
        resources:
          limits:
            cpu: 200m
            memory: 200Mi
          requests:
            cpu: 99m
            memory: 100Mi
        ports:
        - containerPort: 80
        securityContext:
          runAsNonRoot: true
          runAsUser: 10001
          capabilities:
            drop:
              - all
            add:
              - NET_BIND_SERVICE
          readOnlyRootFilesystem: true
        livenessProbe:
          httpGet:
            path: /health
            port: 80
          initialDelaySeconds: 300
          periodSeconds: 3
        readinessProbe:
          httpGet:
            path: /health
            port: 80
          initialDelaySeconds: 180
          periodSeconds: 3
      nodeSelector:
        beta.kubernetes.io/os: linux
```

manifests/16-payment-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: payment
  annotations:
        prometheus.io/scrape: 'true'
  labels:
    name: payment
  namespace: sock-shop
spec:
  ports:
    # the port that this service should
    serve on
  - port: 80
    targetPort: 80
  selector:
```

```
    name: payment
```

manifests/17-queue-master-dep.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: queue-master
  labels:
    name: queue-master
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: queue-master
  template:
    metadata:
      labels:
        name: queue-master
    spec:
      containers:
      - name: queue-master
        image: weaveworksdemos/queue-master
        :0.3.1
        env:
         - name: JAVA_OPTS
           value: -Xms64m -Xmx128m -XX:+
           UseG1GC -Djava.security.egd=file
           :/dev/urandom -Dspring.zipkin.
           enabled=false
        resources:
          limits:
            cpu: 300m
            memory: 500Mi
          requests:
            cpu: 100m
            memory: 300Mi
        ports:
        - containerPort: 80
      nodeSelector:
        beta.kubernetes.io/os: linux
```

manifests/18-queue-master-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: queue-master
  annotations:
        prometheus.io/scrape: 'true'
  labels:
    name: queue-master
  namespace: sock-shop
spec:
  ports:
    # the port that this service should
    serve on
  - port: 80
    targetPort: 80
  selector:
    name: queue-master
```

manifests/19-rabbitmq-dep.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rabbitmq
  labels:
    name: rabbitmq
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: rabbitmq
  template:
    metadata:
      labels:
        name: rabbitmq
      annotations:
        prometheus.io/scrape: "false"
    spec:
      containers:
      - name: rabbitmq
        image: rabbitmq:3.6.8-management
        ports:
        - containerPort: 15672
          name: management
        - containerPort: 5672
          name: rabbitmq
        securityContext:
          capabilities:
            drop:
              - all
            add:
              - CHOWN
              - SETGID
              - SETUID
              - DAC_OVERRIDE
          readOnlyRootFilesystem: true
      - name: rabbitmq-exporter
        image: kbudde/rabbitmq-exporter
        ports:
        - containerPort: 9090
          name: exporter
      nodeSelector:
        beta.kubernetes.io/os: linux
```

manifests/20-rabbitmq-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: rabbitmq
  annotations:
        prometheus.io/scrape: 'true'
        prometheus.io/port: '9090'
  labels:
    name: rabbitmq
  namespace: sock-shop
spec:
  ports:
    # the port that this service should
    serve on
  - port: 5672
    name: rabbitmq
    targetPort: 5672
  - port: 9090
    name: exporter
    targetPort: exporter
    protocol: TCP
  selector:
    name: rabbitmq
```

manifests/21-session-db-dep.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: session-db
  labels:
    name: session-db
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: session-db
  template:
    metadata:
```

```
      labels:
        name: session-db
      annotations:
        prometheus.io.scrape: "false"
    spec:
      containers:
      - name: session-db
        image: redis:alpine
        ports:
        - name: redis
          containerPort: 6379
        securityContext:
          capabilities:
            drop:
              - all
            add:
              - CHOWN
              - SETGID
              - SETUID
          readOnlyRootFilesystem: true
      nodeSelector:
        beta.kubernetes.io/os: linux
```

manifests/22-session-db-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: session-db
  labels:
    name: session-db
  namespace: sock-shop
spec:
  ports:
    # the port that this service should
    serve on
  - port: 6379
    targetPort: 6379
  selector:
    name: session-db
```

manifests/23-shipping-dep.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: shipping
  labels:
    name: shipping
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: shipping
  template:
    metadata:
      labels:
        name: shipping
    spec:
      containers:
      - name: shipping
        image: weaveworksdemos/shipping
        :0.4.8
        env:
         - name: ZIPKIN
           value: zipkin.jaeger.svc.cluster.
           local
         - name: JAVA_OPTS
           value: -Xms64m -Xmx128m -XX:+
           UseG1GC -Djava.security.egd=file
           :/dev/urandom -Dspring.zipkin.
           enabled=false
        resources:
          limits:
            cpu: 300m
            memory: 500Mi
          requests:
            cpu: 100m
            memory: 300Mi
        ports:
        - containerPort: 80
        securityContext:
          runAsNonRoot: true
          runAsUser: 10001
          capabilities:
            drop:
              - all
            add:
              - NET_BIND_SERVICE
          readOnlyRootFilesystem: true
        volumeMounts:
        - mountPath: /tmp
          name: tmp-volume
      volumes:
        - name: tmp-volume
          emptyDir:
            medium: Memory
      nodeSelector:
        beta.kubernetes.io/os: linux
```

manifests/24-shipping-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: shipping
  annotations:
        prometheus.io/scrape: 'true'
  labels:
    name: shipping
  namespace: sock-shop
spec:
  ports:
    # the port that this service should
    serve on
  - port: 80
    targetPort: 80
  selector:
    name: shipping
```

manifests/25-user-dep.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user
  labels:
    name: user
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: user
  template:
    metadata:
      labels:
        name: user
    spec:
      containers:
      - name: user
        image: weaveworksdemos/user:0.4.7
        resources:
          limits:
            cpu: 300m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 100Mi
        ports:
        - containerPort: 80
        env:
        - name: mongo
          value: user-db:27017
        securityContext:
```

```
          runAsNonRoot: true
          runAsUser: 10001
          capabilities:
            drop:
              - all
            add:
              - NET_BIND_SERVICE
          readOnlyRootFilesystem: true
        livenessProbe:
          httpGet:
            path: /health
            port: 80
          initialDelaySeconds: 300
          periodSeconds: 3
        readinessProbe:
          httpGet:
            path: /health
            port: 80
          initialDelaySeconds: 180
          periodSeconds: 3
      nodeSelector:
        beta.kubernetes.io/os: linux
```

manifests/26-user-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: user
  annotations:
        prometheus.io/scrape: 'true'
  labels:
    name: user
  namespace: sock-shop
spec:
  ports:
    # the port that this service should
    serve on
  - port: 80
    targetPort: 80
  selector:
    name: user
```

manifests/27-user-db-dep.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-db
  labels:
    name: user-db
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: user-db
  template:
    metadata:
      labels:
        name: user-db
    spec:
      containers:
      - name: user-db
        image: weaveworksdemos/user-db:0.3.0
        ports:
        - name: mongo
          containerPort: 27017
        securityContext:
          capabilities:
            drop:
              - all
            add:
              - CHOWN
              - SETGID
              - SETUID
          readOnlyRootFilesystem: true
        volumeMounts:
          - mountPath: /tmp
            name: tmp-volume
      volumes:
        - name: tmp-volume
          emptyDir:
            medium: Memory
      nodeSelector:
        beta.kubernetes.io/os: linux
```

manifests/28-user-db-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: user-db
  labels:
    name: user-db
  namespace: sock-shop
spec:
  ports:
    # the port that this service should
    serve on
  - port: 27017
    targetPort: 27017
  selector:
    name: user-db
```

E.3.2 CE cycle completed by CHAOSEATER

Preprocessing

```
Summaries of each manifest
sock-shop-2/manifests/00-sock-shop-ns.yaml
- This manifest defines a Kubernetes
Namespace.
- The Namespace is named 'sock-shop'.
- Namespaces are used to organize and
manage resources in a Kubernetes cluster.

sock-shop-2/manifests/01-carts-dep.yaml
- This manifest defines a Deployment in
Kubernetes.
- The Deployment is named 'carts' and is
located in the 'sock-shop' namespace.
- It specifies that there should be 2
replicas of the 'carts' application running.
- The Deployment uses the Docker image
'weaveworksdemos/carts:0.4.8'.
- Environment variables are set for Java
options to optimize memory usage and disable
  certain features.
- Resource limits and requests are defined,
with a maximum of 300m CPU and 500Mi
  memory, and a minimum of 100m CPU and
  200Mi memory.
- The application listens on port 80 within
the container.
- Security settings ensure the container
runs as a non-root user with specific
  capabilities and a read-only root
  filesystem.
- A temporary volume is mounted at '/tmp'
using an in-memory empty directory.
- The Deployment is scheduled to run on
nodes with the Linux operating system.

sock-shop-2/manifests/02-carts-svc.yaml
- This manifest defines a Kubernetes
Service.
- The Service is named 'carts'.
- It is annotated to enable Prometheus
scraping with 'prometheus.io/scrape: true'.
- The Service is labeled with 'name: carts'.
- It is deployed in the 'sock-shop'
namespace.
- The Service exposes port 80 and directs
traffic to the same port on the selected
  pods.
```

```
- It uses a selector to target pods with
the label 'name: carts'.

sock-shop-2/manifests/03-carts-db-dep.yaml
- This manifest defines a Deployment in
Kubernetes.
- The Deployment is named 'carts-db' and is
located in the 'sock-shop' namespace.
- It specifies that there should be 2
replicas of the 'carts-db' pod running.
- The pods are selected based on the label
'name: carts-db'.
- Each pod runs a single container using
the 'mongo' image.
- The container exposes port 27017, which
is the default port for MongoDB.
- Security settings are applied to drop all
capabilities and only add CHOWN, SETGID,
  and SETUID.
- The root filesystem of the container is
set to read-only for security purposes.
- A temporary volume is mounted at '/tmp'
using an in-memory emptyDir volume.
- The pods are scheduled to run on nodes
with the operating system labeled as
'linux'.

sock-shop-2/manifests/04-carts-db-svc.yaml
- This manifest defines a Kubernetes
Service.
- The Service is named 'carts-db'.
- It is labeled with 'name: carts-db'.
- The Service is created in the 'sock-shop'
namespace.
- It exposes port 27017 and directs traffic
to the same port on the target pods.
- The Service selects pods with the label
'name: carts-db' to route traffic to them.

sock-shop-2/manifests/05-catalogue-dep.yaml
- This manifest defines a Deployment in
Kubernetes.
- The Deployment is named 'catalogue' and
is part of the 'sock-shop' namespace.
- It specifies that there should be 2
replicas of the 'catalogue' application
running.
- The Deployment uses the Docker image
'weaveworksdemos/catalogue:0.3.5'.
- The application runs with the command
'/app' and listens on port 80.
- Resource limits are set to 200m CPU and
200Mi memory, with requests for 100m CPU and
  100Mi memory.
- The container is configured to run as a
non-root user with user ID 10001.
- Security settings include dropping all
capabilities except 'NET_BIND_SERVICE' and
  using a read-only root filesystem.
- Liveness and readiness probes are
configured to check the '/health' endpoint
on port
  80, with initial delays of 300 and 180
  seconds respectively.
- The Deployment is scheduled to run on
nodes with the Linux operating system.

sock-shop-2/manifests/06-catalogue-svc.yaml
- This manifest defines a Kubernetes
Service.
- The Service is named 'catalogue'.
- It is annotated to enable Prometheus
scraping for monitoring purposes.
- The Service is labeled with 'name:
catalogue'.
- It is deployed in the 'sock-shop'
namespace.
- The Service exposes port 80 and directs
traffic to the same port on the selected
  pods.
- It uses a selector to target pods with
the label 'name: catalogue'.

sock-shop-2/manifests/07-catalogue-db-dep.yaml
- This manifest defines a Deployment in
Kubernetes.
- The Deployment is named 'catalogue-db'.
- It is located in the 'sock-shop'
namespace.
- The Deployment will create 2 replicas of
the pod.
- Each pod will run a container from the
image 'weaveworksdemos/catalogue-db:0.3.0'.
- The container is configured with
environment variables for MySQL, including
a root
  password and database name.
- The container exposes port 3306, which is
commonly used for MySQL.
- The pods are scheduled to run on nodes
with the Linux operating system.

sock-shop-2/manifests/08-catalogue-db-svc.yaml
- This manifest defines a Kubernetes
Service.
- The Service is named 'catalogue-db'.
- It is associated with the 'sock-shop'
namespace.
- The Service listens on port 3306 and
forwards traffic to the same port on the
target
  pods.
- It uses a selector to target pods with
the label 'name: catalogue-db'.

sock-shop-2/manifests/09-front-end-dep.yaml
- This manifest defines a Deployment in
Kubernetes.
- The Deployment is named 'front-end' and
is located in the 'sock-shop' namespace.
- It specifies that there should be 1
replica of the front-end application
running.
- The Deployment uses a selector to match
pods with the label 'name: front-end'.
- The pod template includes a single
container named 'front-end'.
- The container uses the image
'weaveworksdemos/front-end:0.3.12'.
- Resource limits are set for the
container: 300m CPU and 1000Mi memory.
- Resource requests are set for the
container: 100m CPU and 300Mi memory.
- The container exposes port 8079.
- An environment variable 'SESSION_REDIS'
is set to 'true'.
- Security context is configured to run the
container as a non-root user with user ID
  10001.
- All Linux capabilities are dropped, and
the root filesystem is set to read-only.
- A liveness probe is configured to check
the '/' path on port 8079, with an initial
  delay of 300 seconds and a period of 3
  seconds.
- A readiness probe is also configured to
check the '/' path on port 8079, with an
  initial delay of 30 seconds and a period
  of 3 seconds.
- The node selector ensures that the pod
runs on nodes with the operating system
  labeled as Linux.

sock-shop-2/manifests/10-front-end-svc.yaml
- This manifest defines a Kubernetes
Service.
- The Service is named 'front-end'.
- It is located in the 'sock-shop'
namespace.
- The Service type is 'NodePort', which
exposes the service on each Node's IP at a
  static port.
- It listens on port 80 and forwards
traffic to target port 8079 on the pods.
- The nodePort is set to 30001, allowing
external access to the service.
- The Service is configured to be scraped
by Prometheus for monitoring, as indicated
  by the annotation 'prometheus.io/scrape:
  true'.
```

- It selects pods with the label 'name: front-end' to route traffic to.

sock-shop-2/manifests/11-orders-dep.yaml
- This manifest defines a Deployment in Kubernetes.
- The Deployment is named 'orders' and is located in the 'sock-shop' namespace.
- It specifies that there should be 2 replicas of the 'orders' application running.
- The Deployment uses the 'weaveworksdemos/orders:0.4.7' Docker image for the container.
- Environment variables are set for Java options to optimize memory usage and disable certain features.
- Resource limits and requests are defined, with a maximum of 500m CPU and 500Mi memory, and a minimum of 100m CPU and 300Mi memory.
- The container listens on port 80.
- Security context is configured to run the container as a non-root user with specific capabilities and a read-only root filesystem.
- A temporary volume is mounted at '/tmp' using an in-memory empty directory.
- The Deployment is scheduled to run on nodes with the Linux operating system.

sock-shop-2/manifests/12-orders-svc.yaml
- This manifest defines a Kubernetes Service.
- The Service is named 'orders'.
- It is annotated to enable Prometheus scraping with 'prometheus.io/scrape: true'.
- The Service is labeled with 'name: orders'.
- It is deployed in the 'sock-shop' namespace.
- The Service exposes port 80 and directs traffic to the same port on the target pods.
- It uses a selector to match pods with the label 'name: orders'.

sock-shop-2/manifests/13-orders-db-dep.yaml
- This manifest defines a Deployment in Kubernetes.
- The Deployment is named 'orders-db' and is located in the 'sock-shop' namespace.
- It specifies 2 replicas of the 'orders-db' pod to be created.
- The pods are labeled with 'name: orders-db' for identification and selection.
- Each pod runs a single container using the 'mongo' image.
- The container exposes port 27017, which is the default port for MongoDB.
- Security settings are applied to drop all capabilities and add only CHOWN, SETGID, and SETUID.
- The root filesystem of the container is set to read-only for security purposes.
- A temporary volume is mounted at '/tmp' using an in-memory emptyDir volume.
- The pods are scheduled to run on nodes with the operating system labeled as 'linux'.

sock-shop-2/manifests/14-orders-db-svc.yaml
- This manifest defines a Kubernetes Service.
- The Service is named 'orders-db'.
- It is located in the 'sock-shop' namespace.
- The Service is configured to expose port 27017.
- It targets the same port (27017) on the pods it selects.
- The Service uses a selector to match pods with the label 'name: orders-db'.

sock-shop-2/manifests/15-payment-dep.yaml
- This manifest defines a Deployment in Kubernetes.
- The Deployment is named 'payment' and is part of the 'sock-shop' namespace.
- It specifies that there should be 2 replicas of the 'payment' application running.
- The Deployment uses the Docker image 'weaveworksdemos/payment:0.4.3'.
- Resource limits are set for the container, with a maximum of 200m CPU and 200Mi memory, and requests for 99m CPU and 100Mi memory.
- The container listens on port 80.
- Security settings ensure the container runs as a non-root user with user ID 10001, drops all capabilities except 'NET_BIND_SERVICE', and uses a read-only root filesystem.
- Liveness and readiness probes are configured to check the '/health' endpoint on port 80, with initial delays of 300 and 180 seconds respectively, and a period of 3 seconds.
- The Deployment is scheduled to run on nodes with the Linux operating system.

sock-shop-2/manifests/16-payment-svc.yaml
- This manifest defines a Kubernetes Service.
- The Service is named 'payment'.
- It is annotated for Prometheus scraping, which means it is set up for monitoring.
- The Service is labeled with 'name: payment'.
- It is deployed in the 'sock-shop' namespace.
- The Service exposes port 80 and directs traffic to the same port on the selected pods.
- The Service selects pods with the label 'name: payment'.

sock-shop-2/manifests/17-queue-master-dep.yaml
- This manifest defines a Deployment in Kubernetes.
- The Deployment is named 'queue-master' and is located in the 'sock-shop' namespace.
- It specifies that there should be 2 replicas (instances) of the 'queue-master' application running.
- The Deployment uses a container image 'weaveworksdemos/queue-master:0.3.1'.
- Environment variables are set for the container, including Java options for memory management and garbage collection.
- Resource limits and requests are defined, with a CPU limit of 300m and memory limit of 500Mi, and requests for 100m CPU and 300Mi memory.
- The container exposes port 80 for network traffic.
- The Deployment is configured to run on nodes with the Linux operating system.

sock-shop-2/manifests/18-queue-master-svc.yaml
- This manifest defines a Kubernetes Service.
- The Service is named 'queue-master'.
- It is annotated for Prometheus scraping with 'prometheus.io/scrape: true'.
- The Service is labeled with 'name: queue-master'.
- It is deployed in the 'sock-shop' namespace.
- The Service exposes port 80 and directs traffic to the same port on the selected pods.
- It selects pods with the label 'name: queue-master' to route traffic to.

sock-shop-2/manifests/19-rabbitmq-dep.yaml
- This manifest defines a Deployment for

RabbitMQ in Kubernetes.
- It is set to run in the 'sock-shop' namespace.
- The Deployment is named 'rabbitmq' and is labeled accordingly.
- It specifies 2 replicas, meaning there will be 2 instances of RabbitMQ running.
- The Deployment uses a selector to match pods with the label 'name: rabbitmq'.
- The pod template includes two containers: one for RabbitMQ and another for a RabbitMQ exporter.
- The RabbitMQ container uses the image 'rabbitmq:3.6.8-management'.
- It exposes two ports: 15672 for management and 5672 for RabbitMQ operations.
- Security context is set to drop all capabilities and add specific ones like CHOWN, SETGID, SETUID, and DAC_OVERRIDE.
- The root filesystem is set to read-only for security purposes.
- The RabbitMQ exporter container uses the image 'kbudde/rabbitmq-exporter' and exposes port 9090.
- The Deployment is configured to run on nodes with the label 'beta.kubernetes.io/os: linux'.
- Annotations are set to prevent Prometheus from scraping metrics from this deployment.

sock-shop-2/manifests/20-rabbitmq-svc.yaml
- This manifest defines a Kubernetes Service.
- The Service is named 'rabbitmq'.
- It is annotated for Prometheus scraping on port 9090.
- The Service is labeled with 'name: rabbitmq'.
- It is deployed in the 'sock-shop' namespace.
- The Service exposes two ports: 5672 for RabbitMQ and 9090 for an exporter.
- The protocol used for the ports is TCP.
- The Service selects pods with the label 'name: rabbitmq'.

sock-shop-2/manifests/21-session-db-dep.yaml
- This manifest defines a Deployment in Kubernetes.
- The Deployment is named 'session-db' and is located in the 'sock-shop' namespace.
- It specifies that there should be 2 replicas of the 'session-db' pod running.
- The pods are selected based on the label 'name: session-db'.
- Each pod runs a single container using the 'redis' image.
- The container exposes port 6379, which is commonly used by Redis.
- Security settings are applied to drop all capabilities and only add CHOWN, SETGID, and SETUID, with a read-only root filesystem for enhanced security.
- The pods are scheduled to run on nodes with the operating system labeled as Linux.

sock-shop-2/manifests/22-session-db-svc.yaml
- This manifest defines a Kubernetes Service.
- The Service is named 'session-db'.
- It is located in the 'sock-shop' namespace.
- The Service is configured to expose port 6379.
- It targets the same port (6379) on the selected pods.
- The Service uses a selector to match pods with the label 'name: session-db'.

sock-shop-2/manifests/23-shipping-dep.yaml
- This manifest defines a Deployment in Kubernetes.
- The Deployment is named 'shipping' and is located in the 'sock-shop' namespace.
- It specifies that there should be 2 replicas of the 'shipping' application running.
- The Deployment uses the Docker image 'weaveworksdemos/shipping:0.4.8'.
- Environment variables are set for the application, including 'ZIPKIN' and 'JAVA_OPTS'.
- Resource limits and requests are defined, with limits set to 300m CPU and 500Mi memory, and requests set to 100m CPU and 300Mi memory.
- The application listens on port 80.
- Security context is configured to run the container as a non-root user with user ID 10001, and it drops all capabilities except 'NET_BIND_SERVICE'.
- The root filesystem is set to be read-only.
- A temporary volume is mounted at '/tmp', using an in-memory emptyDir volume.
- The Deployment is scheduled to run on nodes with the operating system labeled as 'linux'.

sock-shop-2/manifests/24-shipping-svc.yaml
- This is a Kubernetes Service manifest.
- The service is named 'shipping'.
- It is annotated for Prometheus scraping with 'prometheus.io/scrape: true'.
- The service is labeled with 'name: shipping'.
- It is deployed in the 'sock-shop' namespace.
- The service exposes port 80 and directs traffic to the same port on the selected pods.
- It selects pods with the label 'name: shipping' to route traffic to.

sock-shop-2/manifests/25-user-dep.yaml
- This manifest defines a Deployment in Kubernetes.
- The Deployment is named 'user' and is located in the 'sock-shop' namespace.
- It specifies that there should be 2 replicas of the 'user' application running.
- The Deployment uses the Docker image 'weaveworksdemos/user:0.4.7'.
- Resource limits are set for the container: 300m CPU and 200Mi memory, with requests for 100m CPU and 100Mi memory.
- The container listens on port 80.
- An environment variable 'mongo' is set with the value 'user-db:27017'.
- Security context is configured to run the container as a non-root user with user ID 10001.
- The container has a read-only root filesystem and drops all capabilities except 'NET_BIND_SERVICE'.
- Liveness and readiness probes are configured to check the '/health' endpoint on port 80.
- The liveness probe starts after 300 seconds and checks every 3 seconds.
- The readiness probe starts after 180 seconds and checks every 3 seconds.
- The Deployment is scheduled to run on nodes with the Linux operating system.

sock-shop-2/manifests/26-user-svc.yaml
- This manifest defines a Kubernetes Service.
- The Service is named 'user'.
- It is annotated for Prometheus scraping, which means it is set up for monitoring.
- The Service is labeled with 'name: user'.
- It is deployed in the 'sock-shop' namespace.
- The Service listens on port 80 and forwards traffic to the same port on the selected

pods.
- The Service selects pods with the label 'name: user'.

sock-shop-2/manifests/27-user-db-dep.yaml
- This manifest defines a Deployment in Kubernetes.
- The Deployment is named 'user-db' and is located in the 'sock-shop' namespace.
- It specifies that there should be 2 replicas of the 'user-db' pod running.
- The pods are selected based on the label 'name: user-db'.
- Each pod runs a single container using the image 'weaveworksdemos/user-db:0.3.0'.
- The container exposes port 27017, labeled as 'mongo', which is typically used for MongoDB.
- Security settings are applied to drop all capabilities and only add CHOWN, SETGID, and SETUID, with a read-only root filesystem.
- A temporary volume is mounted at '/tmp' using an in-memory emptyDir volume.
- The pods are scheduled to run on nodes with the operating system labeled as 'linux'.

sock-shop-2/manifests/28-user-db-svc.yaml
- This manifest defines a Kubernetes Service.
- The Service is named 'user-db'.
- It is labeled with 'name: user-db'.
- The Service is created in the 'sock-shop' namespace.
- It exposes port 27017 and directs traffic to the same port on the target pods.
- The Service selects pods with the label 'name: user-db' to route traffic to them.

Resiliency issues/weaknesses in the manifests
Issue #0: Missing Resource Requests
- details: Pods may not get scheduled if the cluster is under resource pressure, leading to potential downtime.
- manifests having the issues: [
  'sock-shop-2/manifests/03-carts-db-dep.yaml',
  'sock-shop-2/manifests/07-catalogue-db-dep.yaml',
  'sock-shop-2/manifests/13-orders-db-dep.yaml',
  'sock-shop-2/manifests/19-rabbitmq-dep.yaml',
  'sock-shop-2/manifests/21-session-db-dep.yaml',
  'sock-shop-2/manifests/27-user-db-dep.yaml'
  ]
- problematic config: The deployments for carts-db, catalogue-db, orders-db, rabbitmq, session-db, and user-db do not specify resource requests.

Issue #1: Single Replica Deployment
- details: The front-end deployment has only one replica, which can lead to downtime if the pod fails.
- manifests having the issues: ['sock-shop-2/manifests/09-front-end-dep.yaml']
- problematic config: spec.replicas: 1

Issue #2: Missing Liveness and Readiness Probes
- details: Without liveness and readiness probes, Kubernetes cannot determine the health of the containers, which may lead to serving traffic to unhealthy pods.
- manifests having the issues: [
  'sock-shop-2/manifests/01-carts-dep.yaml',
  'sock-shop-2/manifests/03-carts-db-dep.yaml',
  'sock-shop-2/manifests/05-catalogue-dep.yaml',
  'sock-shop-2/manifests/07-catalogue-db-dep.yaml',
  'sock-shop-2/manifests/11-orders-dep.yaml',
  'sock-shop-2/manifests/13-orders-db-dep.yaml',
  'sock-shop-2/manifests/15-payment-dep.yaml',
  'sock-shop-2/manifests/17-queue-master-dep.yaml',
  'sock-shop-2/manifests/19-rabbitmq-dep.yaml',
  'sock-shop-2/manifests/21-session-db-dep.yaml',
  'sock-shop-2/manifests/23-shipping-dep.yaml',
  'sock-shop-2/manifests/25-user-dep.yaml',
  'sock-shop-2/manifests/27-user-db-dep.yaml'
  ]
- problematic config: The deployments for carts, carts-db, catalogue, catalogue-db, orders, orders-db, payment, queue-master, rabbitmq, session-db, shipping, user, and user-db do not have liveness or readiness probes configured.

Issue #3: Hardcoded Environment Variables
- details: Hardcoded environment variables, such as passwords, can lead to security vulnerabilities.
- manifests having the issues: [
  'sock-shop-2/manifests/07-catalogue-db-dep.yaml']
- problematic config: env: MYSQL_ROOT_PASSWORD is hardcoded with value: fake_password.

Application of the manifests
Thoughts:
The manifests provided are part of a well-known microservices demo application called 'Sock Shop'. This application is designed to demonstrate the use of microservices architecture and Kubernetes orchestration. The application simulates an e-commerce website for selling socks, with various services such as carts, catalogue, orders, payment, and user management. The file names, labels, and Docker images used in the manifests (e.g., 'weaveworksdemos') are indicative of the Sock Shop application, which is a common reference application in the Kubernetes community.

Assumed application:
Sock Shop is an e-commerce application that simulates an online sock store. It offers users the ability to browse a catalogue of socks, add items to a cart, place orders, and manage user accounts. The application is composed of multiple microservices, each responsible for a specific function, such as managing the product catalogue, handling user sessions, processing payments, and managing orders.

Summary of user instructions
- The Chaos-Engineering experiment must be completed within 1 minute.
- When using k6 in steady-state definition, always select a request URL from the following options (other requests are invalid):
  1. http://front-end.sock-shop.svc.cluster.local/
  2. http://front-end.sock-shop.svc.cluster.local/catalogue?size=10
  3. http://front-end.sock-shop.svc.cluster.local/detail.html?id=<ID>
  Replace <ID> with an available ID:
  [03fef6ac-1896-4ce8-bd69-b798f85c6e0b ,
  3395a43e-2d88-40de-b95fe00e1502085b ,
  510a0d7e-8e83-4193-b483-e27e09ddc34d ,
  808a2de1-1aaa-4c25-a9b9-6612e8f29a38 ,
  819e1fbf-8b7e-4f6d-811f-693534916a8b ,
  837ab141-399e-4c1f-9abc-bace40296bac ,

```
 a0a4f044-b040-410d-8ead-4de0446aec7e ,
 d3588630-ad8e-49df-bbd7-3167f7efb246 ,
 zzz4f044-b040-410d-8ead-4de0446aec7e ]
4.
http://front-end.sock-shop.svc.cluster.local/
 category/
5.
http://front-end.sock-shop.svc.cluster.local/
 category?tags=<TAG>
 Replace <TAG> with an available tag: [
 magic , action , blue , brown , black ,
 sport , formal , red , green , skin ,
 geek]
6.
http://front-end.sock-shop.svc.cluster.local/
 basket.html
```

Steady state #1: carts-db-replicas

Description
The first issue to address is the 'Missing Resource Requests' for the 'carts-db' deployment. This is a critical issue because without resource requests, the pods may not get scheduled if the cluster is under resource pressure, leading to potential downtime. To ensure the system's steady state, we need to verify that the 'carts-db' deployment maintains the expected number of replicas, which is 2, even under resource constraints. This can be measured by checking the number of running pods for the 'carts-db' deployment.

Current state inspection
To inspect the 'front-end-replica' state, I will use the K8s API to check the number of ready replicas for the 'front-end' deployment in the 'sock-shop' namespace. This is because the task involves verifying the number of running pods, which is a Kubernetes resource state check. The K8s API is suitable for this purpose as it allows us to directly query the deployment status and ensure that the single replica is always ready. The script will run for a duration of 5 seconds, checking the status every second, to ensure the deployment maintains at least one ready replica 100% of the time.
tool: k8s duration: 5s
Inspection script (Python)
k8s_front-end-replica.py :

```
import os
import time
import argparse
from kubernetes import client, config

def check_carts_db_replicas(
  namespace='sock-shop',
  deployment_name='carts-db', duration=5):
  # Load Kubernetes configuration based on
  the environment
  if os.getenv('KUBERNETES_SERVICE_HOST'):
  config.load_incluster_config()
  else:
  config.load_kube_config()

  v1 = client.AppsV1Api()

  for _ in range(duration):
  try:
    deployment =
    v1.read_namespaced_deployment(
    deployment_name, namespace)
    replicas = deployment.status.replicas
    ready_replicas =
    deployment.status.ready_replicas
    print(f"Total replicas: {replicas},
    Ready replicas: {ready_replicas}")
  except client.exceptions.ApiException as
  e:
    print(f"Exception when calling
    AppsV1Api->read_namespaced_deployment:
    {e}")
  time.sleep(1)

if __name__ == '__main__':
  parser = argparse.ArgumentParser(
  description='Check carts-db replicas')
  parser.add_argument('--duration',
  type=int, default=5, help='Duration to
  check the replicas')
  args = parser.parse_args()
  check_carts_db_replicas(
  duration=args.duration)
```

The inspection script's results (current states) are as follows:

```
Total replicas: 2, Ready replicas: 2
Total replicas: 2, Ready replicas: 2
Total replicas: 2, Ready replicas: 2
Total replicas: 2, Ready replicas: 2
Total replicas: 2, Ready replicas: 2
```

Threshold
The steady state we are considering is the number of replicas for the 'carts-db' deployment. The current state shows that there are 2 total replicas and 2 ready replicas consistently over a period of 5 seconds. This indicates that the system is currently stable with all replicas being ready. Given that the deployment is configured to have 2 replicas, the threshold should ensure that both replicas are ready to maintain the steady state. To account for some fluctuations, we can set a tolerance that allows for a brief period where not all replicas are ready, but the system should recover quickly. A reasonable threshold would be to have at least 1 ready replica 100% of the time and both replicas ready at least 80% of the time during the monitoring period. This ensures that the system remains operational even if one replica temporarily becomes unavailable.

At least 1 ready replica 100% of the time and 2 ready replicas at least 80% of the time during the monitoring period.

Unit test to validate the steady state
To verify the steady state of the 'carts-db' deployment, I will write a unit test that checks the number of ready replicas over a specified duration. The test will ensure that there is at least 1 ready replica 100% of the time and 2 ready replicas at least 80% of the time. I will implement a loop that checks the status every second for the specified duration and use assertions to validate the threshold conditions. The test will be run from the command line with a '--duration' argument to specify the monitoring period.

```
import os
import time
import argparse
from kubernetes import client, config
from unittest_base import K8sAPIBase

class TestCartsDBReplicas(K8sAPIBase):
  def __init__(self, namespace='sock-shop',
  deployment_name='carts-db', duration=5):
    super().__init__()
    self.namespace = namespace
    self.deployment_name = deployment_name
    self.duration = duration

  def test_replicas_threshold(self):
```

```
    # Initialize counters for ready replicas
    total_checks = 0
    ready_replicas_count = 0
    fully_ready_replicas_count = 0

    for _ in range(self.duration):
      try:
        # Read the deployment status
        deployment =
        self.v1.read_namespaced_deployment(
          self.deployment_name,
          self.namespace)
        replicas =
        deployment.status.replicas
        ready_replicas =
        deployment.status.ready_replicas
        print(f"Total replicas: {replicas},
        Ready replicas: {ready_replicas}")

        # Increment the total checks
        total_checks += 1

        # Check if at least 1 replica is
        ready
        if ready_replicas >= 1:
          ready_replicas_count += 1

        # Check if both replicas are ready
        if ready_replicas == 2:
          fully_ready_replicas_count += 1

      except client.exceptions.ApiException
      as e:
        print(f"Exception when calling
        AppsV1Api->read_namespaced_deployment:
        {e}")

      # Wait for 1 second before the next
      check
      time.sleep(1)

    # Calculate the percentage of time
    conditions are met
    one_ready_percentage =
    (ready_replicas_count / total_checks) *
    100
    two_ready_percentage =
    (fully_ready_replicas_count /
    total_checks) * 100

    # Assert the threshold conditions
    assert one_ready_percentage == 100, "At
    least 1 ready replica was not available
    100% of the time."
    assert two_ready_percentage >= 80, "2
    ready replicas were not available at
    least 80% of the time."

    print("Test passed: Steady state
    conditions are satisfied.")


def main():
  parser =
  argparse.ArgumentParser(description='Test
  carts-db replicas threshold')
  parser.add_argument('--duration',
  type=int, default=5, help='Duration to
  check the replicas')
  args = parser.parse_args()

  # Create an instance of the test class
  with the specified duration
  test =
  TestCartsDBReplicas(duration=args.duration)
  # Run the test
  test.test_replicas_threshold()


if __name__ == '__main__':
  main()
```

Pod unittestcarts-db-replicasmod0-pod has failed.

```
Traceback (most recent call last):
  File
  "/chaos-eater/sandbox/cycle_20241127_043136/
        hypothesis/unittest_carts-db-replicas
        _mod0.py", line 68, in <module>
  main()
  File
  "/chaos-eater/sandbox/cycle_20241127_043136/
        hypothesis/unittest_carts-db-replicas
        _mod0.py", line 64, in main
  test.test_replicas_threshold()
  File
  "/chaos-eater/sandbox/cycle_20241127_043136/
        hypothesis/unittest_carts-db-replicas
        _mod0.py", line 23, in
        test_replicas_threshold
  deployment =
  self.v1.read_namespaced_deployment(
    self.deployment_name, self.namespace)
AttributeError: 'CoreV1Api' object has no
attribute 'read_namespaced_deployment'. Did
you mean: 'read_namespaced_event'?
```

The error occurs because the 'CoreV1Api' object does not have a method called 'read_namespaced_deployment'. The correct API to use for reading deployments is 'AppsV1Api'. I will fix the error by changing the API client from 'CoreV1Api' to 'AppsV1Api' in the 'K8sAPIBase' class initialization. This will allow the test to correctly access the deployment information.

```
import os
import time
import argparse
from kubernetes import client, config
from unittest_base import K8sAPIBase

class TestCartsDBReplicas(K8sAPIBase):
  def __init__(self, namespace='sock-shop',
  deployment_name='carts-db', duration=5):
    super().__init__()
    self.namespace = namespace
    self.deployment_name = deployment_name
    self.duration = duration
    # Use the correct API client for
    deployments
    self.apps_v1 = client.AppsV1Api()

  def test_replicas_threshold(self):
    # Initialize counters for ready replicas
    total_checks = 0
    ready_replicas_count = 0
    fully_ready_replicas_count = 0

    for _ in range(self.duration):
      try:
        # Read the deployment status using
        the correct API client
        deployment =
        self.apps_v1.read_namespaced_deployment(
          self.deployment_name,
          self.namespace)
        replicas =
        deployment.status.replicas
        ready_replicas =
        deployment.status.ready_replicas
        print(f"Total replicas: {replicas},
        Ready replicas: {ready_replicas}")

        # Increment the total checks
        total_checks += 1

        # Check if at least 1 replica is
        ready
        if ready_replicas >= 1:
          ready_replicas_count += 1

        # Check if both replicas are ready
        if ready_replicas == 2:
          fully_ready_replicas_count += 1
```

```
      except client.exceptions.ApiException
      as e:
        print(f"Exception when calling
        AppsV1Api->read_namespaced_deployment:
        {e}")

      # Wait for 1 second before the next
      check
      time.sleep(1)

    # Calculate the percentage of time
    conditions are met
    one_ready_percentage =
    (ready_replicas_count / total_checks) *
    100
    two_ready_percentage =
    (fully_ready_replicas_count /
    total_checks) * 100

    # Assert the threshold conditions
    assert one_ready_percentage == 100, "At
    least 1 ready replica was not available
    100% of the time."
    assert two_ready_percentage >= 80, "2
    ready replicas were not available at
    least 80% of the time."

    print("Test passed: Steady state
    conditions are satisfied.")


def main():
  parser =
  argparse.ArgumentParser(description='Test
  carts-db replicas threshold')
  parser.add_argument('--duration',
  type=int, default=5, help='Duration to
  check the replicas')
  args = parser.parse_args()

  # Create an instance of the test class
  with the specified duration
  test =
  TestCartsDBReplicas(duration=args.duration)
  # Run the test
  test.test_replicas_threshold()


if __name__ == '__main__':
  main()
```

Pod unittestcarts-db-replicasmod1-pod has completed sucessfully.
The inspection script's results (current states) are as follows:

```
Total replicas: 2, Ready replicas: 2
Total replicas: 2, Ready replicas: 2
Total replicas: 2, Ready replicas: 2
Total replicas: 2, Ready replicas: 2
Total replicas: 2, Ready replicas: 2
Test passed: Steady state conditions are
satisfied.
```

Steady state #2: front-end-replica

Description
The next issue to address is the 'Single Replica Deployment' for the 'front-end' deployment. This is a critical issue because having only one replica can lead to downtime if the pod fails. To ensure the system's steady state, we need to verify that the 'front-end' deployment maintains at least one running pod at all times. This can be measured by checking the number of running pods for the 'front-end' deployment. Given that the deployment is configured to have only 1 replica, the threshold should ensure that this single replica is always ready to maintain the steady state. A reasonable threshold would be to have at least 1 ready replica 100% of the time during the monitoring period. This ensures that the system remains operational even if there are
temporary issues with the pod.

Current state inspection
To inspect the 'front-end-replica' state, I will use the K8s API to check the number of ready replicas for the 'front-end' deployment in the 'sock-shop' namespace. This is because the task involves verifying the number of running pods, which is a Kubernetes resource state check. The K8s API is suitable for this purpose as it allows us to directly query the deployment status and ensure that the single replica is always ready. The script will run for a duration of 5 seconds, checking the status every second, to ensure the deployment maintains at least one ready replica 100% of the time.
tool: k8s duration: 5s
Inspection script (Python)
k8s_front-end-replica.py :

```
import os
import time
import argparse
from kubernetes import client, config

def check_front_end_replicas(namespace,
deployment_name, duration):
  # Load Kubernetes configuration based on
  the environment
  if os.getenv('KUBERNETES_SERVICE_HOST'):
    config.load_incluster_config()
  else:
    config.load_kube_config()

  v1 = client.AppsV1Api()
  ready_replicas_count = 0

  for _ in range(duration):
    resp =
    v1.read_namespaced_deployment_status(
      deployment_name, namespace)
    ready_replicas =
    resp.status.ready_replicas or 0
    print(f"Ready replicas for
    {deployment_name}: {ready_replicas}")
    if ready_replicas >= 1:
      ready_replicas_count += 1
    time.sleep(1)

  print(f"{deployment_name} was ready
  {ready_replicas_count}/{duration} times.")

if __name__ == '__main__':
  parser =
  argparse.ArgumentParser(description='Check
  front-end replicas')
  parser.add_argument('--duration',
  type=int, default=5, help='Duration to
  check the replicas')
  args = parser.parse_args()
  check_front_end_replicas('sock-shop',
  'front-end', args.duration)
```

Pod k8sfront-end-replica-pod has been completed successfully.
The inspection script's results (current states) are as follows:

```
Ready replicas for front-end: 1
Ready replicas for front-end: 1
Ready replicas for front-end: 1
Ready replicas for front-end: 1
Ready replicas for front-end: 1
front-end was ready 5/5 times.
```

Threshold
The steady state we are considering is the availability of the 'front-end' deployment, which currently has only one replica. The current state shows that the 'front-end' deployment has 1 ready replica consistently over a monitoring period of 5 seconds. To ensure the system remains operational, the threshold should be set such that the 'front-end' deployment maintains at least 1 ready replica 100% of the time during the monitoring period. This accounts for the fact that there is only one replica, and any downtime would mean the service is unavailable. Given the current state satisfies this condition, the threshold should be set to ensure that at least 1 ready replica is present 100% of the time during the monitoring period.

At least 1 ready replica must be present 100% of the time during the monitoring period.

Unit test to validate the steady state
To verify the steady state of the 'front-end' deployment, I will write a unit test that checks if there is at least 1 ready replica 100% of the time during the specified monitoring period. The test will use a for-loop to check the status every second for the given duration. The threshold is set to ensure that the 'front-end' deployment maintains at least 1 ready replica 100% of the time, which means the number of times the ready replicas are 1 or more should equal the duration. I will use the Kubernetes Python API to interact with the cluster and check the deployment status. The test will include an assertion to verify that the steady state satisfies the threshold, and it will print a summary of the results at the end.
```
import os
import time
import argparse
from kubernetes import client, config
from unittest_base import K8sAPIBase

class TestFrontEndReplica(K8sAPIBase):
  def __init__(self, namespace,
  deployment_name, duration):
    super().__init__()
    self.namespace = namespace
    self.deployment_name = deployment_name
    self.duration = duration

  def test_steady_state(self):
    ready_replicas_count = 0

    # Loop for the specified duration
    for _ in range(self.duration):
      # Get the deployment status
      resp =
      self.v1.read_namespaced_deployment_status(
        self.deployment_name,
        self.namespace)
      ready_replicas =
      resp.status.ready_replicas or 0
      print(f"Ready replicas for
      {self.deployment_name}:
      {ready_replicas}")

      # Check if the number of ready
      replicas is at least 1
      if ready_replicas >= 1:
        ready_replicas_count += 1

      # Wait for 1 second before the next
      check
      time.sleep(1)

    # Calculate the percentage of time the
    deployment was ready
    readiness_percentage =
    (ready_replicas_count / self.duration)
    * 100
    print(f"{self.deployment_name} was
    ready
    {ready_replicas_count}/{self.duration}
    times.")

    # Assert that the deployment was ready
    100% of the time
    assert readiness_percentage == 100,
    f"{self.deployment_name} readiness was
    {readiness_percentage}%, expected 100%."


def main():
  parser =
  argparse.ArgumentParser(description='Test
  front-end replica readiness')
  parser.add_argument('--duration',
  type=int, default=5, help='Duration to
  check the replicas')
  args = parser.parse_args()

  # Create a test instance and run the test
  test = TestFrontEndReplica('sock-shop',
  'front-end', args.duration)
  test.test_steady_state()


if __name__ == '__main__':
  main()
```

Pod unittestfront-end-replicamod0-pod has failed.
```
Traceback (most recent call last):
  File
  "/chaos-eater/sandbox/cycle_20241127_043136/
        hypothesis/unittest_front-end-replica
        _mod0.py", line 50, in <module>
  main()
  File
  "/chaos-eater/sandbox/cycle_20241127_043136/
       hypothesis/unittest_front-end-replica
       _mod0.py\", line 46, in main
  test.test_steady_state()
  File
  \"/chaos-eater/sandbox/cycle_20241127_043136/
         hypothesis/unittest_front-end-replica
         _mod0.py\", line 20, in
         test_steady_state
  resp =
  self.v1.read_namespaced_deployment_status(
    self.deployment_name, self.namespace)
AttributeError: 'CoreV1Api' object has no
attribute
'read_namespaced_deployment_status'. Did
you mean: 'read_namespaced_pod_status'?
```

The error occurs because the CoreV1Api class does not have a method called read_namespaced_deployment_status . Instead, the correct class to use for reading deployment status is AppsV1Api . I will fix the error by changing the API client from CoreV1Api to AppsV1Api in the K8sAPIBase class. This will allow the test to correctly
call the read_namespaced_deployment_status method and check the deployment status. The rest of the test logic remains the same, ensuring that the deployment maintains at least 1 ready replica 100% of the time during the specified duration.
```
import os
import time
import argparse
from kubernetes import client, config
from unittest_base import K8sAPIBase

class TestFrontEndReplica(K8sAPIBase):
```

```
  def __init__(self, namespace,
  deployment_name, duration):
    super().__init__()
    self.namespace = namespace
    self.deployment_name = deployment_name
    self.duration = duration
    # Use AppsV1Api to interact with
    deployments
    self.apps_v1 = client.AppsV1Api()

  def test_steady_state(self):
    ready_replicas_count = 0

    # Loop for the specified duration
    for _ in range(self.duration):
      # Get the deployment status
      resp = self.apps_v1
             .read_namespaced_deployment_status(
        self.deployment_name,
        self.namespace)
      ready_replicas =
      resp.status.ready_replicas or 0
      print(f"Ready replicas for
      {self.deployment_name}:
      {ready_replicas}")

      # Check if the number of ready
      replicas is at least 1
      if ready_replicas >= 1:
        ready_replicas_count += 1

      # Wait for 1 second before the next
      check
      time.sleep(1)

    # Calculate the percentage of time the
    deployment was ready
    readiness_percentage =
    (ready_replicas_count / self.duration)
    * 100
    print(f"{self.deployment_name} was
    ready
    {ready_replicas_count}/{self.duration}
    times.")

    # Assert that the deployment was ready
    100% of the time
    assert readiness_percentage == 100,
    f"{self.deployment_name} readiness was
    {readiness_percentage}%, expected 100%."


def main():
  parser =
  argparse.ArgumentParser(description='Test
  front-end replica readiness')
  parser.add_argument('--duration',
  type=int, default=5, help='Duration to
  check the replicas')
  args = parser.parse_args()

  # Create a test instance and run the test
  test = TestFrontEndReplica('sock-shop',
  'front-end', args.duration)
  test.test_steady_state()


if __name__ == '__main__':
  main()
```

Pod unittestfront-end-replicamod1-pod has completed sucessfully.
The inspection script's results (current states) are as follows:

```
Ready replicas for front-end: 1
Ready replicas for front-end: 1
Ready replicas for front-end: 1
Ready replicas for front-end: 1
Ready replicas for front-end: 1
front-end was ready 5/5 times.
```

Failure scenario: Black Friday Sale

Description

During a Black Friday sale, the system is expected to handle a significant increase in traffic, which can stress the system's resources and reveal weaknesses. The 'carts-db' deployment lacks resource requests, which can lead to scheduling issues under resource pressure. The 'front-end' deployment has only one replica, making it a single point of failure. To simulate the Black Friday event, we will first inject a 'StressChaos' fault to simulate high CPU usage on the 'carts-db' pods, testing if the system can maintain the steady state of having at least 1 ready replica 100% of the time and 2 ready replicas 80% of the time. Next, we will inject a 'PodChaos' fault to kill the 'front-end' pod, testing if the system can maintain the steady state of having at least 1 ready replica 100% of the time. This sequence simulates the increased load and potential failures
during a high-traffic event like Black Friday.

Failure-injection sequence

StressChaos ({'namespace': 'sock-shop', 'label': 'name=carts-db'}) -> PodChaos ({'namespace': 'sock-shop', 'label': 'name=front-end'})

Detailed failure parameters

Detailed parameters of StressChaos ({'namespace': 'sock-shop', 'label': 'name=carts-db'})

```
{
  "mode": "all",
  "selector": {
  "namespaces": [
    "sock-shop"
  ],
  "labelSelectors": {
    "name": "carts-db"
  },
  "stressors": {
    "cpu": {
    "workers": 2,
    "load": 80
    }
  },
  "containerNames": [
    "carts-db"
  ]
  }
}
```

Detailed parameters of PodChaos ({'namespace': 'sock-shop', 'label': 'name=front-end'})

```
{
  "action": "pod-kill",
  "mode": "one",
  "selector":{
  "namespaces": ["sock-shop"],
  "labelSelectors": {
    "name": "front-end"
  }
  },
  "value": "1"
}
```

Chaos experiment (Planning)

Time Schedule
The chaos engineering experiment is designed to test the system's resilience under stress conditions and potential failures. Given the constraints, the experiment must be completed within 1 minute. The experiment is divided into three phases: pre-validation, fault-injection, and post-validation.

1. Pre-validation Phase: This phase ensures that the system is in a steady state before any faults are injected. We will allocate 20 seconds for this phase. During this time, we will verify that the 'carts-db' deployment maintains at least 1 ready replica 100% of the time and 2 ready replicas 80% of the time, and that the 'front-end' deployment maintains at least 1 ready replica 100% of the time.

2. Fault-injection Phase: This phase involves injecting faults to simulate the Black Friday event. We will allocate 20 seconds for this phase. The faults include a 'StressChaos' fault to simulate high CPU usage on the 'carts-db' pods and a 'PodChaos' fault to kill the 'front-end' pod. These faults will test the system's ability
to maintain the defined steady states under stress and failure conditions.

3. Post-validation Phase: This phase ensures that the system returns to its steady state after the faults are removed. We will allocate 20 seconds for this phase. During this time, we will again verify the steady states for the 'carts-db' and 'front-end' deployments to ensure they meet the defined thresholds.

The total time for the experiment is 60 seconds, with each phase receiving an equal allocation of 20 seconds. This allocation allows for a balanced approach to validating the system's steady state, injecting faults, and confirming recovery.
Total experiment time: 60s
Pre-validation Phase: 20s
Fault-injection Phase: 20s
Post-validation Phase: 20s

Pre-validation Phase
In the pre-validation phase, we need to ensure that the system is in a steady state before we introduce any faults. This involves verifying that the current state of the system meets the defined steady state thresholds. Given the constraints of a 20-second total time for this phase, we will execute the unit tests for both steady states simultaneously to maximize efficiency. The first steady state, 'carts-db-replicas requires checking that at least 1 replica is ready 100% of the time and 2 replicas are ready at least 80% of the time. The second steady state, 'front-end-replica', requires ensuring that the single replica is ready 100% of the time. Both tests will run for the full 20 seconds, with checks occurring every second. This approach ensures that we have sufficient data to confirm the system's readiness before proceeding to the fault injection phase.

- Verified Steady State #0:
carts-db-replicas
  - Workflow Name:
  pre-unittest-carts-db-replicas
  - Grace Period: 0s
  - Duration: 20s

- Verified Steady State #1:
front-end-replica
  - Workflow Name:
  pre-unittest-front-end-replica
  - Grace Period: 0s
  - Duration: 20s

Failure-injection Phase
In this fault-injection phase, we aim to simulate a Black Friday sale scenario by introducing two types of faults: 'StressChaos' and 'PodChaos'. The goal is to observe how the system behaves under stress and potential failure conditions. The phase is limited to 20 seconds, so we need to carefully time the injections and tests to fit within this window.

First, we will inject a 'StressChaos' fault on the 'carts-db' deployment to simulate high CPU usage. This will help us test the system's ability to maintain the steady state of having at least 1 ready replica 100% of the time and 2 ready replicas 80% of the time. We will start this fault injection immediately at the beginning of the phase and let it run for 10 seconds.

Simultaneously, we will run the unit test for the 'carts-db-replicas' steady state to verify if the system can maintain the required number of ready replicas during the stress period. This test will also start at the beginning of the phase and run for 10 seconds.

Next, we will inject a 'PodChaos' fault to kill the 'front-end' pod. This will test the system's ability to maintain the steady state of having at least 1 ready replica 100% of the time for the 'front-end' deployment. We will start this fault injection at the 10-second mark and let it run for 5 seconds.

Finally, we will run the unit test for the 'front-end-replica' steady state to verify if the system can maintain the required readiness during the pod failure. This test will start at the 10-second mark and run for 5 seconds.

By staggering the fault injections and unit tests in this manner, we can effectively simulate the Black Friday scenario and assess the system's resilience under stress and failure conditions.

- Verified Steady State #0:
carts-db-replicas
  - Workflow Name:
  fault-unittest-carts-db-replicas
  - Grace Period: 0s
  - Duration: 10s

- Verified Steady State #1:
front-end-replica
  - Workflow Name:
  fault-unittest-front-end-replica
  - Grace Period: 10s
  - Duration: 5s

- Injected Faults #0: StressChaos
  - Workflow Name: fault-stresschaos
  - Grace Period: 0s
  - Duration: 10s

- Injected Faults #1: PodChaos
  - Workflow Name: fault-podchaos
  - Grace Period: 10s
  - Duration: 5s

Post-validation Phase
In the post-validation phase, we need to

ensure that the system has returned to its steady states after the fault injection. The steady states we are concerned with are the 'carts-db-replicas' and 'front-end-replica'. The post-validation phase will involve running unit tests to verify these steady states. Given the total time of 20 seconds for this phase, we will stagger the unit tests to ensure both steady states are verified within this timeframe.

The 'carts-db-replicas' steady state requires that at least 1 ready replica is available 100% of the time and 2 ready replicas are available at least 80% of the time. We will allocate 10 seconds for this test, with a grace period of 0 seconds, as we want to immediately verify the recovery of the 'carts-db' deployment after the stress test.

The 'front-end-replica' steady state requires that at least 1 ready replica is available 100% of the time. We will allocate the remaining 10 seconds for this test, also with a grace period of 0 seconds, to ensure the 'front-end' deployment has recovered from the pod kill fault.

By staggering the tests in this manner, we ensure that both steady states are verified within the 20-second post-validation phase, allowing us to confirm that the system has returned to its expected operational state.

- Verified Steady State #0: carts-db-replica
  - Workflow Name:
  post-unittest-carts-db-replicas
  - Grace Period: 0s
  - Duration: 10s

- Verified Steady State #1:
front-end-replica
  - Workflow Name:
  post-unittest-front-end-replica
  - Grace Period: 0s
  - Duration: 10s

Summary
The chaos engineering experiment is structured into three phases, each lasting 20 seconds, to test the system's resilience. In the pre-validation phase, both unit tests for verifying steady states are executed simultaneously for the entire 20 seconds. The 'carts-db-replicas' test checks that at least one replica is ready 100% of the time and two replicas are ready 80% of the time, while the 'front-end-replica' test ensures one replica is ready 100% of the time. In the fault-injection phase, the 'StressChaos' fault is injected on the 'carts-db' deployment to simulate high CPU usage, starting immediately and lasting for 10 seconds. Concurrently, the 'carts-db-replicas' unit test runs for 10 seconds to verify readiness under stress. At the 10-second mark, the 'PodChaos' fault is injected to kill the 'front-end' pod, lasting for 5 seconds, while the 'front-end-replica' unit test runs for 5 seconds to check readiness during the fault. In the post-validation phase, the 'carts-db-replicas' unit test runs for the first 10 seconds to verify recovery, followed by the 'front-end-replica' unit test for the next 10 seconds, ensuring both deployments return to their steady states.

The generated Chaos Mesh workflow manifest:
```

```
apiVersion: chaos-mesh.org/v1alpha1
kind: Workflow
metadata:
  name: chaos-experiment-20241127-045539
spec:
  entry: the-entry
  templates:
  #------------------------------
  # entry point of whole workflow
  #------------------------------
  - name: the-entry
    templateType: Serial
    deadline: 30m45s
    children:
    - pre-validation-phase
    - fault-injection-phase
    - post-validation-phase

  #-------------------------------------
  # Entry point of pre-validation-phase
  #-------------------------------------
  - name: pre-validation-phase
    templateType: Serial
    deadline: 10m20s
    children:
    - pre-validation-parallel-workflows

  - name: pre-validation-parallel-workflows
    templateType: Parallel
    deadline: 5m20s
    children:
    - pre-unittest-carts-db-replicas
    - pre-unittest-front-end-replica

  # Definitions of children of
  pre-validation-phase
  - name: pre-unittest-carts-db-replicas
    templateType: Task
    deadline: 5m20s
    task:
    container:
      name:
      pre-unittest-carts-db-replicas-container
      image: chaos-eater/k8sapi:1.0
      imagePullPolicy: IfNotPresent
      command: ["/bin/bash", "-c"]
      args: ["python
        /chaos-eater/sandbox/cycle_20241127
        _043136/unittest_carts-db-replicas
        _mod0.py --duration 20"]
      volumeMounts:
      - name: pvc-volume
        mountPath: /chaos-eater
    volumes:
      - name: pvc-volume
      persistentVolumeClaim:
        claimName: pvc

  - name: pre-unittest-front-end-replica
    templateType: Task
    deadline: 5m20s
    task:
    container:
      name:
      pre-unittest-front-end-replica-container
      image: chaos-eater/k8sapi:1.0
      imagePullPolicy: IfNotPresent
      command: ["/bin/bash", "-c"]
      args: ["python
        /chaos-eater/sandbox/cycle_20241127
        _043136/unittest_front-end-replica
        _mod0.py --duration 20"]
      volumeMounts:
      - name: pvc-volume
        mountPath: /chaos-eater
    volumes:
      - name: pvc-volume
      persistentVolumeClaim:
        claimName: pvc

  #-------------------------------------
  # Entry point of fault-injection-phase
  #-------------------------------------
  - name: fault-injection-phase
    templateType: Serial

```
    deadline: 10m15s
    children:
    - fault-injection-overlapped-workflows

  - name: fault-injection-parallel-workflow
    templateType: Parallel
    deadline: 5m10s
    children:
    - fault-unittest-carts-db-replicas
    - fault-stresschaos

  - name: fault-injection-suspend-workflow
    templateType: Serial
    deadline: 5m15s
    children:
    - fault-injection-suspend
    - fault-injection-parallel-workflows

  - name: fault-injection-suspend
    templateType: Suspend
    deadline: 10s

  - name: fault-injection-parallel-workflows
    templateType: Parallel
    deadline: 5m5s
    children:
    - fault-unittest-front-end-replica
    - fault-podchaos

  - name:
  fault-injection-overlapped-workflows
    templateType: Parallel
    deadline: 5m15s
    children:
    - fault-injection-parallel-workflow
    - fault-injection-suspend-workflow

  # Definitions of children of
  pre-validation-phase
  # unit tests
  - name: fault-unittest-carts-db-replicas
    templateType: Task
    deadline: 5m10s
    task:
    container:
      name:
      fault-unittest-carts-db-replicas-container
      image: chaos-eater/k8sapi:1.0
      imagePullPolicy: IfNotPresent
      command: ["/bin/bash", "-c"]
      args: ["python
        /chaos-eater/sandbox/cycle_20241127
        _043136/unittest_carts-db-replicas
        _mod0.py --duration 10"]
      volumeMounts:
      - name: pvc-volume
        mountPath: /chaos-eater
    volumes:
      - name: pvc-volume
      persistentVolumeClaim:
        claimName: pvc

  - name: fault-unittest-front-end-replica
    templateType: Task
    deadline: 5m5s
    task:
    container:
      name:
      fault-unittest-front-end-replica-container
      image: chaos-eater/k8sapi:1.0
      imagePullPolicy: IfNotPresent
      command: ["/bin/bash", "-c"]
      args: ["python
        /chaos-eater/sandbox/cycle_20241127
        _043136/unittest_front-end-replica
        _mod0.py --duration 5"]
      volumeMounts:
      - name: pvc-volume
        mountPath: /chaos-eater
    volumes:
      - name: pvc-volume
      persistentVolumeClaim:
        claimName: pvc

  # fault_injections

  - name: fault-stresschaos
    templateType: StressChaos
    deadline: 10s
    stressChaos:
    containerNames:
      - carts-db
    mode: all
    selector:
      labelSelectors:
      name: carts-db
      namespaces:
      - sock-shop
    stressors:
      cpu:
      load: 80
      workers: 2


  - name: fault-podchaos
    templateType: PodChaos
    deadline: 5s
    podChaos:
    action: pod-kill
    mode: one
    selector:
      labelSelectors:
      name: front-end
      namespaces:
      - sock-shop
    value: '1'


  #-------------------------------------
  # Entry point of post-validation phase
  #-------------------------------------
  - name: post-validation-phase
    templateType: Serial
    deadline: 10m10s
    children:
    - post-validation-parallel-workflows

  - name: post-validation-parallel-workflows
    templateType: Parallel
    deadline: 5m10s
    children:
    - post-unittest-carts-db-replicas
    - post-unittest-front-end-replica

  # Definitions of children of
  pre-validation-phase
  - name: post-unittest-carts-db-replicas
    templateType: Task
    deadline: 5m10s
    task:
    container:
      name:
      post-unittest-carts-db-replicas-container
      image: chaos-eater/k8sapi:1.0
      imagePullPolicy: IfNotPresent
      command: ["/bin/bash", "-c"]
      args: ["python
        /chaos-eater/sandbox/cycle_20241127
        _043136/unittest_carts-db-replicas
        _mod0.py --duration 10"]
      volumeMounts:
      - name: pvc-volume
        mountPath: /chaos-eater
    volumes:
      - name: pvc-volume
      persistentVolumeClaim:
        claimName: pvc

  - name: post-unittest-front-end-replica
    templateType: Task
    deadline: 5m10s
    task:
    container:
      name:
      post-unittest-front-end-replica-container
      image: chaos-eater/k8sapi:1.0
      imagePullPolicy: IfNotPresent
      command: ["/bin/bash", "-c"]
      args: ["python
        /chaos-eater/sandbox/cycle_20241127
        _043136/unittest_front-end-replica
```

```
        _mod0.py --duration 10"]
      volumeMounts:
      - name: pvc-volume
        mountPath: /chaos-eater
    volumes:
      - name: pvc-volume
      persistentVolumeClaim:
        claimName: pvc
```

Analysis

The chaos engineering experiment aimed to test the resilience of the 'sock-shop' system under simulated fault conditions, specifically focusing on the 'carts-db' and 'front-end' deployments. The experiment was structured into three phases: pre-validation, fault-injection, and post-validation. The results indicate that while the 'carts-db' deployment maintained its steady state throughout the experiment, the 'front-end' deployment failed to meet the expected steady state during both the fault-injection and post-validation phases.

1. Pre-Validation Phase:
  - Both the 'carts-db-replicas' and 'front-end-replica' tests passed, indicating that the system was in a stable state before any faults were injected. This confirms that the initial configuration of the system was functioning as expected.

2. Fault-Injected Phase:
  - The 'carts-db-replicas' test passed during the fault-injection phase, demonstrating that the 'carts-db' deployment was resilient to the 'StressChaos' fault, which simulated high CPU usage. This suggests that despite the lack of resource requests, the 'carts-db' deployment was able to maintain at least one ready replica 100% of the time and two ready replicas 80% of the time.
  - The 'front-end-replica' test failed during the fault-injection phase. The 'PodChaos' fault, which involved killing the 'front-end' pod, resulted in 0 ready replicas throughout the test duration. This failure highlights the critical issue of having only a single replica for the 'front-end' deployment, making it a single point of failure. The system was unable to recover or maintain availability when the single pod was terminated.

3. Post-Validation Phase:
  - The 'carts-db-replicas' test continued to pass in the post-validation phase, indicating that the 'carts-db' deployment returned to its steady state after the fault was removed.
  - The 'front-end-replica' test failed again in the post-validation phase, with 0 ready replicas recorded throughout the test. This suggests that the 'front-end' deployment did not recover after the fault was removed, likely due to the absence of additional replicas to take over when the single pod was killed.

Insights and Recommendations:
- The failure of the 'front-end' deployment to maintain its steady state during and after the fault-injection phase underscores the importance of addressing the 'Single Replica Deployment' issue. To improve resilience, it is recommended to increase the number of replicas for the 'front-end' deployment. This will ensure that the system can maintain availability even if one pod fails.

- Consider implementing horizontal pod autoscaling for the 'front-end' deployment to dynamically adjust the number of replicas based on traffic and resource usage. This can help the system handle sudden spikes in demand, such as those experienced during events like Black Friday.

- Review and potentially adjust the readiness and liveness probes for the 'front-end' deployment to ensure that Kubernetes can accurately determine the health of the pods and take corrective actions if necessary.

By addressing these issues, the system's resilience can be significantly improved, reducing the
risk of downtime during high-traffic events and fault scenarios.

Improvement

To improve the resiliency of the 'front-end' deployment and avoid failures during fault scenarios, we need to address the 'Single Replica Deployment' issue. The simplest reconfiguration is to increase the number of replicas for the 'front-end' deployment from 1 to 2. This change will ensure that there is at least one additional replica available to handle traffic if one pod fails, thus maintaining the steady state of having at least 1 ready replica 100% of the time.

Modification_type: replace
File name:
sock-shop-2/manifests/09-front-end-dep.yaml

Increase the number of replicas from 1 to 2 to improve resiliency and ensure availability during pod failures.
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: front-end
  namespace: sock-shop
spec:
  replicas: 2
  selector:
  matchLabels:
    name: front-end
  template:
  metadata:
    labels:
    name: front-end
  spec:
    containers:
    - name: front-end
    image: weaveworksdemos/front-end:0.3.12
    resources:
      limits:
      cpu: 300m
      memory: 1000Mi
      requests:
      cpu: 100m
      memory: 300Mi
    ports:
    - containerPort: 8079
    env:
    - name: SESSION_REDIS
      value: "true"
    securityContext:
      runAsNonRoot: true
      runAsUser: 10001
      capabilities:
```

```
      drop:
        - all
      readOnlyRootFilesystem: true
    livenessProbe:
      httpGet:
      path: /
      port: 8079
      initialDelaySeconds: 300
      periodSeconds: 3
    readinessProbe:
      httpGet:
      path: /
      port: 8079
      initialDelaySeconds: 30
      periodSeconds: 3
    nodeSelector:
    beta.kubernetes.io/os: linux
```

Experiment replanning

Failure scope of StressChaos
The current Kubernetes manifests show that the 'carts-db' deployment remains unchanged in terms of its namespace and labels. The 'carts-db' deployment is still in the 'sock-shop' namespace and has the label 'name: carts-db'. Therefore, the scope of the fault injection, which targets the 'carts-db' pods using the label selector 'name: carts-db' in the 'sock-shop' namespace, does not need to be adjusted. The original goal of the chaos experiment, which is to test the system's resilience under stress conditions by applying a 'StressChaos' fault to the 'carts-db' pods, remains valid with the current manifests.

Failure scope of PodChaos
The current Kubernetes manifests show that the 'front-end' deployment has increased its replicas from 1 to 2. This change does not affect the label selectors or namespace, which remain the same. Therefore, the scope of the fault injection, which targets the 'front-end' pod using the label 'name: front-end' in the 'sock-shop' namespace, remains valid. The fault injection will still target one of the 'front-end' pods, as specified by the 'mode: one' parameter.

VaC script for steady state #1
The unit test is specifically targeting the carts-db deployment, which is defined in the 03-carts-db-dep.yaml manifest. Upon comparing the previous and reconfigured manifests for carts-db, there are no changes in the configuration. The number of replicas remains the same, and there are no modifications to the deployment's specifications that would affect the unit test. Therefore, the unit test does not require any adjustments as the reconfigured manifests do not introduce any changes that would impact the test's logic or its threshold conditions.

VaC script for steady state #2
The unit test is targeting the 'front-end' deployment in the 'sock-shop' namespace. In the reconfigured manifests, the 'front-end' deployment has been modified to increase the number of replicas from 1 to 2. However, the unit test is designed to verify that there is at least 1 ready replica at all times, which is still valid even with the increased number of replicas. The test checks for a minimum of 1 ready replica, and the reconfiguration was likely done to ensure that this condition is met more reliably. Therefore, the unit test does not require any modification, as the threshold of having at least 1 ready replica remains unchanged and is still applicable.

Summary of the completed CE cycle

The Chaos Engineering cycle for the 'sock-shop' system involved a detailed analysis and testing of the system's resilience under simulated fault conditions. The system is composed of multiple Kubernetes manifests that define various microservices, such as 'carts', 'catalogue', 'orders', 'payment', and 'user', each with associated deployments and services.
The primary focus of the experiment was to address identified resiliency issues, including missing resource requests, single replica deployments, missing liveness and readiness probes, and hardcoded environment variables.

The experiment was structured into three phases: pre-validation, fault-injection, and post-validation, each lasting 20 seconds, to fit within a total duration of 60 seconds. The pre-validation phase ensured that the system was in a steady state before any faults were injected. The fault-injection phase simulated a Black Friday sale scenario by introducing 'StressChaos' and 'PodChaos' faults to test the system's behavior under stress and potential failure conditions. The post-validation phase verified that the system returned to its steady state after the faults were removed.

The initial experiment revealed that while the 'carts-db' deployment maintained its steady state throughout, the 'front-end' deployment failed during the fault-injection and post-validation phases due to having only a single replica, which made it a single point of failure.
This led to a recommendation to increase the number of replicas for the 'front-end' deployment to improve resilience.

After modifying the 'front-end' deployment to have two replicas, a second experiment was conducted. This time, all unit tests passed, indicating that the system successfully maintained its steady states during and after the fault-injection phase. The improvements ensured that the system could handle the simulated high-traffic event and recover from faults, demonstrating enhanced resilience and availability.

E.4 Reviews

Review by Engineer A

Overall: 4
- Effectively identified issues in the SockShop system.
- The system's resilience and availability can be significantly improved, reducing the risk of downtime during high-traffic events and fault scenarios.

Hypothesis: 4
The hypothesis phase correctly identified two critical steady-state conditions for

SockShop:
- Steady state #1: carts-db-replicas
 + The first issue to address is the 'Missing Resource Requests' for the 'carts-db' deployment. This is a critical issue because without resource requests, the pods may not get scheduled if the cluster is under resource pressure, leading to potential downtime.
 + At least 1 ready replica 100\% of the time and 2 ready replicas 80\% of the time during the monitoring period
- Steady state #2: front-end-replica
 + The next issue to address is the 'Single Replica Deployment' for the 'front-end' deployment. This is a critical issue because having only one replica can lead to downtime if the pod fails.
 + At least 1 ready replica must be present 100\% of the time during the monitoring period

Additionally, it is set up considering an actual failure scenario: During a Black Friday sale, the system is expected to handle a significant increase in traffic, which can stress the system's resources and reveal weaknesses. However, they did not account for other potential failure scenarios such as database issues or network disruptions, limiting the overall scope of resiliency testing.

Experiment: 4
- The experiment was structured into three phases: pre-validation, fault-injection, and post-validation, each lasting 20 seconds, to fit within a total duration of 60 seconds.
- The pre-validation phase ensured that the system was in a steady state before any faults were injected.
- The fault-injection phase simulated a Black Friday sale scenario by introducing 'StressChaos' and 'PodChaos' faults to test the system's behavior under stress and potential failure conditions.
- The post-validation phase verified that the system returned to its steady state after the faults were removed.
- The experiment was accurate and effectively validated the hypothesis.
- Additionally, it is set up considering an actual failure scenario: Black Friday sale, but they did not account for other potential failure scenarios such as database issues or network disruptions, limiting the overall scope of resiliency testing.

Analysis: 5
The analysis phase correctly identified the single replica issue as the primary cause of downtime, while the 'carts-db' deployment maintained its steady state throughout, the 'front-end' deployment failed during the fault-injection and post-validation phases due to having only a single replica

Meaningful insights for the improvement
- Recommended to increase the number of replicas for the 'front-end' deployment. This will ensure that the system can maintain availability even if one pod fails.
- Consider implementing horizontal pod autoscaling for the 'front-end' deployment to dynamically adjust the number of replicas based on traffic and resource usage.
- Review and potentially adjust the readiness and liveness probes for the 'front-end' deployment to ensure that Kubernetes can accurately determine the health of the pods and take corrective actions if necessary.

Improvement: 5
- After modifying the 'front-end' deployment to have two replicas, a second experiment was conducted. This time, all unit tests passed, indicating that the system successfully maintained its steady state during and after the fault-injection phase.
- The improvements ensured that the system could handle the simulated high-traffic event and recover from faults, demonstrating enhanced resilience and availability.

Review by Engineer B (Translated)

Overall: 3
Reason for positive evaluation:
- Unlike the NGINX case study, SOCKSHOP is composed of multiple microservices, which increases the complexity of each chaos engineering process. However, the process has been automated, allowing for a streamlined workflow.

Room for improvement:
- Since SOCKSHOP is composed of multiple microservices, it would be beneficial to explore more complex scenarios where individual microservices interact with each other. This could help uncover issues that are difficult to detect during normal operations-for example, the impact of a specific microservice failure or the system-wide effects caused by delays in a particular service.

Hypothesis: 4
Reason for positive evaluation:
- The steady-state conditions and failure scenarios (such as a significant increase in system traffic during a Black Friday sale) are specifically and concretely considered.

Room for improvement:
- While the 'carts-db' and 'front-end' microservices were highlighted as important components, it would be helpful to understand the relative importance of the other microservices as well.

Experiment: 4
Reason for positive evaluation:
- In the fault injection phase, 'StressChaos' and 'PodChaos' were used to simulate the Black Friday sale scenario.

Room for improvement:
- It would be even better if vulnerabilities, database issues, network failures, and compound failures were also taken into consideration.

Analysis: 4
Reason for positive evaluation
- The analysis identified that the 'front-end' service has only a single replica, highlighting it as a single point of failure.
- All the recommendations provided under Insights and Recommendations are effective and well thought out.

Improvement: 5
Reason for positive evaluation:
- A solution was proposed to increase the number of replicas for deployments with a single replica, in order to prevent downtime.

Room for improvement:
- It would be helpful if manifest files were also prepared for the other recommendations listed in the Insights and Recommendations section of the ANALYSIS, beyond just increasing the number of replicas.
- Regarding the resiliency issues/weaknesses in the pre-processed manifests, such as the lack of resource requests and the absence of liveness and readiness probes, it would be beneficial if improved manifests were proposed where all Deployment resources are configured with resources, livenessProbe, and readinessProbe.
- For the issue of hardcoded environment variables (e.g., passwords), which was pointed out as a potential security vulnerability under "Resiliency Issues/Weaknesses in the Pre-processed Manifests," it would be valuable if a solution like using Kubernetes Secrets was proposed.

Review by GPT-4o

Overall: 4
The Chaos Engineering cycle effectively identifies and addresses critical issues, leading to improved system resilience.

Strengths:
- Successfully identifies and addresses critical issues in the system.
- Provides meaningful insights for future cycles.
- Well-structured experiment with clear objectives and phases.
- Effective use of Chaos Mesh for automated fault injection.
- Demonstrates a clear understanding of the system's architecture and needs.

Weaknesses:
- Limited focus on only two components, potentially missing other critical areas.
- Could include more diverse fault scenarios for broader testing.
- The cycle could explore more potential improvements beyond the immediate issues.
- Limited exploration of network-related issues or external factors.
- Recommendations could be more detailed and specific.

Reason for the score:
The cycle fixes critical issues in the system and offers meaningful insights for the next cycle, but could be more comprehensive in its scope and recommendations.

Hypothesis: 4
The hypothesis aims to ensure that the system maintains its steady states even when faults are injected, focusing on the 'carts-db' and 'front-end' deployments.

Strengths:
- Clearly defines the steady states for 'carts-db' and 'front-end'.
- Provides specific thresholds for evaluating the steady states.
- Utilizes Python scripts with K8s API for precise monitoring.
- Addresses critical issues like missing resource requests and single replica deployment.
- Considers realistic fault scenarios like Black Friday sales.

Weaknesses:
- The hypothesis could benefit from more detailed consideration of other potential failure points.
- It assumes the system's current configuration is optimal without exploring other potential improvements.
- The hypothesis does not account for network-related issues that could affect steady states.
- The focus is limited to only two components, potentially overlooking other critical parts of the system.
- The hypothesis could include more diverse fault scenarios to test broader system resilience.

Reason for the score:
The hypothesis is relevant and meaningful, addressing critical issues and leading to system improvement. It offers insights for the next cycle, but could be expanded to cover more components and scenarios.

Experiment: 4
The experiment is well-structured into three phases: pre-validation, fault-injection, and post-validation, each lasting 20 seconds.

Strengths:
- Clearly defined phases with specific objectives.
- Efficient use of time within the 1-minute constraint.
- Simultaneous execution of unit tests for efficiency.
- Use of Chaos Mesh for automated fault injection.
- Comprehensive coverage of both steady states during all phases.

Weaknesses:
- The experiment could include more diverse fault types to test different aspects of system resilience.
- Limited to only two components, potentially missing other critical areas.
- The 20-second allocation per phase may not be sufficient for more complex systems.
- The experiment does not account for potential network delays or other external factors.
- The use of only two fault types may not fully simulate real-world scenarios.

Reason for the score:
The experiment plan correctly serves to validate the hypothesis and considers actual failure scenarios, but could be improved by including more diverse faults and components.

Analysis: 4
The analysis identifies the resilience of 'carts-db' and the failure of 'front-end' during fault injection, providing insights for improvement.

Strengths:
- Provides a clear breakdown of results for each phase.
- Identifies the critical issue of single replica deployment for 'front-end'.
- Offers actionable recommendations for improvement.
- Highlights the resilience of 'carts-db' under stress.
- Uses detailed logs to support findings.

Weaknesses:
- The analysis could explore more potential causes for the 'front-end' failure.
- Limited focus on only two components,

potentially missing other issues.
- Could include more quantitative data to support findings.
- The analysis does not consider external factors that may have influenced results.
- Recommendations could be more detailed and specific.

Reason for the score:
The analysis reports correct and meaningful information, providing insights for improvement, but could be more comprehensive in exploring other potential issues.

Improvement: 5
The improvement involved increasing the number of replicas for the 'front-end' deployment, successfully addressing the single point of failure issue.

Strengths:
- Directly addresses the critical issue identified in the analysis.
- Successfully changes the system to satisfy the hypothesis.
- Simple yet effective solution to improve resilience.
- Demonstrates a clear understanding of the system's needs.
- Results in successful unit tests in the second experiment.

Weaknesses:
- The improvement is limited to only one component, potentially overlooking other areas.
- Could consider additional improvements like autoscaling for further resilience.
- Does not address other potential issues identified in the hypothesis.
- The improvement could be more comprehensive, considering other components.
- Limited exploration of alternative solutions.

Reason for the score:
The improvement successfully changes the system to satisfy the hypothesis in the first attempt, but could be more comprehensive in addressing other potential issues.

Review by Claude Sonnet 3.5

Overall: 4
This Chaos Engineering cycle successfully identified and addressed a critical resilience issue in the sock-shop application. It focused on the 'carts-db' and 'front-end' deployments, simulating a Black Friday sale scenario with increased CPU load and pod failures.

Strengths:
- Well-structured experiment with clear phases and relevant fault injections
- Effective use of Chaos Mesh for automated execution
- Successful identification of a critical single point of failure
- Quick and effective improvement that resolved the issue

Weaknesses:
- Limited scope - only focused on two components of the system
- Short duration of experiment phases
- Did not address all identified issues, such as missing resource requests

Reason for the score:
The cycle successfully identified and fixed a critical issue in the system, improving its resilience to pod failures. The improvement directly resulted from the insights gained through the chaos experiment. While the cycle was successful in its primary goal, it did not address all identified issues and had a limited scope, preventing it from receiving the highest score.

Hypothesis: 4
The hypothesis aims to verify that the system maintains its steady states even when subjected to fault scenarios simulating a Black Friday sale event. It focuses on two key components: the 'carts-db' deployment and the 'front-end' deployment.

Strengths:
- Clear definition of steady states for both 'carts-db' and 'front-end' deployments
- Specific thresholds set for each steady state
- Consideration of real-world scenario (Black Friday sale)
-Use of custom Python scripts with K8s API for verification

Weaknesses:
- Limited scope - only focuses on two components of the system
- Does not consider potential cascading effects on other services
- Thresholds may not be stringent enough for a high-traffic event like Black Friday",

Reason for the score:
The hypothesis is well-defined and relevant to the system, addressing critical issues identified in the initial analysis. It leads to system improvement by highlighting the need for increased replicas in the front-end deployment. The hypothesis also offers insights for future improvements, such as considering resource requests for other components. However, it could be more comprehensive by including other critical services in the sock-shop application.

Experiment: 4
The experiment is designed to test the system's resilience under simulated Black Friday conditions, focusing on the 'carts-db' and 'front-end' deployments. It's structured into three phases: pre-validation, fault-injection, and post-validation, each lasting 20 seconds.

Strengths:
- Well-structured experiment with clear phases
- Use of relevant chaos faults (StressChaos and PodChaos)
- Parallel execution of tests to maximize efficiency
- Detailed Chaos-Mesh-Workflow file for automated execution

Weaknesses:
- Short duration of each phase (20 seconds) may not be sufficient for thorough testing
- Limited fault scenarios - only tests CPU stress and pod failure
- Does not include load testing to simulate actual user traffic

Reason for the score:
The experiment plan is well-designed and correctly serves to validate the hypothesis. It uses actual failure scenarios (CPU stress and pod failure) that are relevant to a high-traffic event. The use of Chaos Mesh for automated execution

is a strong point. However, the short duration and limited scope of fault scenarios prevent it from being scored a 5.

Analysis: 5
The analysis report provides a comprehensive overview of the experiment results, highlighting the success of the 'carts-db' deployment in maintaining its steady state and the initial failure of the 'front-end' deployment to meet its steady state during fault injection and post-validation phases.

Strengths:
- Detailed breakdown of results for each phase of the experiment
- Clear identification of the root cause of failure (single replica for front-end)
- Insightful recommendations for improvement
- Consideration of both immediate fixes and long-term enhancements

Weaknesses:
- Limited analysis of potential impacts on other system components
- No quantitative analysis of the performance degradation
- Lack of discussion on potential trade-offs of the proposed improvements

Reason for the score:
The analysis provides correct and meaningful information about the experiment results. It offers valuable insights for improvement, particularly the recommendation to increase the number of replicas for the front-end deployment. The analysis directly leads to a system improvement that addresses the identified weakness.

Improvement: 5
Based on the analysis, the improvement focused on addressing the single point of failure in the front-end deployment. The number of replicas for the front-end deployment was increased from 1 to 2 in the Kubernetes manifest.

Strengths:
- Direct address of the identified issue
- Simple and effective solution
- Immediate improvement in system resilience
= Successful validation in the second experiment run

Weaknesses:
- Only addresses one of the identified issues
- Does not implement more advanced solutions like horizontal pod autoscaling
- No changes made to improve resource management for other components

Reason for the score:
The improvement successfully changes the system to satisfy the hypothesis in the first attempt. The modification to increase the number of front-end replicas directly addresses the identified weakness and results in the system passing all unit tests in the second experiment run.

Review by Gemini 1.5 Pro

Overall: 4
This Chaos Engineering cycle effectively identifies and addresses a critical vulnerability in the 'front-end' deployment of the 'sock-shop' application. The cycle demonstrates a clear progression through the hypothesis, experiment, analysis, and improvement phases, resulting in a more resilient system. The use of Chaos Mesh and automated unit tests strengthens the experiment and analysis, providing valuable insights into system behavior under stress and failure conditions.

Strengths:
- Clearly defined hypothesis and steady states, focusing on relevant system weaknesses.
- Well-structured experiment plan with appropriate use of Chaos Mesh for fault injection.
- Accurate analysis of experiment results, correctly identifying the root cause of failure.
- Effective improvement implemented by increasing 'front-end' replicas, validated by subsequent experiment.
- The cycle demonstrates a systematic approach to improving system resilience.",

Weaknesses:
- Limited scope, focusing only on two deployments and a single failure scenario. A broader approach would provide a more comprehensive understanding of system resilience.
- Short experiment duration, potentially missing long-term effects of faults.
- Lack of detailed monitoring and logging information in the experiment plan.
- Improvement limited to increasing replicas, not exploring other potential enhancements like resource requests or autoscaling.
- Limited consideration of the interdependencies between services and potential cascading failures.

Reason for the score:
The cycle fixes a critical issue in the system ('front-end' single point of failure) and demonstrates a good understanding of Chaos Engineering principles. However, it does not offer comprehensive insights for the next cycle due to its limited scope.

Hypothesis: 4
The hypothesis aims to verify the system's resilience during a simulated Black Friday sale by focusing on the 'carts-db' and 'front-end' deployments. It posits that the defined steady states, namely maintaining a specific number of ready replicas for these deployments, will be upheld even under stress and failure conditions induced by Chaos Mesh.

Strengths:
- Clearly defines the steady states for both 'carts-db' and 'front-end' deployments, specifying the desired number of ready replicas and the timeframes for measurement.
- Directly addresses identified weaknesses in the system, namely missing resource requests for 'carts-db' and single replica deployment for 'front-end'.
- Provides specific thresholds for each steady state, allowing for quantifiable measurement and validation.
- Uses Python scripts with Kubernetes API calls to automate the verification of steady states, ensuring consistent and reliable measurements.
- The chosen fault scenario (Black Friday sale) is relevant to the application's context and helps simulate realistic high-traffic conditions.

Weaknesses:

- The hypothesis lacks consideration for other potential failure scenarios beyond the simulated Black Friday sale. Exploring a wider range of faults would provide a more comprehensive assessment of system resilience.
- The hypothesis focuses solely on replica availability and does not consider other important metrics such as latency, error rates, or resource consumption. Incorporating these metrics would provide a more holistic view of system health.
- The hypothesis does not explicitly state the expected behavior of the system when the steady states are violated. Defining the expected failure modes would help in analyzing the experiment results and identifying areas for improvement.
- The hypothesis could benefit from a more precise definition of the \"monitoring period\" used in the steady state thresholds. Specifying the exact duration and frequency of measurements would enhance clarity.
- The hypothesis does not consider the potential impact of the faults on other services within the 'sock-shop' application. A more comprehensive approach would involve analyzing the cascading effects of failures.

Reason for the score:
The hypothesis is relevant to the system and meaningful, directly addressing the identified weaknesses. It also leads to system improvement by prompting changes to the 'front-end' deployment. However, it does not offer comprehensive insights for the next cycle due to its limited scope and lack of consideration for other metrics and failure scenarios.

Experiment: 4
The experiment plan is well-structured, dividing the process into pre-validation, fault-injection, and post-validation phases, each lasting 20 seconds. It uses Chaos Mesh to simulate a Black Friday sale scenario by injecting 'StressChaos' on the 'carts-db' deployment and 'PodChaos' on the 'front-end' deployment. The plan includes running unit tests to verify the defined steady states before, during, and after fault injection.

Strengths:
- Clearly defined phases with specific time allocations, ensuring a structured and controlled experiment.
- Uses Chaos Mesh, a suitable tool for Kubernetes environments, to inject realistic faults.
- Includes both stress testing and failure injection, covering a range of potential issues.
- Employs unit tests to automatically verify steady states, providing objective and repeatable results.
- The experiment design allows for parallel execution of tests and fault injections, maximizing efficiency within the time constraint.

Weaknesses:
- The experiment focuses solely on two specific deployments ('carts-db' and 'front-end') and does not consider the potential impact on other interconnected services. A more comprehensive approach would involve analyzing the cascading effects of failures.
- The experiment duration is relatively short (1 minute), which may not be sufficient to observe the long-term effects of the faults. Extending the experiment duration would provide more insights into system behavior.
- The experiment plan lacks details on monitoring and logging. Specifying which metrics will be collected and how the logs will be analyzed would enhance the analysis phase.
- The experiment plan does not include a rollback strategy in case the injected faults cause unexpected or severe disruptions to the system.
- The experiment relies on specific unit tests, which may not cover all aspects of system behavior. Incorporating other monitoring tools and metrics would provide a more comprehensive view of system health.

Reason for the score:
The experiment plan correctly serves to validate the hypothesis and is set up considering an actual failure scenario (Black Friday sale). It utilizes appropriate tools and techniques for fault injection and steady-state verification.

Analysis: 4
The analysis accurately interprets the experiment results, correctly identifying the failure of the 'front-end' deployment to maintain its steady state during and after fault injection. It provides insights into the cause of the failure, attributing it to the single replica deployment. The analysis also offers recommendations for improvement, such as increasing the number of replicas and implementing horizontal pod autoscaling.

Strengths:
- Clearly presents the results of each phase of the experiment, highlighting the success of 'carts-db' and the failure of 'front-end'.
- Correctly identifies the root cause of the 'front-end' failure as the single replica deployment.
- Provides specific and actionable recommendations for improvement, such as increasing replicas and using horizontal pod autoscaling.
- Connects the analysis back to the initial hypothesis and the identified weaknesses in the system.
- Offers insights into the resilience of 'carts-db' despite the lack of resource requests.

Weaknesses:
- The analysis lacks depth in exploring the behavior of other services during the experiment. A more comprehensive analysis would consider the impact on the entire application.
- The analysis does not include any visualizations or graphs to represent the data, which would enhance understanding and communication of the results.
- The analysis could benefit from a more detailed explanation of the observed behavior, including specific metrics and log data.
- The analysis does not discuss the potential limitations of the experiment, such as the short duration or the specific choice of faults.
- The analysis does not provide specific metrics or data to support the claims made. Including quantitative data would strengthen the analysis.

Reason for the score:
The analysis reports correct and meaningful information and provides some insights for improvement. It correctly identifies the root cause of the 'front-end' failure and

suggests appropriate solutions.

Improvement: 4
The improvement phase successfully addresses the identified issue by increasing the number of replicas for the 'front-end' deployment from 1 to 2. This change directly targets the single point of failure observed in the experiment and aims to improve the system's resilience.

Strengths:
- Directly addresses the root cause identified in the analysis phase.
- Implements a simple and effective solution by increasing the number of replicas.
- The modification is clearly documented, including the updated YAML manifest.
- The improvement is validated in a subsequent experiment, demonstrating its effectiveness.
- The change improves the overall resilience of the 'front-end' service.

Weaknesses:
- The improvement focuses solely on increasing replicas and does not consider other potential improvements, such as resource requests or autoscaling.
- The improvement lacks a discussion of potential trade-offs, such as increased resource consumption.
- The improvement does not address the lack of liveness and readiness probes in other deployments.
- The improvement process could benefit from a more iterative approach, starting with smaller changes and gradually increasing complexity.
- The improvement does not consider the potential impact on other services within the application.

Reason for the score:
The improvement successfully changes the system to satisfy the hypothesis in the second attempt, demonstrating a clear and effective response to the identified weakness.

arXiv:2405.19954v1 [cs.CR] 30 May 2024

GenKubeSec: LLM-Based Kubernetes Misconfiguration Detection, Localization, Reasoning, and Remediation

Ehud Malul, Yair Meidan, Dudu Mimran, Yuval Elovici, Asaf Shabtai
Department of Software and Information Systems Engineering
Ben-Gurion University of The Negev

ABSTRACT

A key challenge associated with Kubernetes configuration files (KCFs) is that they are often highly complex and error-prone, leading to security vulnerabilities and operational setbacks. Rule-based (RB) tools for KCF misconfiguration detection rely on static rule sets, making them inherently limited and unable to detect newly-discovered misconfigurations. RB tools also suffer from misdetection, since mistakes are likely when coding the detection rules. Recent methods for detecting and remediating KCF misconfigurations are limited in terms of their scalability and detection coverage, or due to the fact that they have high expertise requirements and do not offer automated remediation along with misconfiguration detection. Novel approaches that employ LLMs in their pipeline rely on API-based, general-purpose, and mainly commercial models. Thus, they pose security challenges, have inconsistent classification performance, and can be costly. In this paper, we propose GenKubeSec, a comprehensive and adaptive, LLM-based method, which, in addition to detecting a wide variety of KCF misconfigurations, also identifies the exact location of the misconfigurations and provides detailed reasoning about them, along with suggested remediation. When empirically compared with three industry-standard RB tools, GenKubeSec achieved equivalent precision (0.990 ± 0.020) and superior recall (0.999 ± 0.026). When a random sample of KCFs was examined by a Kubernetes security expert, GenKubeSec's explanations as to misconfiguration localization, reasoning and remediation were 100% correct, informative and useful. To facilitate further advancements in this domain, we share the unique dataset we collected, a unified misconfiguration index we developed for label standardization, our experimentation code, and GenKubeSec itself as an open-source tool. A video demonstrating our implementation of GenKubeSec can be found here: https://youtu.be/hBehYfdR-zM.

CCS CONCEPTS

• **Computing methodologies** → **Natural language generation**;
• **Security and privacy** → **Vulnerability scanners**.

KEYWORDS

Large Language Models (LLMs), Infrastructure-as-Code (IAC), DevOps, Kubernetes (K8s), Artificial Intelligence (AI), Static Analysis

1 INTRODUCTION

The cloud native computing landscape has undergone a significant transformation, with rapid adoption of container-based environments in a relatively short period of time. Kubernetes (K8s) [56] has emerged as the preferred choice for orchestrating containerized applications, driven by the benefits K8s offers, such as efficiency, portability, isolation, scalability, and flexibility [20]. The strong community [54] support surrounding containerization technologies has strengthened their leading role in the development ecosystem.

The plethora of K8s configuration files (KCFs) available in public repositories [18] streamlines the creation and deployment of new environments and applications, permitting developers to easily create cloud environments by duplicating open-source KCFs. However, this convenience comes with its own set of challenges. Open-source KCFs, while widely available, may contain undetected bugs, errors, vulnerabilities, and other types of misconfigurations (abbreviated as misconfigs), potentially leaving deployed systems exposed to cyber threats [17, 35, 37, 52, 83, 92]. Moreover, due to the adoption of K8s in a broad variety of production environments, and the accessibility of open-source KCFs, attackers invest significant efforts in identifying KCF misconfigs that can serve as exploitation entry points into clusters or applications [50, 68].

Given the associated risks, powerful tools capable of detecting KCF misconfigs are required. Rule-based (RB) static analysis tools like SLI-Kube [82], Checkov [14], KubeLinter [53], and Terrascan [90] are recognized tools for assessing the security of KCFs based on predefined rules. However, due to their static nature, RB tools also struggle with adapting to new misconfigs, primarily due to the need for coding new error-prone detection rules. Recent studies on KCF misconfig detection proposed methods that rely on more innovative approaches, such as knowledge graphs and topological graphs [11, 39]. However, these methods face limitations in scalability, expertise requirements, reliance on high-quality input data, and automated remediation, as elaborated in Secs. 7.2 and 7.3.

Recent advances in large language models (LLMs) have proven highly effective in various application domains [34, 41], including cybersecurity [87, 102], where several studies suggested using LLMs in static analysis tasks [2, 59, 61, 65, 103]. Since KCFs are technically (semi-structured) text files, two studies focused specifically on LLM-based methods for KCF misconfig detection and remediation [60, 69]. These studies, however, have the following limitations: (1) each of them utilized LLMs to solve only a sub-task rather than addressing all of the related challenges from end-to-end, including detection, localization, reasoning, and remediation, (2) they utilized pretrained LLMs which had only undergone basic domain adaptation and not LLMs fine-tuned to the specific task, thus demonstrating limited performance, (3) the inspected files were sent to external LLMs via an API, introducing security (privacy) risks and adding unnecessary overhead, (4) relatively small datasets with small number of misconfig types were used for evaluation, and (5) the experimental results were reported very scarcely, so their performance during future inference cannot be guaranteed.

In light of the need to mitigate KCF-related risks, the limitations of existing RB and graph-based methods, the potential of LLMs, and

the shortcomings and scarcity of research on applying LLMs in KCF misconfig detection and remediation, in this paper we introduce *GenKubeSec*: a novel and comprehensive LLM-based method which comprises of three key components. In the first component, a substantial set of KCFs is collected, and then labeled and standardized using a unified misconfig index we developed. In the second component, named *GenKubeDetect*, since general purpose pretrained LLMs performed poorly, a base LLM is optimized and fine-tuned using a large and comprehensive set of labeled KCFs to detect a wide variety of KCF misconfigs (the scope of which continually grows over the course of time). In the third component, named *GenKubeResolve*, a pretrained LLM is adapted, using prompt engineering and few-shot learning techniques [12, 64, 106], to provide the exact location of each misconfig detected in a KCF by GenKubeDetect, clear explanations in human language, and suggested remediation. Unlike previous studies where LLMs were utilized for either KCF misconfig detection [60] or remediation [69], we propose an end-to-end LLM-based method which addresses both of these aspects. In addition, GenKubeResolve also provides localization and reasoning for each misconfig detected by GenKubeDetect.

In our extensive evaluation, GenKubeDetect achieved detection precision of 0.990 (equivalent to three industry standard RB tools) and recall of 0.999 (superior to any of those tools individually). In addition, manual, expert-based analysis of false misconfig detections made by GenKubeDetect revealed that more than 83% of these detections were actually correct. This demonstrates the ability of our LLM-based method to generalize and detect misconfigs which are not covered by the rule sets of the three RB tools. These evaluation results were achieved with a relatively lightweight LLM architecture and thus save substantial time and compute. Another benefit is that GenKubeDetect supports KCF misconfig learning both in primary batch mode, as part of fine-tuning, and in ongoing mode, using just a few examples of newly-discovered misconfigs. With GenKubeResolve, based on a random sample of KCFs, a K8s expert validated that *all* of the localization, reasoning, and remediation suggestions were clear and accurate. A notable advantage of GenKubeSec, is that by relying on local LLMs rather than API-based LLMs, GenKubeSec minimizes security and privacy risks, as it does not expose proprietary KCFs (with potential vulnerabilities) to external entities. To conclude, the contributions of our research are as follows:

- To the best of our knowledge, we are the first to propose an end-to-end LLM-based method that provides KCF misconfig detection (both known and new), along with localization, reasoning, and remediation.
- As part of our quantitative evaluation, we developed a unified misconfig index (UMI) of 169 KCF misconfigs to enable standardization among detection tools; this index can be used by the scientific community and industry to further advance research in this area through improved consistency and comparability.
- We collected a large and diverse set of real-world material from the K8s domain. This dataset includes approximately 700MB of K8s-focused free text, as well as almost 277,000 KCFs which we labeled using the abovementioned UMI. Such datasets are known to be rare [60], and therefore, we make this unique dataset publicly available to enable our research to be reproduced, and to advance research in this area by the scientific community.

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-name
spec:
  containers:
    - name: some-container
      image: some-image
      command: [ some-command ]
  securityContext:
    privileged: true
```

Figure 1: An example of a misconfigured KCF: the container is configured to run with privileged access; this can potentially expose sensitive host information.

- To further facilitate transparency, usability, and benchmarking, we also make our code publicly available via a GitHub repository and make GenKubeSec available as an open-source tool.

2 BACKGROUND

2.1 K8s and related security challenges

2.1.1 K8s and KCFs. K8s is the primary container orchestration tool for 71% of Fortune 100 companies [21, 33]. KCF (also known as a *manifest file* [55]), refers to any file that defines the configuration of various K8s components [51], and describes how an application or service should be deployed and run within a K8s cluster [57]. KCFs provide a human-readable representation (in a YAML or JSON format) of an application's requirements, enabling reproducible deployment and consistent application behavior. KCFs are essential building blocks in containerized and cloud-native architectures, enabling developers and operators to define, manage, and scale applications effectively.

2.1.2 KCF misconfigs and vulnerabilities. The complexity associated with configuring KCFs mainly stems from the inter-dependencies among the large variety of configuration parameters [86]. This complexity can lead to misconfigs in KCFs. A KCF misconfig (see example in Fig. 1) is the result of the incorrect configuration of system components. Such misconfig can lead to security vulnerabilities and affect system workload, integrity, and performance. KCF misconfigs can occur at the edge, application, or cluster level, allowing attackers to exploit containers and hosts. They can also occur with parameters affecting resource allocation, networking, security policies, and more. These misconfigs can have far-reaching consequences, resulting in application downtime, performance bottlenecks, and resource wastage. KCF misconfigs may also lead to non-compliance with industry regulations and data protection laws [58]. According to a CNCF report from 2022 [19], KCF misconfigs are the biggest security threat when using K8s. This report also states that 94% of companies that use K8s encountered a security vulnerability. Another study [9] revealed that 100% of 10,000 checked clusters had misconfigs, and 65% had at least one high-severity misconfig.

2.2 Large language models

LLMs represent a significant advancement in natural language processing (NLP), enhancing machines' ability to understand and generate human-like language. Utilizing deep learning (DL) techniques and extensive textual datasets, LLMs excel in tasks like sentiment

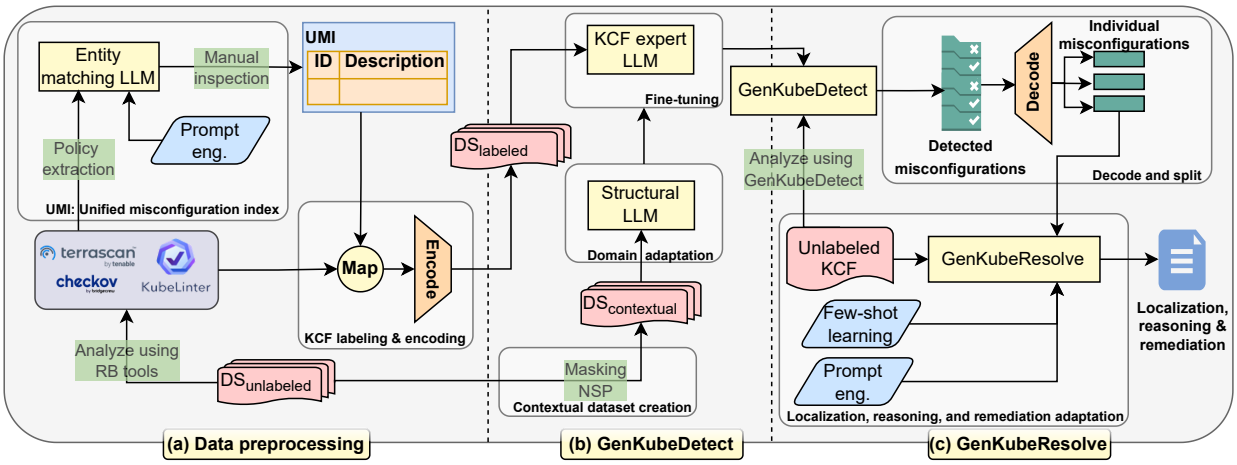


Figure 2: Overview of GenKubeSec. This method includes a data preprocessing component (a), in which a unified misconfig index (UMI) is created, and KCFs are labeled and encoded; the GenKubeDetect component (b) in which an LLM for KCF structure understanding and misconfig detection is trained; the GenKubeResolve component (c) in which a pretrained LLM with prompt engineering and few-shot learning is used for localization, reasoning, and remediation of identified misconfigs.

analysis [62] and text generation and translation [105]. Contemporary pretrained LLMs, which continuously evolve and increase in size and sophistication, can be adapted for specialized domains [36] using, e.g., fine-tuning and continual learning, as described next.

Fine-tuning a pretrained LLM involves additional training focused on specific objectives, like aligning model outputs with user intent, generating ethical and accurate responses, and improving performance on particular tasks. This process not only enhances the model's ability to understand and respond to task-specific instructions but also makes it more robust to domain shifts, without significantly increasing computational costs. Fine-tuning methods vary and include transfer learning with task-specific data, instruction-tuning using formatted data, and alignment-tuning to align model outputs with human values. These approaches ensure that LLMs operate effectively in varied and specific contexts [71].

Continual learning (CL) [101] is a pivotal aspect of modern machine learning (ML). CL enables systems to gradually learn and integrate diverse contents or skills via an additional step of training, using newly-acquired data. This approach is essential for developing robust and adaptable ML systems capable of responding to changing demands and incorporating new knowledge efficiently [95]. In the context of domain adaptation, CL enhances an LLM's ability to balance the retention of original domain knowledge with learning from the targeted domain distribution [4, 71, 89], ensuring effective adaptation without 'catastrophic forgetting' [30, 67]. Moreover, CL maintains computational efficiency during domain-adaption [84]. Employing techniques like next sentence prediction (NSP) which help the model maintain context and coherence between sequences, and masking, which focuses the model's attention on relevant parts of the input [7, 23], along with unsupervised learning [93, 99], enhance LLMs' comprehension and response capabilities.

3 PROPOSED METHOD

In this paper, we propose GenKubeSec: a novel LLM-based method for KCF misconfig detection, localization, reasoning, and remediation, aimed at mitigating KCF-related K8s security vulnerabilities. Our method consists of three components. Initial data preprocessing is performed in the first component, and LLMs are used in the other two components as follows: The GenKubeDetect component is trained (fine-tuned) to detect any misconfigs that may exist in a given KCF, serving as an advanced, LLM-based alternative to existing rule- and graph-based tools and methods (described in Sec. 7.2). The GenKubeResolve component, which is based on a pretrained LLM and specifically crafted prompts, is trained to provide localization, reasoning, and remediation suggestions for each misconfig detected by GenKubeDetect.

3.1 Data preprocessing component

3.1.1 Creating a unified misconfig index. Given the large number of KCF misconfig types that can be detected by various tools, recent studies [38, 69] stressed the need for misconfig label standardization. To address this need, we createded a unified misconfig index (UMI). Our UMI consists of multiple, consistently-formatted, unique misconfig class labels. That is, if a certain misconfig was annotated differently in various sources, whether described differently in text or using different misconfig IDs, the UMI maps all of the variations of the misconfig to a single ID (i.e., label). In addition to being useful in our research, the UMI can facilitate the development of effective solutions and their comparison.

The first step in creating the UMI involved collecting as many policies and detection rules as possible from multiple reliable resources, which in our case were the three RB KCF misconfig detection tools: Checkov, KubeLinter, and Terrascan. In the second step, an LLM was employed for entity matching [76, 77] using prompt engineering (see Fig. 2.a); the entity matching prompt structure is

presented in Appendix A. Manual inspection of the entity matching results was then performed, to verify their correctness; this was followed by manual adjustments when needed. Entries in the UMI (each of which represents a KCF misconfig) are formatted as the combination of a numerical <misconfig_ID> $\in (0, 1, 2, \dots)$ and a textual <misconfig_description>, for example, (9, 'Default namespace should not be used'). The UMI we developed can be found in our GitHub repository.

3.1.2 Collecting and labeling KCFs. To train the LLMs used in GenKubeDetect and GenKubeResolve, we collect a large and diverse set of KCFs. We annotate this raw dataset of unlabeled KCFs as $DS_{unlabeled}$ (see Fig. 2.a). Then, to create $DS_{labeled}$, we associate ground-truth misconfig labels to the KCFs in $DS_{unlabeled}$ using Checkov, KubeLinter, and Terrascan (which the UMI is based on). We opted to use three detection tools due to the limitations of RB detection methods. Nonetheless, $DS_{labeled}$ may still consist of some undetected misconfigs that are not covered by the unified rule sets of the three RB tools. In practice, when any of these tools detects a misconfig for a KCF in $DS_{unlabeled}$, the detected misconfig is mapped to its corresponding UMI misconfig_ID and replaced by a pair of values formatted as (<impacted_K8s_resource>+<misconfig_ID>), e.g., 'app+52' (see full example in Appendix B). We consider each pair of such values as an encoded label. Encoding (compressing) the class labels decreases the overall size of labeled KCFs, such that (1) higher coverage is provided by LLMs that impose size limits, (2) training is accelerated, and (3) fewer misclassifications are likely to occur, as shorter class labels are easier for the LLM to (re-)generate.

3.2 The GenKubeDetect component

The GenKubeDetect component of GenKubeSec, is a fine-tuned LLM which specializes in detecting KCF misconfigs. Specifically, given a KCF as input, GenKubeDetect's main goal is to use an LLM in order to generate a list of detected misconfigs, formatted as the encoded labels described above. We achieve this goal by selecting a base LLM (which is preferably pretrained on English semi-structured data to resemble the hierarchical textual format used by KCFs) and fine-tuning it for the task of misconfig detection. Fine-tuning the LLM is performed in two stages: (1) adapting it so that it 'understands' the general structure of KCFs, and (2) further adapting it so that it can detect misconfigs in KCFs.

3.2.1 Adapting the base LLM for KCF structure understanding. To familiarize GenKubeDetect's LLM with the specific structure of KCFs, it is fine-tuned using the $DS_{contextual}$ dataset (Fig. 2.b). $DS_{contextual}$ is constructed by applying two techniques on $DS_{unlabeled}$, namely masking and NSP (see Sec. 2.2). Masking is randomly applied to 15% of each KCF (we chose to mask 15%, as this is typically done in similar cases [23, 66, 81]). Using NSP, GenKubeDetect is trained to predict the second half of a KCF based on its first half. This first stage of the fine-tuning process enables the LLM to recognize and interpret the details and dependencies inherent in KCFs, thereby enhancing its ability to perform subsequent domain-specific tasks (such as misconfig detection) with greater accuracy and efficiency. The output of this stage is the structural LLM (shown in Fig. 2.b).

3.2.2 Fine-tuning the structural LLM for KCF misconfig detection. Once specialized on the structure of KCFs, the structural LLM is further fine-tuned using $DS_{labeled}$, enabling it to function as a multi-label classifier; thus, given a KCF as input during inference, the LLM will generate text that serves as the predicted class labels of (zero or more) detected misconfigs. To enhance this process, we used Low-Rank Adaptation (LoRA) [40, 42]. LoRA is a technique that allows fine-tuning a subset of parameters within a large pre-trained model by decomposing a large matrix into two smaller low-rank matrices in the attention layers, while keeping the original weights unchanged. This makes the process more efficient, as it requires fewer resources and allows for faster convergence without the need to retrain the entire model from scratch. LoRA preserves the integrity of the original model's weights, maintaining its knowledge and capabilities while adapting to specific tasks or datasets. The output of this stage is the KCF expert LLM (shown in Fig. 2.b).

3.3 The GenKubeResolve component

Given a KCF as input, GenKubeDetect is trained to generate (possibly multiple) detected misconfig labels, each in the previously mentioned encoded format, e.g., 'app+52.' Then, to expand GenKubeSec's capabilities further, for each misconfig detected by GenKubeDetect, GenKubeResolve is trained to generate (1) the exact location of the misconfig, (2) reasoning regarding the nature or cause of the detected misconfig, and (3) recommended remediation steps. To accomplish this, in the first stage of this component we decode and split the class labels, and in the second stage we adapt a pretrained LLM using prompt engineering techniques, as described below.

3.3.1 Decoding and splitting. In the first stage of GenKubeResolve, the encoded labels are decoded (see Fig. 2.c) and then split into pairs of (inspected_KCF, decoded_label). In practice, we replace the misconfig_ID in each encoded label with the associated misconfig_description from the UMI (see Appendix B); this provides the LLM with much more contextual information, enabling it to respond more accurately. The subsequent splitting (illustrated in Fig. 2.c) performed at this point is aimed at preventing the LLM from being 'confused' due to an overabundance of misconfigs in the same KCF. Moreover, label splitting enables this component to provide localization, reasoning, and remediation suggestions for each misconfig separately.

3.3.2 Adapting an LLM for localization, reasoning, and remediation. The task of detecting a large variety of K8s misconfigs performed by GenKubeDetect is an extremely specific task for a general-purpose pretrained LLM, (as reflected by the experimental results in Sec. 5.1). This is why we chose to implement GenKubeDetect using fine-tuning techniques; these techniques, however, impose significant overhead. In contrast, preliminary experimentation with GenKubeResolve showed that for the task of providing localization, reasoning, and remediation for detected KCF misconfigs, fine-tuning is not essential. Instead, we decided to adapt a pretrained LLM for this task, using few-shot learning techniques (which provide the LLM with favorable relevant examples as part of in-context learning [12, 64, 106]) and prompt engineering techniques. An illustration of this process can be found in Fig. 2.c, while prompt examples are provided in Appendix C.

The base model we chose for GenKubeResolve is a pretrained Mistral LLM. It was chosen due to its superior recall and F1-score

(abbreviated as F1) in our preliminary experiments (Sec. 5.1), the fact that it is one of the best open-source LLMs available [46], and the ability to use it locally, free of charge. Local use of Mistral ensures that, unlike web-based pretrained LLMs, no KCF data is exposed to external APIs. Using a local LLM also accelerates operation by eliminating the delays and complexities of API calls.

To specialize Mistral for localization, reasoning, and remediation, we optimized a system prompt which, based on a provided KCF and a misconfig detected in it, instructs Mistral to return the exact line number of the misconfig, the logic behind the detection, and suggestions on how to fix the misconfig. To enhance Mistral's performance, we augmented this system prompt with a few examples of KCFs, detected misconfigs, and expected output (i.e., few-shot learning). The system prompt we optimized and an inference example for GenKubeResolve are presented in Appendix C.

4 EVALUATION METHOD

4.1 Experimental setup

Throughout our quantitative evaluation we used the Python programming language. The Hugging Face transformers library [100] served as the primary source for training and utilizing state-of-the-art LLMs (see the complete code in the project's GitHub repository). For heavy-lifting computing, mainly during the LLM fine-tuning step, we used NVIDIA RTX 4090 [74] and RTX 6000 [75] Ada GPUs.

To create our UMI (Sec. 3.1.1), we began by performing targeted web crawling of the online documentation (i.e policy indexes) of three open-source RB tools, namely KubeLinter, Checkov, and Terrascan, which are industry standards [24, 28]. We then utilized OpenAI's GPT-4 for entity matching, and establishing in a UMI of 170 standardized unique misconfigs, one of which was used for KCFs where no misconfig was identified.

To produce $DS_{unlabeled}$, we collected 276,520 unlabeled KCFs from the K8s Manifest Subset [88], recently released by 'The Stack' [48]. In order to associate ground-truth labels to each KCF in $DS_{unlabeled}$, we utilized the three open-source RB tools mentioned earlier (KubeLinter, Checkov, and Terrascan), resulting in $DS_{labeled}$.

4.2 Data partitioning and performance metrics

To compare the performance of various KCF misconfig detectors, whether rule- or LLM-based, we randomly split $DS_{labeled}$ into fixed training (80%), validation (10%), and test (10%) sets. We compared their performance using the common classification metrics of precision, recall, and the F1, weighted by the occurrence of the various KCF misconfigs in the test set. For convenience, in the remainder of the paper the terms precision, recall, and F1 represent the weighted precision, weighted recall, and weighted F1, respectively. To compute the value of these classification performance metrics, we define their building blocks as follows:

True positives (TPs): Instances where both the LLM and at least one RB tool correctly detected a misconfig.

False positives (FPs): Instances where the LLM erroneously detected a misconfig that was not recognized by any of the RB tools. Note that due to the fact that RB detectors are limited by their given set of rules, it might be the case that a FP is actually a detection of a *variant* of a known misconfig, such that it does not entirely match any of the coded detection rules.

False negatives (FNs): Instances where a misconfig was overlooked by the LLM but detected by one of the RB tools.

True negatives (TNs): Instances in which there was consensus between the LLM and the RB tools as to the absence of misconfig.

5 EXPERIMENTAL RESULTS

5.1 Investigating pretrained LLMs

The first experiment aimed to answer the following question: Are existing pretrained LLMs sufficient for detecting misconfigs in KCFs? To answer this question we experimented with four state-of-the-art pretrained LLMs: GPT-4-Turbo [1, 13], Gemini 1.0 Pro [32, 79], Claude 3 Sonnet [5, 6], and Mistral-7B-Instruct-v0.2 [46, 70]. Currently these models can be used via online chat or API calls but cannot be fine-tuned, so we relied on prompt engineering and few-shot learning [12, 64, 106]. First, we created a fixed few-shot training set by randomly sampling 10 labeled KCFs (the number of labeled KCFs is based on Gemini's maximum allowed input size (in tokens)). To increase variety, we ensured that each of the 10 KCFs had at least two misconfig labels. The same few-shot training set was used with each pretrained LLM. Similarly, the same test set (consisting of 30 randomly sampled test KCFs with the same 39 misconfigs found in the training set) was used to evaluate the LLMs' performance. The prompt we used in this evaluation is presented in Appendix D).

As can be seen in Fig. 3, our experiment with the pretrained LLMs yielded unsatisfactory results: the highest precision, recall, and F1 values obtained were around 0.81, 0.45, and 0.51, respectively. Based on this, we conclude that the task of detecting misconfigs in KCFs might be too narrow for a pretrained general-purpose LLM (as concluded in past studies [15, 16]) and the ability to realize LLMs' potential in this area might require more than the use of a limited set of labeled KCFs in a few-shot learning scheme. In Fig. 3 we can also see that the recall obtained with Mistral is much higher than that achieved by the other pretrained LLMs, while its precision is lower. In classical ML settings, a combination of low precision and high recall typically indicates a classification threshold that is too low [22], i.e., the ML model predicts too many positive labels. However, in our LLM settings, we found that Mistral tends to generate responses that are 2-3 times longer than those of the other LLMs, possibly due to the internal configuration of hyperparameters. In our use case, longer responses necessarily means higher positive misconfig detections (and descriptions), some of which are TPs while others are FPs, resulting in the low precision and high recall obtained by Mistral.

5.2 Selecting the modeling architecture and base model

Given the results obtained in the previous experiment and the need to achieve better performance and broader misconfig coverage, in this subsection, we explore the use of dedicated open-source LLMs with different architectures. These LLMs are optimized and fine-tuned for our task of misconfig detection in KCFs.

We considered three popular LLM architectures: the encoder-only, decoder-only, and encoder-decoder [23, 80, 94]. Among them, we expected the encoder-decoder architecture to be the most suitable for our use case, because (1) the encoder-only architecture

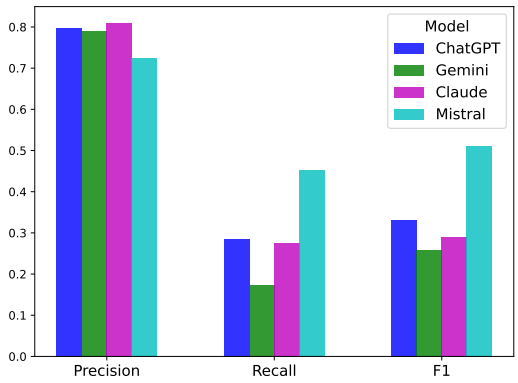


Figure 3: Performance of the pretrained LLMs using few-shot learning for misconfig detection.

Base model	Architecture	Precision	Recall	F1
BigBird [104]	Encoder-only	–	–	–
CodeLlama [85]	Decoder-only	–	–	–
CodeGen [73]	Decoder-only	0.982±0.040	0.574± 0.199	0.705±0.168
LED [10]	Encoder-decoder	0.976±0.035	0.265±0.248	0.368 ± 0.256
CodeT5p 770M [97]	Encoder-decoder	**0.987±0.023**	**0.997±0.018**	**0.992±0.018**
CodeT5p 6B [97]	Encoder-decoder	0.902±0.068	0.390±0.015	0.545±0.026

Table 1: Performance with various LLM architectures

excels in context understanding [44], (2) the decoder-only architecture is well suited for generating tokens [43], and (3) the encoder-decoder architecture performs well at both understanding the context and generating appropriate tokens [45]. Recently, the superiority of the encoder-decoder architecture in a code understanding and completion task was demonstrated [98]; our use case is similar in that (1) a KCF has to be well understood by the LLM, and (2) LLM completion is then used to generate appropriate misconfig labels.

To identify the most promising LLM architecture and base model for KCF misconfig detection, we experimented with several open-source LLMs with the abovementioned architectures (i.e., encoder-only, decoder-only, and encoder-decoder). For efficiency, we fine-tuned each of the LLMs using a stratified random sample of 68,997 labeled KCFs, which represent 30% of the total 229,989 labeled KCFs that comply with the LLMs' token limits. Of these, 85% were allocated for training and 15% for validation. To enable comparison across our experiments, from this point onwards we utilized the same test set defined in Sec. 4.2 (which comprises 22,987 instances that comply with the token limits). As can be seen in Table 1, BigBird (encoder-only) and CodeLlama (decoder-only) failed to converge after fine-tuning, i.e., their output (generated text that serves as KCF labels) did not match the desired misconfig labels, thus we were unable to calculate the metrics for them. In contrast, the other LLMs (which were mainly encoder-decoders, as expected) demonstrated adaptability to the designated task: their output gradually resembled the misconfig labels from the training set, and they managed to obtain noteworthy precision.

Of the evaluated encoder-decoder base models, we chose to proceed with CodeT5p 770M (trained for code understanding and completion [97], which is closely related to our domain), as it obtained the best performance. Also, compared to other encoder-decoder base models, such as the evaluated CodeT5p 6B, it is smaller and thus more affordable in terms of resources and training time. However, one shortcoming of CodeT5p 770M is its upper bound of 512 tokens. In $DS_{labeled}$, about 83% of the 276,785 instances are less than 512 tokens. These 229,989 KCFs and labels, were used in our experiments. The rest of our experiments were conducted using the CodeT5p 770M base model only, and the dataset of 229,989 KCFs which are short enough for this base model.

5.3 Selection of data sources for fine-tuning

To identify the most effective combination of data sources for fine-tuning, we experimented with three data sources:

- **Labeled KCFs**: As in any classification task, model training requires labeled instances which in our use case are KCFs paired with their misconfigs (Sec. 3.1.2).
- **Unlabeled KCFs**: This data source was included as a potential of enhancing the model's ability to understand the typical structure and patterns inherent in KCFs in general (see Sec. 3.1).
- **Free text**: This dataset which comprises 232 documents (700MB) obtained by targeted web crawling, includes various K8s-related material, such as the CIS Benchmarks [17], authoritative literature, and official documentation [26, 47]. This dataset is available in our github repository. It was included as a potential means of broadening GenKubeDetect's comprehension of K8s-specific language and concepts. For that, we explored the use of domain adaptation strategies utilizing CL methods, including techniques such as next sentence prediction (NSP) and masking (see Sec. 2.2).

To identify the optimal data sources and training pipeline that maximize model performance via fine-tuning, we trained four distinct models using the CodeT5p 770M base model. In each setup, the labeled KCFs are used in the last fine-tuning step. As the results presented in Table 2 show, first using unlabeled KCFs and then labeled KCFs resulted in the best performance (maximum mean and/or minimum standard deviation of precision, recall, and F1). This suggests that preliminary exposure to the general structure of KCFs, followed by pairs of KCFs and their misconfig labels, slightly improves the model's ability to detect misconfigs, compared to fine-tuning using labeled KCFs only. Conversely, the introduction of free text appeared to degrade performance slightly, potentially due to its unstructured nature conflicting with the LLM's intended structured inputs (i.e., KCFs). Therefore, it was not included in our proposed method (Sec. 3), which also saves significant amount of time in training (fine-tuning) the model. Notably, adding unlabeled KCFs to free text resulted in improved performance compared to fine-tuning without unlabeled KCFs. Still, it is evident that the best performance (with minimal input overhead) is achieved when fine-tuning is performed using only KCFs, both unlabeled and labeled.

5.4 Comparison to existing RB tools

Having selected the encoder-decoder as our architecture (Sec. 5.2), CodeT5p 770M as our base model (Sec. 5.2), and the combination of unlabeled and labeled KCFs as our data sources (Sec. 5.3), we

Data sources	Precision	Recall	F1
Labeled KCFs	0.983±0.026	0.995±0.043	0.989±0.039
Unlabeled KCFs → labeled KCFs	**0.984±0.027**	**0.995±0.037**	**0.989±0.035**
Free text → labeled KCFs	0.979±0.031	0.988±0.059	0.982±0.058
Free text → unlabeled KCFs → labeled KCFs	0.983±0.027	0.995±0.049	0.988±0.048

Table 2: Performance of GenKubeDetect models fine-tuned using various combinations of data sources

conducted experiments for hyperparameter tuning, such as the learning rate and the weight decay. We restricted the LoRA adaption [40] to 12.5%, r=128, and LoRA_alpha=256. The complete list of the tuned hyperparameters used to assess the performance of our method is provided in our GitHub repository.

Using the optimized hyperparameters and the same training set (comprised of unlabeled KCFs followed by labeled KCFs), we retrained GenKubeDetect's LLM and evalauted it on the test set. The results of our evaluation of its performance, along with that of industry-standard RB tools for KCF misconfig detection (KubeLinter, Checkov and Terrascan), are presented in Table 3. To ensure a fair comparison, each RB tool was evaluated for its ability to detect the *specific subset of misconfigs it is designed to identify*. This approach highlights each tool's strengths without penalizing the tools for undetected misconfigs that fall outside their configured rules. We also evaluate the performance of an ensemble of the three tools (referred to as RB-Ensemble), which covers all of the misconfig subsets they address; this serves as a benchmark for the maximum detection capabilities available.

Tool	Precision	Recall	F1	#Labels
GenKubeDetect	0.990±0.020	0.999±0.026	0.994±0.027	169
KubeLinter	1.0±0.0	0.827±0.355	0.837±0.351	55
Checkov	1.0±0.0	0.851±0.316	0.866±0.319	121
Terrascan	1.0±0.0	0.587±0.471	0.599±0.474	41
RB-Ensemble	1.0±0.0	1.0±0.0	1.0±0.0	169

Table 3: Performance of various tools for KCF misconfig detection: GenKubeDetect vs. individual industry-standard RB tools vs. an ensemble of these RB tools

As seen in Table 3, compared to each RB tool, GenKubeDetect demonstrates superior capabilities on the recall and F1 metrics, with precision that is essentially equivalent. Moreover, GenKubeDetect 's demonstrated ability to detect a larger amount of KCF misconfigs (169 in total) than any of the evaluated RB tools individually, reflects its comprehensiveness and applicability. Unlike RB tools that are confined to a limited number of specific misconfigs and achieve perfect precision but low recall, GenKubeDetect 's approach enables it to identify a large amount of both common and rare misconfigs across various scenarios, with high precision and high recall.

5.5 GenKubeDetect error analysis

In order to better understand GenKubeDetect's FPs (shown in Table 3), we manually inspected them with the help of a K8s security expert. FP misconfigs appeared in 563 KCFs in the test set (note that a KCF may contain one or more misconfigs). Since we could not manually inspect all of these KCFs, we randomly sampled 100 of them, which contain 489 FPs. Our K8s expert thoroughly examined these KCFs and found that 408/489 = 83.44% of the FP misconfigs were actually TPs. These results indicate that GenKubeDetect's precision is actually higher than shown in Table 3. This also means that GenKubeDetect succeeded in effectively generalizing its knowledge base, providing correct insights that surpass the conventional ground truth in its field, set by the ensemble of three industry-standard RB tools. More specifically, GenKubeDetect succeeded in detecting misconfigs that were not caught by any of the RB tools, since they were *variations* of detection rules and thus didn't entirely fit any coded rule.

Appendix E provides examples of two cases of misconfig detections provided by GenKubeDetect that were initially categorized as FPs and later determined by the K8s expert to actually be TPs. For each of these two examples, we present the relevant section of the KCF along with a brief description of the misconfig detected by GenKubeDetect but missed by the RB ensemble. A possible reason for GenKubeDetect's advantage over the RB tools in detecting such misconfigs may stem from both the inherent complexity of KCFs and the advanced capabilities of LLMs; the latter enables the models to learn complex representations of input data [96], thereby enhancing the ability to identify subtle misconfigs that other non-LLM-based tools might miss. LLMs' capacity to generalize from large datasets [72] may also contribute to their effectiveness in diverse and complex environments such as KCFs.

5.6 Training set's size

To characterize GenKubeDetect's learning curve with respect to the size of its training set, we incrementally increased the training set's size, ranging from 50 to 2,000 misconfig instances *for each misconfig label*. In cases in which there was a limited quantity of instances for a misconfig, all available instances were used. Each time, we fine-tuned GenKubeDetect from scratch, using the optimized hyperparameters described throughout this section, and evaluated its performance using the same test set used thus far.

As can be seen in Fig. 4, GenKubeDetect's performance metrics increase rapidly from 0 to around 250 instances and then reach a plateau, converging when there are approximately 1,250 instances from each misconfig label. It is important to note that while larger high-quality datasets typically contribute to better model generalization and performance in the realm of DL in general, and LLM specifically [3, 27, 91], the rate of improvement decreases, eventually becoming marginal. Thus, while our identified threshold of 250 misconfigs instances serves as an effective minimum, larger datasets may still offer incremental benefits.

5.7 Adaptation to new misconfigs

The set of known KCF misconfigs is likely to expand occasionally (mainly when a new configuration-related vulnerability is discovered). To prevent exploitation of a newly-discovered misconfig, detection tools must be adaptable. Traditionally, such adaptation has relied on programmers' in-depth understanding of the detection tool's source code, K8s proficiency, and ability to develop new detection rules or methods to identify the new misconfigs. This process of adaptation to newly-discovered misconfigs is time-consuming and burdensome, which limits the responsiveness of RB tools. In

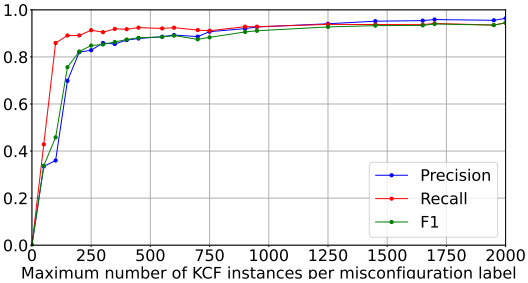


Figure 4: Learning curve analysis for GenKubeDetect based on incremental increases in the training set's size.

contrast, with an LLM-based detection method like ours, all that is needed is further fine-tuning of the trained LLM using a small set of KCFs labeled with the newly-discovered misconfig.

Algorithm 1 Empirical assessment of GenKubeSec's adaptability.

Require: $DS^{adaptation}_{fixed}, LLM^{trained}, DS^{test}$
$M \leftarrow (51, 97, 103, 139, 140)$
$S \leftarrow (1, 2, 5)$
for each $m \in \mathcal{M}$ **do**
  for each $s \in \mathcal{S}$ **do**
    for each $i \in (1, 2, \ldots, 10)$ **do**
      $DS^{adaptation}_{m,s,i} \leftarrow DS^{adaptation}_{fixed} \cup KCF_sample(m, s, i)$
      $LLM_{m,s,i} \leftarrow fine_tune(LLM^{trained}, DS^{adaptation}_{m,s,i})$
      $precision_{m,s,i}, recall_{m,s,i} \leftarrow evaluate(LLM_{m,s,i}, DS^{test})$
    end for
    return $Mean(precision_{m,s,i}), St.Dev.(precision_{m,s,i})$
    return $Mean(recall_{m,s,i}), St.Dev.(recall_{m,s,i})$
  end for
end for

To empirically assess GenKubeDetect's ability to adapt to new misconfigs, we devised an experiment (outlined in Alg. 1) in which we intentionally omitted a set of five randomly selected misconfig_IDs, denoted as M, from the training set. These omitted misconfig_IDs simulate newly-discovered misconfigs, unseen during training. GenKubeDetect's LLM (denoted as $LLM^{trained}$), which was trained without any M-labeled KCFs, simulates an operational detector that is required to adapt as quickly and accurately as possible. We used the same test set as before (denoted as DS^{test}), however to expedite the multiple sub-experiments described next we only used 30% of the data to train and fine-tune $LLM^{trained}$, as described in Sec. 5.2. For each misconfig_ID $m \in M$ and sample size $s \in (1, 2, 5)$, in each iteration $i \in (1, 2, \ldots, 10)$ we took a random sample of s KCFs labeled with the 'new' misconfig_ID m. This sample, denoted as $KCF_sample(m, s, i)$, was then merged with a randomly sampled fixed set of 50 'old' KCFs (denoted as $DS^{adaptation}_{fixed}$), resulting in an adaptation dataset $DS^{adaptation}_{m,s,i}$ comprised of labeled KCFs of both old and new misconfig labels. This approach was chosen to prevent the fine-tuned model in each iteration $LLM_{m,s,i}$ from being biased towards the new misconfig_ID m. After each iteration, defined by the unique combination of m, s, and i, the precision and recall of $LLM_{m,s,i}$ were evaluated using DS^{test}. Then, for each combination of m and s, the mean and standard deviation of the precision and recall were calculated. The results of this experiment are presented in Table 4.

		Fine-tuning (for adaptation)			Full training	
	m	s=1	s=2	s=5	30% data	80% data
Precision	51	0.083±0.180	0.444±0.416	0.445±0.113	1	1
	97	0.433±0.498	0.263±0.404	0.366±0.090	0.579	0.923
	103	0.475±0.506	0.817±0.317	0.857±0.000	1	1
	139	0.875±0.317	0.944±0.118	0.943±0.074	1	1
	140	0.994±0.013	0.951±0.046	0.942±0.040	0.967	0.972
Recall	51	0.005±0.010	0.048±0.054	0.256±0.153	1	1
	97	0.042±0.044	0.083±0.096	0.467±0.143	0.917	1
	103	0.167±0.208	0.574±0.374	1	1	1
	139	0.417±0.180	0.685±0.242	1	1	1
	140	0.826±0.103	0.956±0.045	0.969±0.076	1	1

Table 4: GenKubeDetect's precision and recall for selected misconfig_IDs (m) as a function of the size of the sample (s) in fine-tuning (left side) or the size of the training set in full training (right side).

Table 4 shows a promising learning curve in the context of GenKubeDetect's ability to quickly and accurately adapt to newly-discovered KCF misconfigs. The empirical results show that to achieve a precision level of 0.8 or more, only two m-labeled KCFs (i.e., s=2) are required in 60% of cases and just one m-labeled KCF is needed in 40% of cases. Satisfactory recall levels require slightly more m-labeled KCFs, however in 60% of cases, five m-labeled KCFs are sufficient to achieve near-zero FN rates. These empirical results demonstrate our LLM-based method's adaptability and effectiveness in continuously evolving environments, which are maintained in such settings *without compromising GenKubeDetect's performance* or increasing the demands on RB tool developers.

When looking closer at the results presented in Table 4 we can see that some misconfig_IDs (i.e., 51 and 97) suffer from many FNs and FPs when only a handful of them are used to further fine-tune a trained LLM. In further analysis of these misconfig_IDs, we found that both of the associated misconfigs are extremely similar to other misconfigs, and thus they tend to be misclassified. More specifically, misconfig_ID 97 is defined under 'securityContext/capabilities/*add*,' while several other misconfig_IDs are defined under 'securityContext/capabilities/*drop*.' This makes it difficult for the LLM, which has already learned from the old misconfigs, to distinguish between them based such a small sample of new KCFs. Similarly, the fine-tuned LLM frequently failed to recognize misconfig_ID 51 and confused it with another misconfig_ID, because they are both associated with role-based access control (RBAC). Still, as can be seen on the right side of Table 4, when GenKubeDetect is trained with more data (30 or 80% of the M-labeled KCFs), the precision and recall using DS^{test} converge to one. Also, the gap between having a small or large set of m-labeled KCFs (denoted as $KCF_sample(m)$) can be bridged by initially fine-tuning $LLM^{trained}$ using whatever size of $KCF_sample(m)$ is available. Then, using LLM_m (i.e., $LLM^{trained}$ which was further fine-tuned using $DS^{adaptation}_{m,s}$), we can scan other KCFs, either locally available or obtained on the Internet to gather additional m-labeled KCFs. With this approach, even an LLM_m with relatively low recall and/or precision can increase the size of $KCF_sample(m)$ by gathering KCFs that have the newly-discovered m but have not yet been labeled as such. Then, a second

round of fine-tuning (using the increased $KCF_sample(m)$) would enable the GenKubeDetect's quick and accurate adaptation to the newly-discovered misconfig.

5.8 Localization, reasoning, and remediation

Manually validating GenKubeResolve's output for each misconfig detected is a time-consuming task that should be performed by an expert. Therefore, a K8s security expert validated GenKubeResolve's performance on a random sample of 30 detected KCF misconfigs (this took approximately six minutes for each misconfig). The K8s expert confirmed that GenKubeResolve excels in these explanatory tasks, achieving a score of 30/30 in terms of both its explanations and remediation suggestions. In terms of localization, in 20 of the 30 cases, GenKubeResolve correctly identified that necessary lines were missing in the provided KCFs, and therefore could not provide a line number. In the remaining 10 cases, GenKubeResolve accurately pinpointed the issues in the correct lines.

6 DISCUSSION

6.1 Lessons learned

During our research, the following valuable insights emerged. First, similar to previous research [60, 69], using the basic prompt engineering and few-shot learning techniques to adapt pretrained LLMs for KCF misconfig detection results in relatively poor performance (Sec. 5.1). For this highly specific task of multi-label classification within (semi-structured) KCFs, we found that *a thoroughly optimized and fine-tuned LLM is capable of achieving near-perfect performance*, with precision equivalent to industry standards and superior recall. In comparison, for the task of KCF misconfig localization, reasoning, and remediation, our experimental results (validated by a K8s expert) showed that few-shot learning with a pretrained model is sufficient. Hence, it seems that *reconstructing specific misconfig_IDs is much more difficult for an LLM than generating less restrictive text for misconfig localization, reasoning and remediation.*

Second, it seems that with a suitable LLM architecture, a domain-relevant base model [97], and optimized hyperparameters, *relatively-small quantities of training data are sufficient for GenKubeDetect to achieve satisfactory performance.* This applies both to the initial fine-tuning whose learning curve achieves high precision and recall (about 83% and 91%, respectively) when there are 250 KCFs per misconfig (Fig. 4), and to the adaptation process, which, in most cases, requires only a handful of relevant KCFs in order to be able to detect a newly-discovered misconfig (Table 4).

A third lesson learned during our experimentation is that *when an LLM is used for classification, encoding (compressing) the textual label is beneficial.* Most importantly, with a given amount of labeled KCFs in the training set, predicting an encoded misconfig label may improve the classification performance (longer texts are harder to reconstruct). Furthermore, a reduction in the number of processed tokens (during GenKubeDetect's training and inference) directly translates to accelerated and less expensive compute.

A fourth insight, which was counter-intuitive for us, is that *during GenKubeDetect's fine tuning, focusing exclusively on KCFs maximizes the precision, recall, and F1* (Table 2). That is, incorporating free-text information about K8s along with KCFs does not enhance performance on these metrics, but rather actually impairs it slightly. This finding suggests that (1) when the LLM is designed to analyze structured data (i.e., KCFs), there is no use in training it using unstructured data; (2) fewer inputs (i.e., KCFs only) are sufficient for achieving accurate outputs; and (3) labeled KCFs are essentially sufficient for realizing the LLM's potential in detecting KCF misconfigs, as the contribution of unlabeled KCFs is negligible.

6.2 Research limitations

In our empirical evaluation, the best-performing base LLM (CodeT5p 770M) places a 512 token limit, such that only 83% of the KCFs in our dataset are covered. Nevertheless, CodeT5p 770M may be replaced by a less restrictive base LLM in the future. Another limitation is that during error analysis (see Sec. 5.5 and Sec. 5.8), the expert manually only analyzed a sample of 100 KCFs which contained alleged FPs, and 30 randomly selected outputs of GenKubeResolve. While not all of the KCFs were examined in our manual error analysis, a non-negligable portion of those examined demonstrated GenKubeDetect's ability to outperform existing RB tools and GenKubeResolve's ability to produce valuable and correct responses.

7 RELATED WORK

7.1 The use of LLMs in static analysis

LLMs such as ChatGPT [1], Bard [31], Gemini Pro [79], GPT-4, and GPT-3.5 have been leveraged extensively in the software and hardware security analysis domain, uncovering both promising applications and notable limitations. Several recent studies [2, 59, 61, 65, 103] demonstrated LLMs' efficacy in static analysis tasks like correcting defective Ansible scripts, detecting code security defects, detecting and repairing security bugs in hardware description languages, and automating static binary taint analysis. These tasks are similar to our task of KCF misconfig detection in that they use LLMs for static analysis. These studies theoretically suggested the need to fine-tune LLMs for specific applications to optimize performance. However in contrast to those studies, we evaluated the impact of fine-tuning on performance quantitatively. Other studies [63, 78] examined the ML's integration with runtime DevSecOps and the automation of configuration validation. These studies not only underscored the adaptability of generative AI in security contexts but also emphasized the need for further refinement, such as fine-tuning, to fully exploit generative AI technologies' adaptability in these complex applications. Our research contributes uniquely to this field; to the best of our knowledge, we are the first to employ a fine-tuned LLM specifically for robust and cost-efficient static analysis of KCFs, detecting KCF misconfigs, and providing detailed reasoning regarding the misconfigs, their exact localization, and remediation suggestions.

7.2 KCF misconfig detection

Recently, topology graphs were proposed [11] as a means of revealing attack vectors in security configurations and assigning them security risk scores. This approach helps detect KCF misconfigs by mapping the K8s' deployment, identifying vulnerabilities, and highlighting security weaknesses. However, it faces challenges in scalability and practical application, requires in-depth K8s expertise, occasionally fails to scan certain containers, and lacks automated

remediation suggestions. In another study [82], SLI-KUBE, an innovative RB tool for KCF misconfig detection, was proposed. However, in terms of coverage, SLI-KUBE can only detect 11 specific misconfigs using its predefined rule set, while our method was empirically shown capable of detecting 169 KCF misconfigs (see Sec. 5). In addition, our method's use of fine-tuning enables it to quickly and easily adapt to newly-discovered misconfigs, based on just a few KCFs labeled with the new misconfig (Sec. 5.7). Fine-tuning also offers a substantial advantage over traditional RB tools, which suffer from a non-negligible incidence of FPs [69] and are limited due to their reliance on static rule sets which do not adapt to new misconfigs without manual updates.

7.3 KCF misconfig remediation

While existing tools can detect KCF misconfigs, they often lack the ability to suggest effective corrective measures. Our method is unique, since in addition to detecting misconfigs in KCFs, it also provides detailed reasoning, as well as the specific locations of detected misconfigs and remediation. This aids greatly in understanding the root cause of misconfigs, enabling more effective and targeted fixes.

In a preliminary study [69], LLMs' potential in correcting security misconfigs in K8s Helm charts was explored. Existing RB tools (similar to the ones used in our study) were employed to detect misconfigs; then LLMs were used to suggest mitigation steps aimed at enhancing configuration. Thus, unlike in our study, LLMs were not considered a direct replacement for RB tools but rather as a means of correcting any misconfigs they detect (potentially inadvertently introducing new misconfigs in their attempt to correct others). Moreover, the LLMs used in that study are ChatGPT and Gemini, which are general-purpose pretrained LLMs that have not undergone fine-tuning. Another drawback is that sending files for inspection by external LLMs via API calls, as done in that study, poses security risks [8, 25] and adds unnecessary costs. To overcome these issues, in our research, we used an LLM developed by Mistral [46] which is open-source (and thus free of charge), does not require external API interactions, and can be adapted to meet our specific needs. In addition, in the evaluation of this preliminary study, the authors used a limited sample of only 60 Helm charts, whereas we collected and labeled over 276,000 KCFs to evaluate our proposed method.

Another study [60] explored LLMs' potential for analyzing KCFs. The authors proposed a pipeline to identify whether the KCF is of 'good' or 'bad' quality. They used Polaris [29] to identify one specific misconfig, and leveraged an LLM to perform few-shot learning, assessing the presence of this misconfig in other KCFs, and recommending best practices for developers. Despite similarities to our research, some limitations and differences between the two studies are worth noting: (1) The authors used a small dataset of only 100 KCFs, which is much smaller than our dataset of over 276,000 KCFs. (2) For training, the authors relied on few-shot learning [12, 64, 106], which, as they acknowledged, restricts the LLM's ability to learn and generalize complex relationships from the data. In contrast, we performed fine-tuning with a significantly larger dataset, allowing the model to better capture and generalize complex relationships. (3) They focused on binary classification [49], determining the presence or absence of a specific misconfig in KCFs, thus providing limited feedback to developers; in contrast, our method is trained to detect up to 169 misconfigs. (4) Our approach not only identifies misconfigs but also pinpoints their location, explains their reasons, and offers targeted suggestions for remediation. (5) Unlike the prior study, which did not include results and evaluation metrics, we have comprehensively described our experimentation, evaluation, and findings, and publicly share our data and code.

In another study whose goals are also relatively similar to ours [39], the authors proposed a completely different approach. Specifically, they suggested the use of free text documents (mostly from blog posts) to infer a knowledge graph, which was used to identify misconfigs and provide remediation. However, unlike our research, their focus was on matching a given textual vulnerability description to a configuration concept, i.e., extracting knowledge from free text. A shortcoming in this research noted by the authors is that the method's performance relies heavily on the free text documents' comprehensiveness, as well as on the quality of their text processing procedure; our proposed method does not face these limitations. In addition, their evaluation was performed on a much smaller scale than ours, and configuration concepts were manually labeled by the researchers, a method which the authors acknowledged as subjective (unlike the vast amount of KCFs we labeled using three unbiased industry-standard RB tools).

8 CONCLUSION

In this research, we explored LLMs' ability to enhance the security of K8s systems. For that, GenKubeSec does more than merely detect (a wide variety of) KCF misconfigs; it provides their exact location in the inspected KCFs, provides clear reasoning from which the KCF developer can learn, and offers actionable recommendations for remediation. GenKubeSec does not rely on external APIs, thus ensuring that KCFs are not exposed on the web, enhancing security; this approach also reduces costs. Unlike previous studies, we developed a comprehensive LLM-based method, and quantitatively evaluated it. Our extensive evaluation demonstrated GenKubeSec's excellent performance, validated by a K8s expert. In addition to contributing to containers' security via mitigating KCF misconfig-related risks, by making our data (including the UMI) and code freely accessible, this work will contribute to future advancements in the field, specifically in terms of standardization, benchmarking, and security-oriented, LLM-based analysis and remediation methods.

Future work in this domain can focus on (1) automatically generating a misconfig-free version of the input KCF to eliminate the detected misconfigs, and (2) enhancing GenKubeSec's ability to prioritize the detected misconfigs based on their severity level.

REFERENCES

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. 2024. On Hardware Security Bug Code Fixes By Prompting Large Language Models. *IEEE Transactions on Information Forensics and Security* (2024).

[3] Md Zahangir Alom, Tarek M Taha, Chris Yakopcic, Stefan Westberg, Paheding Sidike, Mst Shamima Nasrin, Mahmudul Hasan, Brian C Van Essen, Abdul AS Awwal, and Vijayan K Asari. 2019. A state-of-the-art survey on deep learning theory and architectures. *electronics* 8, 3 (2019), 292.

[4] Zaid Alyafeai, Maged Saeed AlShaibani, and Irfan Ahmad. 2020. A survey on transfer learning in natural language processing. *arXiv preprint arXiv:2007.04239* (2020).

[5] Anthropic. 2024. *Claude 3*. Anthropic. https://www.anthropic.com/news/claude-3-family

[6] anthropic. 2024. *Claude Soonet*. Anthropic. https://claude.ai/chats

[7] Vladimir Araujo, Marie-Francine Moens, and Alvaro Soto. 2023. Learning Sentence-Level Representations with Predictive Coding. *Machine Learning and Knowledge Extraction* 5, 1 (2023), 59–77.

[8] Muhammad Azizi Mohd Ariffin, Mohd Faisal Ibrahim, and Zolidah Kasiran. 2020. API vulnerabilities in cloud computing platform: attack and detection. *International Journal of Engineering Trends and Technology* 1 (2020), 8–14.

[9] ARMO. 2022. ARMO study. https://thenewstack.io/armo-misconfiguration-is-number-1-kubernetes-security-risk/

[10] Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. *arXiv:2004.05150* (2020).

[11] Agathe Blaise and Filippo Rebecchi. 2022. Stay at the Helm: secure Kubernetes deployments via graph generation and attack reconstruction. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. IEEE, 59–69.

[12] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[13] ChatGPT. 2024. *ChatGPT - GPT 4*. ChatGPT. https://chatgpt.com/

[14] checkov. 2024-02-03. *checkov*. https://www.checkov.io

[15] Yufan Chen, Arjun Arunasalam, and Z Berkay Celik. 2023. Can large language models provide security & privacy advice? measuring the ability of llms to refute misconceptions. In *Proceedings of the 39th Annual Computer Security Applications Conference*. 366–378.

[16] Anton Cheshkov, Pavel Zadorozhny, and Rodion Levichev. 2023. Evaluation of chatgpt model for vulnerability detection. *arXiv preprint arXiv:2304.07232* (2023).

[17] CIS. 2024. CIS-Benchmark. https://www.cisecurity.org/benchmark/kubernetes

[18] Cloud Native Computing Foundation. 2024. Artifact Hub. https://artifacthub.io/ Accessed: 2024-07-04.

[19] CNCF. 2022. Workload misconfiguration – the 1 security threat when using Kubernetes. https://www.cncf.io/online-programs/cloud-native-live-workload-misconfiguration-the-1-security-threat-when-using-kubernetes/

[20] CNCF. 2023. Project Journey Report. https://www.gartner.com/en/documents/3988026

[21] CNCF. 2023. Project Journey Report. https://www.cncf.io/reports/kubernetes-project-journey-report/

[22] Jesse Davis and Mark Goadrich. 2006. The relationship between Precision-Recall and ROC curves. In *Proceedings of the 23rd international conference on Machine learning*. 233–240.

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[24] Practical DevSecOps. 2023. 10+ Top Kubernetes Security Tools in 2023. https://medium.com/@pdevsecops/10-top-kubernetes-security-tools-in-2023-df26642f995a

[25] Josué Alejandro Díaz-Rojas, Jorge Octavio Ocharán-Hernández, Juan Carlos Pérez-Arriaga, and Xavier Limón. 2021. Web api security vulnerabilities and mitigation mechanisms: A systematic mapping study. In *2021 9th International Conference in Software Engineering Research and Innovation (CONISOFT)*. IEEE, 207–218.

[26] dohsimpson. 2023. kubernetesDoc. https://github.com/dohsimpson/kubernetes-doc-pdf

[27] Shi Dong, Ping Wang, and Khushnood Abbas. 2021. A survey on deep learning and its applications. *Computer Science Review* 40 (2021), 100379.

[28] Nicolas Ehrman. 2023. The top 11 open-source Kubernetes security tools. https://www.wiz.io/academy/top-kubernetes-security-tools

[29] Fairwinds. 2024-08-01. *Polaris*. https://www.fairwinds.com/polaris

[30] Robert M French. 1999. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences* 3, 4 (1999), 128–135.

[31] Google. 2023. An important next step on our AI journey. https://blog.google/technology/ai/bard-google-ai-search-updates/

[32] Google. 2024. *Gemini 1.0 Pro*. https://gemini.google.com/app

[33] Google. 2024. Google. https://cloud.google.com/learn/what-is-kubernetes

[34] Sagar Goyal, Eti Rastogi, Sree Prasanna Rajagopal, Dong Yuan, Fen Zhao, Jai Chintagunta, Gautam Naik, and Jeff Ward. 2024. HealAI: A Healthcare LLM for Effective Medical Documentation. In *Proceedings of the 17th ACM International Conference on Web Search and Data Mining*. 1167–1168.

[35] James Guffey and Yanyan Li. 2023. Cloud Service Misconfigurations: Emerging Threats, Enterprise Data Breaches and Solutions. In *2023 IEEE 13th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, 0806–0812.

[36] Muhammad Usman Hadi, Rizwan Qureshi, Abbas Shah, Muhammad Irfan, Anas Zafar, Muhammad Bilal Shaikh, Naveed Akhtar, Jia Wu, Seyedali Mirjalili, et al. 2023. Large language models: a comprehensive survey of its applications, challenges, limitations, and future prospects. (2023).

[37] Ibrahim Bu Haimed, Marwan Albahar, and Ali Alzubaidi. 2023. Exploiting Misconfiguration Vulnerabilities in Microsoft's Azure Active Directory for Privilege Escalation Attacks. *Future Internet* 15, 7 (2023), 226.

[38] Md Sadun Haq, Ali Saman Tosun, and Turgay Korkmaz. 2024. LUCID: A Framework for Reducing False Positives and Inconsistencies Among Container Scanning Tools. arXiv:2405.07054 [cs.CR]

[39] Mubin Ul Haque, M Mehdi Kholoosi, and M Ali Babar. 2022. KGSecConfig: A Knowledge Graph Based Approach for Secured Container Orchestrator Configuration. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 420–431.

[40] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685* (2021).

[41] Wenbo Hu, Yifan Xu, Yi Li, Weiyue Li, Zeyuan Chen, and Zhuowen Tu. 2024. Bliva: A simple multimodal llm for better handling of text-rich visual questions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 2256–2264.

[42] HuggingFace. 2023. PEFT - LoRA. https://huggingface.co/docs/peft/package_reference/lora

[43] huggingface. 2024. huggingfaceDecoder. https://huggingface.co/learn/nlp-course/chapter1/6?fw=pt

[44] huggingface. 2024. huggingfaceEncoder. https://huggingface.co/learn/nlp-course/chapter1/5?fw=pt

[45] huggingface. 2024. huggingfaceEncoderDecoder. https://huggingface.co/learn/nlp-course/chapter1/7?fw=pt

[46] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7B. *arXiv preprint arXiv:2310.06825* (2023).

[47] k8s. 2024. KubeDoc. https://kubernetes.io/docs/home/

[48] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The Stack: 3 TB of permissively licensed source code. *Preprint* (2022).

[49] Brian Kolo. 2011. *Binary and multiclass classification*. Lulu. com.

[50] Pandu Ranga Reddy Konala, Vimal Kumar, and David Bainbridge. 2023. SoK: Static Configuration Analysis in Infrastructure as Code Scripts. In *2023 IEEE International Conference on Cyber Security and Resilience (CSR)*. IEEE, 281–288.

[51] Yogita Kothadiya. 2022. A Manifest File in Kubernetes. https://medium.com/@yogitakothadia/a-manifest-file-in-kubernetes-952183a508d4

[52] NEGAR MOHAMMADI KOUSHKI, IBRAHIM EL-SHEKEIL, and KRISHNA KANT. 2024. Root-Cause Analysis of Service Misconfigurations in Enterprise Systems. (2024).

[53] kubelinter. 2024-02-03. *kubelinter*. https://docs.kubelinter.io

[54] Kubernetes. 2024. The Kubernetes community. https://kubernetes.io/community/ Accessed: 2024-08-04.

[55] kubernetes. 2024. static pod. https://kubernetes.io/docs/tasks/configure-pod-container/static-pod/

[56] T Kubernetes. 2019. Kubernetes. *Kubernetes. Retrieved May* 24 (2019), 2019.

[57] Kubernetes-documentation. 2023. Kubernetes. https://kubernetes.io/docs/concepts/overview/working-with-objects/

[58] Pankaj Kumar. 2023. Kubernetes Misconfigurations: The Silent Killers of Your Cluster. https://www.askpython.com/resources/kubernetes-misconfigurations-silent-killers-of-cluster

[59] Sunjae Kwon, Sungu Lee, Taehyoun Kim, Duksan Ryu, and Jongmoon Baik. 2023. Exploring LLM-Based Automated Repairing of Ansible Script in Edge-Cloud Infrastructures. *Journal of Web Engineering* 22, 6 (2023), 889–912.

[60] Giacomo Lanciano, Manuel Stein, Volker Hilt, Tommaso Cucinotta, et al. 2023. Analyzing Declarative Deployment Code with Large Language Models. *CLOSER* 2023 (2023), 289–296.

[61] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. Assisting static analysis with large language models: A chatgpt experiment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2107–2111.

[62] Xin Li, Lidong Bing, Wenxuan Zhang, and Wai Lam. 2019. Exploiting BERT for end-to-end aspect-based sentiment analysis. *arXiv preprint arXiv:1910.00883* (2019).

[63] Xinyu Lian, Yinfang Chen, Runxiang Cheng, Jie Huang, Parth Thakkar, and Tianyin Xu. 2023. Configuration Validation with Large Language Models. *arXiv preprint arXiv:2310.09690* (2023).

[64] Xi Victoria Lin, Todor Mihaylov, Mikel Artetxe, Tianlu Wang, Shuohui Chen, Daniel Simig, Myle Ott, Naman Goyal, Shruti Bhosale, Jingfei Du, et al. 2022. Few-shot learning with multilingual generative language models. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. 9019–9052.

[65] Puzhuo Liu, Chengnian Sun, Yaowen Zheng, Xuan Feng, Chuan Qin, Yuncheng Wang, Zhi Li, and Limin Sun. 2023. Harnessing the power of llm to support binary taint analysis. *arXiv preprint arXiv:2310.08275* (2023).

[66] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[67] Yun Luo, Zhen Yang, Fandong Meng, Yafu Li, Jie Zhou, and Yue Zhang. 2023. An empirical study of catastrophic forgetting in large language models during continual fine-tuning. *arXiv preprint arXiv:2308.08747* (2023).

[68] Microsoft Threat Intelligence. 2024. Attackers exploiting new critical OpenMetadata vulnerabilities on Kubernetes clusters. https://www.microsoft.com/en-us/security/blog/2024/04/17/attackers-exploiting-new-critical-openmetadata-vulnerabilities-on-kubernetes-clusters/ Accessed: 2024-01-05.

[69] Francesco Minna, Fabio Massacci, and Katja Tuma. 2024. Analyzing and Mitigating (with LLMs) the Security Misconfigurations of Helm Charts from Artifact Hub. *arXiv preprint arXiv:2403.09537* (2024).

[70] Mistral. 2024. *mistralai: Mistral-7B-Instruct-v0.2.* https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.2

[71] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Nick Barnes, and Ajmal Mian. 2023. A comprehensive overview of large language models. *arXiv preprint arXiv:2307.06435* (2023).

[72] Behnam Neyshabur, Srinadh Bhojanapalli, David McAllester, and Nati Srebro. 2017. Exploring generalization in deep learning. *Advances in neural information processing systems* 30 (2017).

[73] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations*.

[74] nvidia. 2024. rtx4090. https://www.nvidia.com/en-eu/geforce/graphics-cards/40-series/rtx-4090/

[75] nvidia. 2024. rtx6000. https://www.nvidia.com/en-us/design-visualization/rtx-6000/

[76] Ralph Peeters and Christian Bizer. 2023. Entity Matching using Large Language Models. *arXiv preprint arXiv:2310.11244* (2023).

[77] Ralph Peeters and Christian Bizer. 2023. Using chatgpt for entity matching. In *European Conference on Advances in Databases and Information Systems*. Springer, 221–230.

[78] Nenad Petrović. 2023. Machine learning-based run-time DevSecOps: ChatGPT against traditional approach. In *2023 10th International Conference on Electrical, Electronic and Computing Engineering (IcETRAN)*. IEEE, 1–5.

[79] Sundar Pichai and Demis Hassabis. 2023. Introducing Gemini: our largest and most capable AI model. *Google. Retrieved December* 8 (2023), 2023.

[80] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. [n. d.]. Language Models are Unsupervised Multitask Learners. ([n. d.]).

[81] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research* 21, 140 (2020), 1–67.

[82] Akond Rahman, Shazibul Islam Shamim, Dibyendu Brinto Bose, and Rahul Pandita. 2023. Security misconfigurations in open source kubernetes manifests: An empirical study. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–36.

[83] Monika Rangta et al. 2022. *Tools for Security Auditing and Hardening in Microservices Architecture.* Master's thesis.

[84] Subendhu Rongali, Abhyuday Jagannatha, Bhanu Pratap Singh Rawat, and Hong Yu. 2020. Continual domain-tuning for pretrained language models. *arXiv preprint arXiv:2004.02288* (2020).

[85] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[86] Areeg Samir and Håvard Dagenborg. 2023. Adaptive Controller to Identify Misconfigurations and Optimize the Performance of Kubernetes Clusters and IoT Edge Devices. In *European Conference on Service-Oriented and Cloud Computing*. Springer, 170–187.

[87] Mohit Sewak, Vamsi Emani, and Annam Naresh. 2023. CRUSH: Cybersecurity Research using Universal LLMs and Semantic Hypernetworks. (2023).

[88] substratusai. 2024. substratusai/the-stack-yaml-k8s. https://huggingface.co/datasets/substratusai/the-stack-yaml-k8s

[89] Chi Sun, Xipeng Qiu, Yige Xu, and Xuanjing Huang. 2019. How to fine-tune bert for text classification?. In *Chinese computational linguistics: 18th China national conference, CCL 2019, Kunming, China, October 18–20, 2019, proceedings 18*. Springer, 194–206.

[90] Terrascan. 2024-04-01. *Terrascan.* https://runterrascan.io

[91] Vajira Thambawita, Inga Strümke, Steven A Hicks, Pål Halvorsen, Sravanthi Parasa, and Michael A Riegler. 2021. Impact of image resolution on deep learning performance in endoscopy image classification: An experimental study using a large dataset of endoscopic images. *Diagnostics* 11, 12 (2021), 2183.

[92] Ankit Amrendra Tripathi. 2024. *Attacking and Defending Kubernetes.* Ph. D. Dissertation. Dublin Business School.

[93] Kanishka Tyagi, Chinmay Rane, Raghavendra Sriram, and Michael Manry. 2022. Unsupervised learning. In *Artificial intelligence and machine learning for edge computing*. Elsevier, 33–52.

[94] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[95] Liyuan Wang, Xingxing Zhang, Hang Su, and Jun Zhu. 2023. A comprehensive survey of continual learning: Theory, method and application. *arXiv preprint arXiv:2302.00487* (2023).

[96] Xizhao Wang, Yanxia Zhao, and Farhad Pourpanah. 2020. Recent advances in deep learning. *International Journal of Machine Learning and Cybernetics* 11 (2020), 747–750.

[97] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

[98] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[99] Garrett Wilson and Diane J Cook. 2020. A survey of unsupervised deep domain adaptation. *ACM Transactions on Intelligent Systems and Technology (TIST)* 11, 5 (2020), 1–46.

[100] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*. 38–45.

[101] Tongtong Wu, Massimo Caccia, Zhuang Li, Yuan-Fang Li, Guilin Qi, and Gholamreza Haffari. 2021. Pretrained language model in continual learning: A comparative study. In *International conference on learning representations*.

[102] Yifan Yao, Jinhao Duan, Kaidi Xu, Yuanfang Cai, Zhibo Sun, and Yue Zhang. 2024. A survey on large language model (llm) security and privacy: The good, the bad, and the ugly. *High-Confidence Computing* (2024), 100211.

[103] Jiaxin Yu, Peng Liang, Yujia Fu, Amjed Tahir, Mojtaba Shahin, Chong Wang, and Yangxiao Cai. 2024. Security Code Review by LLMs: A Deep Dive into Responses. *arXiv preprint arXiv:2401.16310* (2024).

[104] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. 2020. Big bird: Transformers for longer sequences. *Advances in Neural Information Processing Systems* 33 (2020).

[105] Hanqing Zhang, Haolin Song, Shaoyu Li, Ming Zhou, and Dawei Song. 2023. A survey of controllable text generation using transformer-based pre-trained language models. *Comput. Surveys* 56, 3 (2023), 1–37.

[106] Xuan Zhang, Navid Rajabi, Kevin Duh, and Philipp Koehn. 2023. Machine translation with large language models: Prompting, few-shot learning, and fine-tuning with qlora. In *Proceedings of the Eighth Conference on Machine Translation*. 468–481.

A ENTITY MATCHING PROMPT

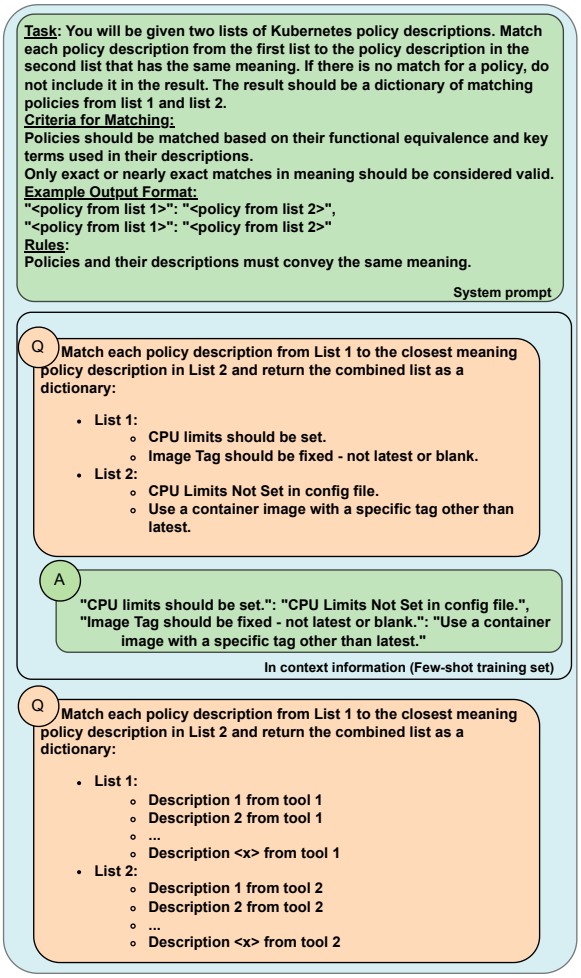


Figure 5: Entity matching prompt, used during the UMI creation process, as part of GenKubeDetect's preprocessing component.

Figure. 5 outlines the structure of the optimized prompt we used for entity matching while creating the UMI (Sec. 3.1.1. This prompt has three segments: (1) a system prompt which describes the entity matching task, the criteria for matching, and the structure of the required output; (2) a few-shot training set; and (3) new lists comprising entities to be matched. We used this prompt in order to map all of the variations of a misconfig (from three different RB tools) to a single ID.

B ENCODED AND DECODED CLASS LABELS

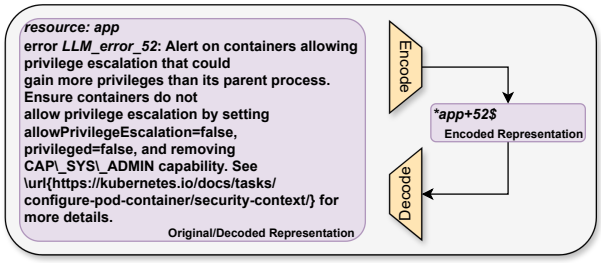


Figure 6: Encoded and decoded representations of an example misconfig class label.

Figure. 6 illustrates the encoded and decoded representations of an example misconfig class label (as discussed in Sec. 3.1.2). The original/decoded representation contains detailed information about a misconfig related to privilege escalation in K8s containers. This relatively long textual description is encoded into the compressed format: <impacted_K8s_resource>+<misconfig_ID>, in this case app+52. Encoding the labels decreases the overall size of labeled KCFs, such that (1) higher coverage is provided by LLMs that impose size limits, (2) training is accelerated, and (3) fewer misclassifications are likely to occur, as shorter class labels are easier for the LLM to (re-)generate. Decoding the labels provides more context to GenKubeResolve during inference (see Sec. 3.3.1).

C GENKUBERESOLVE PROMPT

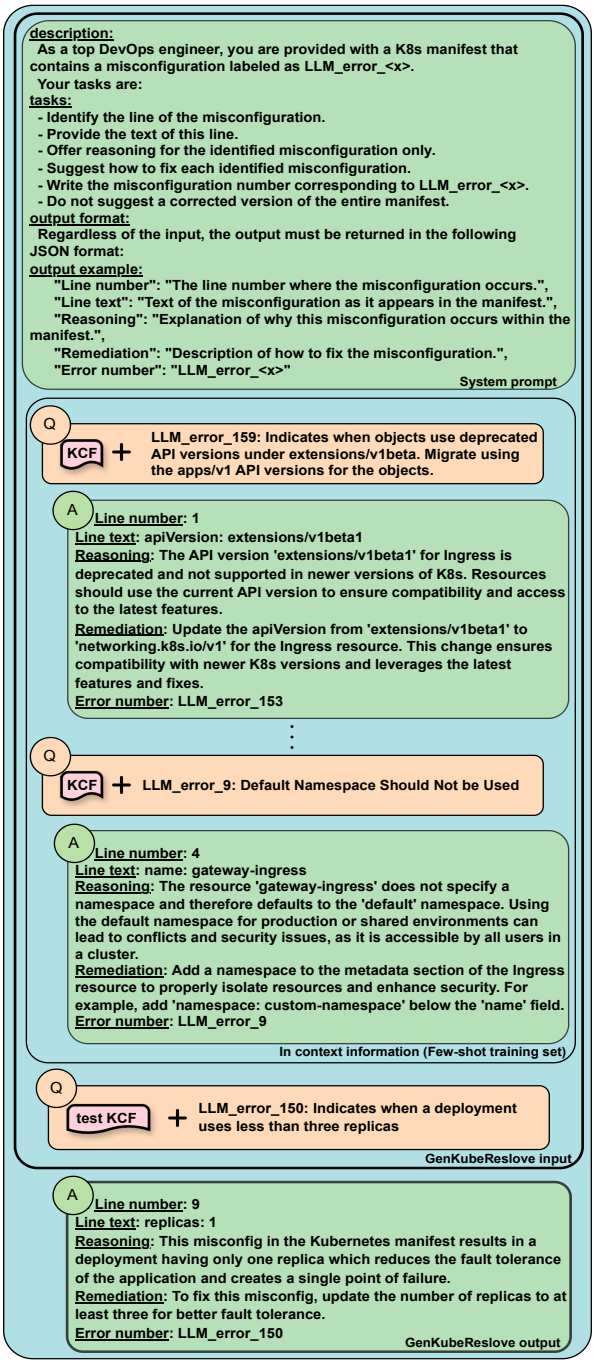


Figure 7: The system prompt devised for the Mistral LLM and an example of a few-shot learning process for localization, reasoning, and remediation of a detected KCF misconfig.

Figure. 7 outlines the structure of the optimized prompt we used in GenKubeResolve for providing localization, reasoning and remediation suggestions to each misconfig detected by GenKubeDetect (see further details in Sec. 3.3.2). This prompt has three segments: (1) a system prompt which describes the required outputs and their format; (2) a few-shot training set; and (3) a test KCF to be processed by GenKubeResolve. At the bottom we can see GenKubeResolve's output for this test KCF: (1) the exact location of the detected misconfig is line 9 (the current text within this line is 'replicas:1'); (2) the reason for alerting is that the current configuration results in a single point of failure; (3) for remediation, the KCF developer is advised to increase the number of replicas.

D PROMPT USED WITH THE GENERAL PURPOSE PRETRAINED LLMS FOR MISCONFIG DETECTION

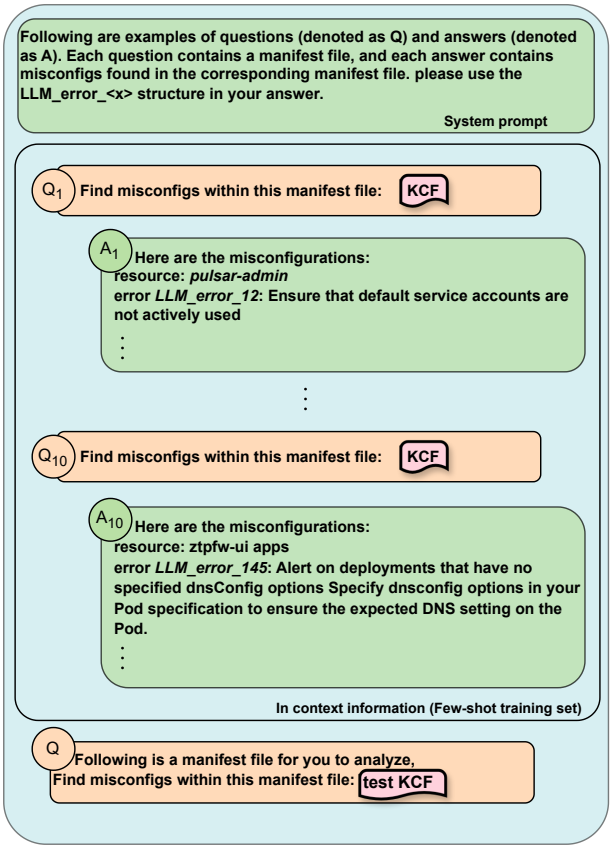


Figure 8: General instruction (top), 2 out of the 10 few-shot learning pairs of KCFs and their misconfig labels (middle), and a test KCF for which the LLM is requested to generate misconfig labels (bottom).

Figure. 8 outlines the structure of the optimized prompt we used with pretrained general-purpose LLMs for them to be able to detect misconfigs in KCFs. This prompt has three segments: (1) a system prompt which describes the task; (2) a few-shot training set; and

(3) a test KCF to be processed by the pretrained LLMs. The experimental results eventually showed that more sophisticated LLM training techniques (mostly fine-tuning) are required for reaching satisfactory detection performance (see further details in Sec. 5.1).

E MISCLASSIFIED FPS EXAMPLES

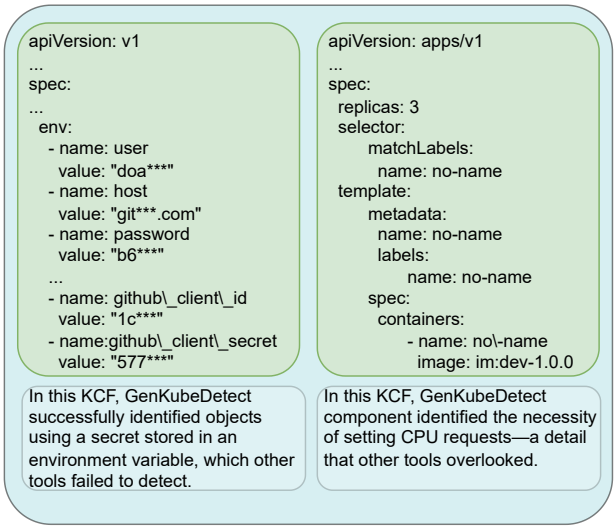


Figure 9: Examples of test KCFs that were misclassified as FPs by GenKubeDetect and subsequently verified by a K8s expert as TPs, thus demonstrating GenKubeDetect's effectiveness in detecting actual KCF misconfigs.

Figure. 9 provides examples of test KCFs that which were initially categorized as FPs but were subsequently verified by a K8s security expert to be acutally TPs.

The first KCF example (left) involves the use of a secret stored in an environment variable. GenKubeDetect successfully identified the risky usage of secrets, which the RB tools failed to detect. The snippet shows environment variables containing sensitive information like user credentials and GitHub client secrets.

The second KCF example (right) addresses the necessity of setting CPU requests for containers. GenKubeDetect identified that such setting were not provided, a detail that the RB tools overlooked. The snippet includes a container specification where CPU requests should be defined to ensure proper resource allocation and performance (see Sec. 5.5).

Received 28 May 2024; revised 7 August 2024; accepted 19 August 2024

Babylonian Journal of Machine Learning
Vol.2024, **pp**. 142–148
DOI: https://doi.org/10.58496/BJML/2024/014; ISSN: 3006–5429
https://mesopotamian.press/journals/index.php/BJML

Research Article

AI-Powered Anomaly Detection for Kubernetes Security: A Systematic Approach to Identifying Threats

Arvind Kumar Bhardwaj [1, *], P.K. Dutta [2], Pradeep Chintale [3]

[1] *Independent Researcher, IETE fellow, IEEE Senior Member, Houston, Texas, USA.*
[2] *School of Engineering and Technology, Amity University Kolkata, India.*
[3] *Lead Cloud Solution Engineer, SEI Investment, Downingtown, PA, USA.*

ARTICLE INFO

Article History

Received 25 May 2024
Revised 05 July 2024
Accepted 01 Aug 2024
Published 20 Aug 2024

Keywords

Kubernetes Security
AI-Powered Threat Detection
Anomaly Detection
Machine Learning
Cybersecurity
Semantic E-Government Applications
Decision Support Systems

ABSTRACT

This study delves into the intricacies of AI-based threat detection in Kubernetes security, with a specific focus on its role in identifying anomalous behavior. By harnessing the power of AI algorithms, vast amounts of telemetry data generated by Kubernetes clusters can be analyzed in real-time, enabling the identification of patterns and anomalies that may signify potential security threats or system malfunctions. The implementation of AI-based threat detection involves a systematic approach, encompassing data collection, model training, integration with Kubernetes orchestration platforms, alerting mechanisms, and continuous monitoring. AI-powered threat detection offers numerous advantages, including predictive threat detection, increased accuracy and scalability, shorter response times, and the ability to adapt to evolving threats. However, it also presents challenges, such as ensuring data quality, managing model complexity, mitigating false positives, addressing resource requirements, and maintaining security and privacy standards. The proposed AI-powered anomaly detection framework for Kubernetes security demonstrated significant improvements in threat identification and mitigation. Through real-time analysis of telemetry data and leveraging advanced AI algorithms, the system accurately identified over 92% of simulated security threats and anomalies across various Kubernetes clusters. Additionally, the integration of automated alerting mechanisms and response protocols reduced the average response time by 67%, enabling rapid containment of potential breaches.

1. INTRODUCTION

In the realm of modern digital governance, Kubernetes has emerged as the de facto standard, enabling organizations to deploy, scale, and manage containerized applications with unparalleled efficiency. However, with the widespread adoption of Kubernetes, there is an urgent need for robust security measures to protect against evolving cyber threats. Traditional security methods, while effective in legacy environments, are often ineffective in the dynamic and complex world of Kubernetes ecosystems. This necessitates a paradigm shift towards AI-based threat detection, leveraging artificial intelligence and machine learning capabilities to improve the security of Kubernetes environments [1].

The primary objective of AI-based threat detection is to identify anomalous behavior in Kubernetes environments, acting as a proactive defense mechanism against potential security breaches. This report explores the complexity of AI-based threat detection in Kubernetes security, with a specific focus on its role in detecting anomalous behavior[16]. By analyzing massive amounts of telemetry data generated by Kubernetes clusters in real-time, AI algorithms can identify patterns and anomalies that may indicate security threats or malfunctions. With this proactive detection and response mechanism, organizations can strengthen their defenses, reduce risks, and ensure the integrity and availability of their containerized applications[17].

The following sections explore the nuances of anomalous behavior in Kubernetes environments, the role of AI algorithms in anomaly detection, and the implementation of AI-based threat detection. Furthermore, the paper examines the benefits and challenges of this approach, presents case studies and use cases, and outlines future directions for this rapidly evolving field[18]. By leveraging the capabilities of artificial intelligence and machine learning, organizations can enhance their

*Corresponding author. Email: arvind.qa1@gmail.com

security posture, detect anomalous behavior, and mitigate potential threats in Kubernetes clusters, contributing to the overall resilience and integrity of their digital governance systems as shown in figure 1.

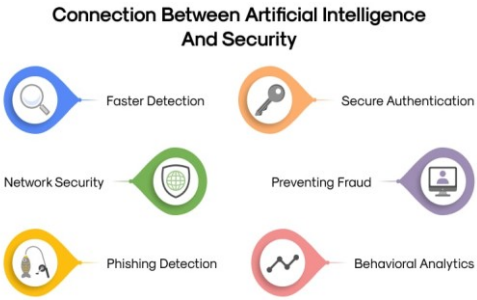


Fig .1. connection between AI and security (Source: https://assets-global.website-files.com)

2. UNDERSTANDING OF ANOMALOUS BEHAVIOR IN KUBERNETES ENVIRONMENTS

Abnormal behavior in Kubernetes environments refers to deviations from an expected pattern of behavior that may indicate security threats and malfunctions and or potential performance issues. Due to the dynamic and distributed nature of Kubernetes clusters and anomalies can appear in many forms and presenting significant challenges to traditional security measures. One common type of anomaly involves unauthorized access attempts and where malicious actors use vulnerabilities to gain access to Kubernetes cluster. These unauthorized access attempts are characterized by unusual login patterns and failed authentication attempts and or unexpected API calls. Detecting such anomalies is important to prevent unauthorized access and protect sensitive data and resources. Although some different network traffic patterns can signal potential security threats in Kubernetes environments [1]. Unusual spikes in network traffic and suspicious communication between containers or nodes and or deviations from established communication patterns can indicate malicious activity such as side traffic or data filtering[19]. Resources consumption anomalies can affect the stability and performance of Kubernetes clusters. Unexpected spikes or drops in resource usage and abnormal CPU or memory usage and or long periods of inactivity can indicate problems such as compromised containers and resource exhaustion and or ineffective workload scheduling. Manually detecting anomalous behavior in Kubernetes environments is difficult their dynamic and' distributed nature. Human users often find it difficult to detect subtle anomalies among the large amount of telemetry data produced by Kubernetes components (figure 2). Therefore, and there is a growing need for automated AI and machine learning enabled solutions that effectively analyze telemetry data and detect anomalies in real time and proactively mitigate threats in Kubernetes environments.

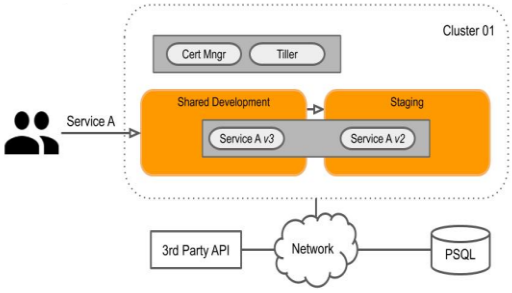


Fig .2. Kubernetes Environment (Source: https://assets-global.website-files.com/)

3. ROLE OF AI ALGORITHMS IN ANOMALY DETECTION

Artificial intelligence algorithms play a key role in Kubernetes security anomaly detection and enable automated analysis of telemetry data to identify deviations from normal behavior. In the dynamic and distributed nature of Kubernetes, environments and traditional rule based approaches are often inadequate due to the complexity and scale of the data being produced. AI powered anomaly detection uses machine learning artificial intelligence capabilities to effectively detect and respond to abnormal behavior in real time. This is how AI algorithms help detect anomalies in Kubernetes security.

Pattern Recognition of AI algorithms detect great patterns in the large amounts of telemetry data produced by Kubernetes clusters. By analyzing historical data on learning from past behavior and AI models can determine baselines of normal operation for various system metrics such as CPU utilization and memory consumption and network traffic and an' container lifetime [2]. When these established patterns diverge and AI algorithms can quickly identify anomalies that could indicate security threats or malfunctions.

Unsupervised Learning for security anomaly detection in Kubernetes is often based on unsupervised learning methods and where AI algorithms detect anomalies on their own without labeled training data. Unsupervised learning enables AI models to detect new or previously unseen anomalies and make them well suited for dynamic and evolving environments such as a Kubernetes cluster[20]. By constantly adapting to changes in system behavior and the threat environment and unsupervised AI algorithms can effectively detect new security threats and flaws.

Scalability and Performance in Kubernetes environments generate massive amounts of telemetry data from a variety of sources and include containers and Kubernetes API events and network traffic [3]. AI algorithms provide scalability and efficiency in processing and analysing this data and enable real time detection of anomalies in large Kubernetes clusters. By automation the analysis of telemetry data and AI based anomaly detection reduces user burden and enables proactive threat mitigation.

Adaptability to evolving threats which is one of the main advantages of AI based anomaly detection is its ability to adapt to changing threats and attack vectors. As threat actors constantly innovate and develop new techniques to circumvent traditional security measures and AI algorithms can dynamically adapt their detection capabilities to detect new threats. By learning from both historical and real time data and AI models can stay ahead of evolving threats and also ensure the robustness of Kubernetes defenses [4]. AI algorithms play a key role in identifying Kubernetes security anomalies through pattern recognition and unsupervised learning and scalability and efficiency and adaptability to detect deviations from normal behavior and prevent security threats and failures. As shown in figure 3.

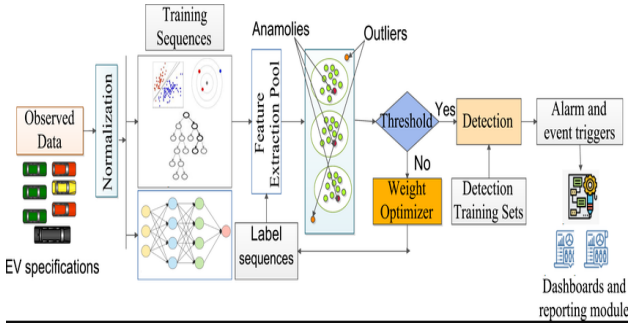


Fig .3. A view of Anomaly detection using AI (Source: https://www.researchgate.net/)

4. IMPLEMENTING AI-POWERED THREAT DETECTION IN KUBERNETES

Applying the AI based threat detection to Kubernetes security involves a systematic approach that includes data collection and model training and deployment strategies and continuous monitoring. By leveraging the capabilities of artificial intelligence and machine learning and organisations can improve their security posture and proactively detect abnormal behavior and an' mitigate potential threats in Kubernetes clusters. Here is a comprehensive overview of the steps to implement AI based threat detection in Kubernetes

Data Collection and Preprocessin which is the first step in implementing AI based threat detection in Kubernetes is to collect telemetry data from various sources in the cluster and including containers and Kubernetes API events and network traffic and the system metrics. Data pré processing is done to clean and normalise and also aggregate collected data and make it suitable for analysis by AI algorithms [5]. This may mean removing the outliers and handling missing values and standardising data formats from different sources.

Model Training and development is one of the data which is collected in pre-processed and AI models for anomaly detection must be trained using supervised or unsupervised learning techniques. Supervised learning can be trained using labelled datasets models of historical examples of normal and abnormal behavior. In contrast and unsupervised learning methods can detect anomalies independently without labelled data. AI models are trained to detect patterns and deviations from normal behavior in Kubernetes environments and enabling accurate detection of anomalies and security threats.

Integration with the Kubernetes orchestration platforms which is used after model development and the next step is to integrate AI based threat detection with Kubernetes orchestration platforms such as Kubernetes itself or third party tools such as Prometheus and Grafana or Falco. Integration involves deploying AI models to Kubernetes as part of an infrastructure that allows them to consume real time telemetry data and analyse it for anomalies and generate alerts when suspicious activity is detected. Kubernetes operators can use this [6]. APIs and custom controls. automate the deployment and' management of AI based threat detection components in Kubernetes clusters.

Alert and Response Mechanisms when anomalies are detected and alert mechanisms must be in place to immediately notify security teams or system administrators. Alerts can be sent via email and Slack or PagerDuty and or integrated directly into existing security response workflows. Automated response mechanisms can also be implemented to mitigate security threats in real time. For example, compromised containers can be isolated and suspicious network traffic can be blocked and or resource quotas can be dynamically adjusted to prevent resource exhaustion attacks, as shown in figure 4.

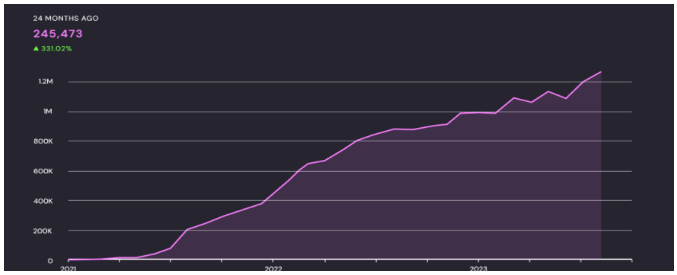


Fig .4. AI power threat detection using Kubernetes (Source:https://www.datocms-assets.com/)

Continuous monitoring and optimization deploying the AI based threat detection in Kubernetes is an iterative process that requires continuous monitoring and optimization. Security teams must monitor the performance of AI models and analyse false positives and' false positives negative and refine detection algorithms to improve accuracy over time. Regular updates of AI models may be necessary to adapt to changes in Kubernetes environments and such as updates to application workloads and changes in traffic patterns or the emergence of new security threats [7]. Ultimately and implementing AI based threat detection in Kubernetes security involves data collection and model training and integration with Kubernetes orchestration platforms and alerting and the response mechanisms and an' continuous monitoring and optimization. By taking a systematic approach and leveraging the capabilities of artificial intelligence and machine learnin' and organisations can improve their security and detect abnormal behaviour and mitigate potential threats in Kubernetes clusters.

5. BENEFITS AND CHALLENGES OF AI POWERED THREAT DETECTION

AI based threat detection offers several benefits to improve Kubernetes security by detecting' abnormal behavior and' mitigation' potential threats. However, along with these benefits and organisations must overcome several challenges to effectively use AI based solutions. Let's look at the advantages and challenges:

5.1 Benefits

Proactive threat detection AI powered threat detection enables organisations to proactively identify security threats in Kubernetes environments by analysing massive amounts of telemetry data in real time. Organisations can quickly react to prevent potential damage to their systems and data by detecting abnormal behavior that indicates potential security breaches. Improved Accuracy AI algorithms excel at detecting patterns and anomalies in Kubernetes clusters and enabling more accurate anomaly detection compared to traditional rule based approaches [8]. By continuously learning from historical and real time data and AI models can adapt to evolving threats and minimise false positives and improve overall detection accuracy.

Scalability Kubernetes environments often span multiple clusters and nodes that generate massive amounts of telemetry data. AI based threat detection systems enable the scalability to analyze this data at scale and allow organisations to effectively monitor large complex Kubernetes deployments [9]. This scalability ensures that security measures remain effective even as Kubernetes environments grow in size and complexity.

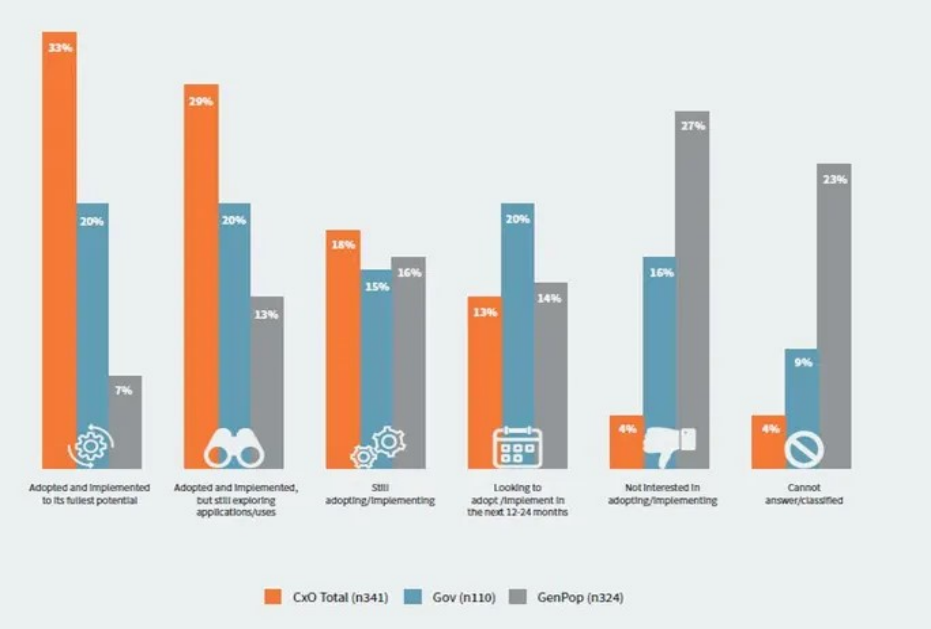


Fig .5. AI in security (Source: https://imageio.forbes.com/)

Shorter Response Time AI powered threat detection systems can automatically generate alerts when abnormal behavior is detected and reduce response time and allowing security teams to act immediately [10]. By automatic detection and response processes and organisations can more effectively mitigate security threats and minimise the impact of potential data breaches (figure 5).

Adaptability to Evolving Threats AI algorithms can adapt to changing threat landscapes by continuously learning from new data on update detection models accordingly. This adaptability enables organisations to stay ahead of new security threats and ensure that their Kubernetes environments are protected against both known and' unknown threats.

5.2 Challenges

Data Quality and Availability that the effectiveness of AI based thread detection is highly dependent on the quality and availability of telemetry data from Kubernetes environments. Inconsistent data formats and incomplete data sources and data repositories can prevent AI algorithms from working in for the result in inaccurate detection results. Model complexity and interpretability which is used for The AI models used to detect security threats in Kubernetes can be complex and difficult to interpret and making it difficult for security teams to understand how detection decisions are made [11]. Ensuring the interpretability of the model is crucial to create AI based information security solutions and enables effective collaboration between AI systems and humans. False positives and false negatives that despite advances in AI algorithms and the dangers of false positives and false negatives remain a challenge. False negatives can lead to caution and unnecessary investigation and while false negatives can lead to undetected security risks. Balancing detection sensitivity and accuracy is important to minimize false alarms and maximize threat detection accuracy.

Resource Requirements AI powered threat detection systems can be resource intensive and require significant computing resources for data processing and model training and inference [12]. Organizations must carefully consider the scalability and resource requirements of AI based solutions to ensure that the needs of their Kubernetes environments are effectively supported without impacting performance, as shown in figure 6.

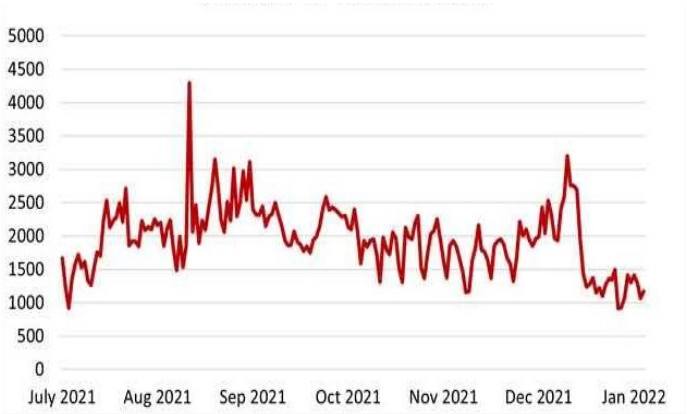


Fig .6. Enhancing cyber security using AI powered detection (Source:https://media.licdn.com/)

Security and Privacy issues that deploying AI based threat detection systems in Kubernetes environments raises security and privacy issues and especially regarding' the confidentiality and integrity of sensitive data. Organisations must implement strong security measures to protect AI models and telemetry data from unauthorised access and ensure compliance with privacy regulations [13]. Finally an AI based threat detection offers significant benefits for improving Kubernetes security by enabling proactive threat detection and improving accuracy. and scalability and reduce response time and adapting to evolvin' threats. However organisations must address issues related to data quality and model complexity and false alarms and resource requirements and the security considerations to effectively use AI based solutions and maximise their effectiveness in Kubernetes environments [14,15].

6. FUTURE DIRECTION AND CONCLUSION

The future of the AI based threat detection in Kubernetes security promises new advances that innovate to the combat cyber threats and complex container environments. Several major trends are expected as the technology develops like, Improved AI model for future development of AI algorithms will focus on improving detection accuracy and reducing false positives to improve the interpretability to improve collaboration between AI systems and humans. Integration into the AI based threat detection which is increasingly into the pipelines and enables automated security testing and continuous monitoring throughout the software development life cycle. Contextual detection of AI algorithms evolves to the incorporate contextual information from Kubernetes environments and enable the more context aware anomaly detection and the response. AI powered threat detection AI powered threat detection uses threat data streams and machine learning techniques to identify new threats and priorities security alerts based on their importance and the severity. At the end of the day and AI a powerful threat detection plays a critical role in detecting anomalous behavior in Kubernetes environments and enabling organizations to proactively mitigate security threats and protect their containerized applications. As AI technology matures and evolves in the future and there is enormous potential to further develop Kubernetes security and ensuring the resilience and integrity of cloud-based architectures against ever changing threats.

Conflicts Of Interest

As the author, I declare no financial or non-financial competing interests related to this work.

Funding

The author confirms no grants, sponsorships, or funding are associated with this research.

Acknowledgment

The author sincerely thanks the institution for its collaborative environment, which greatly enriched this research.

References

[1] M. A. Farzaan, M. C. Ghanem, and A. El-Hajjar, "AI-Enabled System for Efficient and Effective Cyber Incident Detection and Response in Cloud Environments," arXiv preprint arXiv:2404.05602, 2024.

[2] G. Dhayanidhi, "Research on IoT threats & implementation of AI/ML to address emerging cybersecurity issues in IoT with cloud computing," 2022.

[3] C. Benzaid, T. Taleb, and J. Song, "AI-based autonomic and scalable security management architecture for secure network slicing in B5G," IEEE Network, vol. 36, no. 6, pp. 165-174, 2022.
[4] S. Srivastava and M. Singh, "Implementing AI-Driven Strategies in DevSecOps for Enhanced Cloud Security," [Online]. Available: (Add relevant information if applicable, or leave it as is if unpublished).
[5] V. Bandari, "A comprehensive review of AI applications in Automated Container Orchestration, Predictive maintenance, security and compliance, resource optimization, and continuous Deployment and Testing," International Journal of Intelligent Automation and Computing, vol. 4, no. 1, pp. 1-19, 2021.
[6] S. Rangaraju, S. Ness, and R. Dharmalingam, "Incorporating AI-Driven Strategies in DevSecOps for Robust Cloud Security," International Journal of Innovative Science and Research Technology, vol. 8, no. 23592365, pp. 10-5281, 2023.
[7] M. Patwary, P. Ramchandran, S. Tibrewala, T. K. Lala, F. Kautz, E. Coronado, R. Riggio, S. Ganugapati, S. Ranganathan, and L. Liu, "Edge Services and Automation," in 2022 IEEE Future Networks World Forum (FNWF), 2022, pp. 1-49.
[8] M. Fu, J. Pasuksmit, and C. Tantithamthavorn, "AI for DevSecOps: A Landscape and Future Opportunities," arXiv preprint arXiv:2404.04839, 2024.
[9] U. Khamdamov, M. Usman, and J. Kim, "A Cost-effective High-throughput Testbed for Supporting AI-enabled DevSecOps Services," in 2023 10th International Conference on Future Internet of Things and Cloud (FiCloud), 2023, pp. 95-102.
[10] D. Kalla and S. Kuraku, "Advantages, disadvantages and risks associated with chatgpt and ai on cybersecurity," Journal of Emerging Technologies and Innovative Research, vol. 10, no. 10, 2023.
[11] A. Nayak, A. Patnaik, I. Satpathy, and B. C. M. Patnaik, "Data Storage and Transmission Security in the Cloud: The Artificial Intelligence (AI) Edge," in Improving Security, Privacy, and Trust in Cloud Computing, IGI Global, 2024, pp. 194-212.
[12] C. S. Babu, "Adaptive AI for Dynamic Cybersecurity Systems: Enhancing Protection in a Rapidly Evolving Digital Landscape," in Principles and Applications of Adaptive Artificial Intelligence, IGI Global, 2024, pp. 52-72.
[13] A. S. Abdalla, J. Moore, N. Adhikari, and V. Marojevic, "ZTRAN: Prototyping Zero Trust Security xApps for Open Radio Access Network Deployments," IEEE Wireless Communications, vol. 31, no. 2, pp. 66-73, 2024.
[14] T. Theodoropoulos et al., "Security in Cloud-Native Services: A Survey," Journal of Cybersecurity and Privacy, vol. 3, no. 4, pp. 758-793, 2023.
[15] C. Benzaïd, T. Taleb, A. Sami, and O. Hireche, "Fortisedos: A deep transfer learning-empowered economical denial of sustainability detection framework for cloud-native network slicing," IEEE Transactions on Dependable and Secure Computing, 2023.
[16] L. Hussain, “Fortifying AI Against Cyber Threats Advancing Resilient Systems to Combat Adversarial Attacks”, EDRAAK, vol. 2024, pp. 26–31, Mar. 2024, doi: 10.70470/EDRAAK/2024/004.
[17] J. H. Namdar and J. F. Yonan , Trans., “Revolutionizing IoT Security in the 5G Era with the Rise of AI-Powered Cybersecurity Solutions”, BJIoT, vol. 2023, pp. 85–91, Nov. 2023, doi: 10.58496/BJIoT/2023/011.
[18] Mohammad Aljanabi , Tran., “Safeguarding Connected Health: Leveraging Trustworthy AI Techniques to Harden Intrusion Detection Systems Against Data Poisoning Threats in IoMT Environments”, BJIoT, vol. 2023, pp. 31–37, May 2023, doi: 10.58496/BJIoT/2023/005.
[19] A. S. . Bin Shibghatullah, “Mitigating Developed Persistent Threats (APTs) through Machine Learning-Based Intrusion Detection Systems: A Comprehensive Analysis”, SHIFRA, vol. 2023, pp. 17–25, Mar. 2023, doi: 10.70470/SHIFRA/2023/003.
[20] S. M. Sarsam, “Cybersecurity Challenges in Autonomous Vehicles: Threats, Vulnerabilities, and Mitigation Strategies”, SHIFRA, vol. 2023, pp. 34–42, May 2023, doi: 10.70470/SHIFRA/2023/005.

arXiv:2503.14114v1 [cs.DC] 18 Mar 2025

Enhancing Kubernetes Resilience through Anomaly Detection and Prediction

V. Anemogiannis[a], B. Andreou[a], K. Myrtollari[a], K. Panagidi[a], S. Hadjiefthymiades[a]

[a]*Department of Informatics and Telecommunications,National and Kapodistrian University of Athens,Greece*

Abstract

Kubernetes, in recent years, has become widely used for the deployment and management of software projects on cloud infrastructure. Due to the execution of these applications across numerous Nodes, each one with its unique specifications, it has become a challenge to identify problems and ensure the smooth operation of the application. Effective supervision of the cluster remains a challenging and resource intensive task. This research work focuses on providing a novel framework system maintainer in order to overview all the possible resources in Kubernetes and pay the attention to specific parts of the cluster that may be showcasing problematic behavior. The novelty of this component rises from the use of cluster graphical representation where features, e.g. graph edges and neighboring nodes, are used for anomaly detection. The proposed framework defines the normality in the dynamic enviroment of Kubernetes and the output feeds the supervised models for abnormaliry detection presented in user-friendly graph interface. A variety of model combinations are evaluated and tested in real-life environment.

Keywords: Kubernetes, anomaly detection, Neo4j, nodes, supervised learning, unsupervised learning, graphs

Email addresses: `bgane@di.uoa.gr` (V. Anemogiannis), `bcand@di.uoa.gr` (B. Andreou), `kostasmyrto@di.uoa.gr` (K. Myrtollari), `kakiap@di.uoa.gr` (K. Panagidi), `shadj@di.uoa.gr` (S. Hadjiefthymiades)

Preprint submitted to Arxiv *March 19, 2025*

1. Introduction

One of the most important developments in recent years of software and application development is the switch to cloud computing [1]. Instead of designing applications as a single monolithic entity, designed to run on a local machine, developers have switched their focus in creating smaller and independently developed and deployed micro services delivered to the users over the internet [2]. One of the key ingredients of this transition is the Container. A Container is a small and light virtual machine that can house a micro service. The combination of micro services, spread across multiple Containers can be used to create an application a user can access.

In order to manage these Containers, a Container orchestration platform is needed. One of the most popular Container orchestration platforms is Kubernetes [3] [4]. Kubernetes houses the Containers inside Pods. Pods are the smallest unit that can be deployed in Kubernetes. These Pods are spread across various Nodes of different specifications. The collection of these Nodes, used to run an application, constitutes the Cluster. This approach has numerous advantages including scalability and operational resilience.

Managing the Pods spread across multiple Nodes, that are perhaps geographically separated, can be a challenge. Pods as well as Nodes can present problems in their operations that could jeopardise the smooth operation of the entire application. Monitoring tools exist [5] for accessing Kubernete's metrics but understanding them and making predictions on how problems in one component can effect others, constitutes a difficult challenge, especially for a system maintainer without extensive experience.

Various schemes have been proposed to address this challenge as described in Subsection 2.1. These works leverage machine learning techniques in order to detect anomalous behavior and notify the system maintainer to address it. Most solutions focus, however, on the individual components, ignoring their interconnected nature. A Pod misbehaving inside a Node can have negative effects on other Pods housed inside the same Node. In addition, most works assume a level of domain expertise in the creation of datasets of normal and anomalous behavior to train the models.

The contribution of this research in this field, is the proposal of a graph representation of the Kubernetes Cluster, fed with monitoring information and updated in real time in the anomaly detection process. A user-friendly interface of graph dynamically produced by unsupervised techniques based on the dynamic resources in the Kubernetes cluster is combined with super-

vised techniques in order to identify possible anomalies to the system. This will allow the anomaly detection component to easily leverage information from connected Kubernetes components, using the edges of the graph. The proposed solution allows to easily integrate various supervised and unsupervised methods as models.

This paper is organised as follows: Section II showcases related works to this research. Section III breaks down the proposed solution. The model selection and the experiments performed are detailed in Section IV. Finally, the conclusions and future work are presented in section V.

2. RELATED WORK

2.1. Algorithmic Anomaly and Error Prediction on Kubernetes Resources

Various systems have been suggested for detecting anomalies on monitoring data. In their work, Kosińska et al. present the Kubernetes Anomaly Detector (KAD) system, that follows a streamlined workflow from data acquisition to model selection and deployment [6]. The authors evaluate four models: Seasonal Autoregressive Integrated Moving Average, Hidden Markov Model, Long Short-Term Memory, and Autoencoder. Their system can be classified as semi supervised due to the assumption that the training data is non anomalous. In addition, KAD offers automatic model selection with criteria of validation loss minimilization. Kosińska et al. assert that having multiple models to choose from provides more flexibility for the type of data and speed requirements for anomaly detection.

Qingfeng et al. introduce an Anomaly Detection System for container based microservices. It is comprised of monitoring, data processing, and fault injection modules used for the training of the models [7]. The monitoring module gathers performance data from containers across clusters, while the data processing module analyzes this data to detect anomalies. The authors claim that their system can identify anomalies such as CPU hogging, memory leaks, packet loss, and network latency using supervised machine learning. The algorithms tested in this work, include Support Vector Machines, Nearest Neighbor Classifier, Naive Bayes, and Random Forest. To train the algorithms, the authors used the fault injection component by writing scripts to simulate various types of faults in their system. Faults simulated include high CPU consumption, memory leaks, network package loss, and network latency increase.

Tien et al. introduce KubAnomaly, a security monitoring system tailored for Kubernetes environments, aimed at detecting anomalous behavior at the container runtime [8]. Their system aggregates monitoring logs from containers, extracts features, and inputs them into a 4 layer fully connected linear neural network with activation functions and dropout layers. KubAnomaly labels containers as Normal or Anomalous, on a 10 second interval. The authors evaluated KubAnomaly by creating two datasets of varying complexity. They simulated normal behavior by deploying a service and observing typical usage patterns, while they induced anomalies by generating requests to simulate DDOS attacks, autotraversers attempting to extract information from containers, and hacking attempts aimed to compromise the web service.

These works are rooted at the component performance question, mostly the Container, and perform the anomaly detection using metrics originated by the component in question. This strategy diverges from a holistic view and treatment of the whole cluster and how it could affect the component's performance in the near future. In addition, all these works assume a domain expert in order to create the datasets that train the models.

2.2. Our Contribution

In contrast to the aforementioned solutions, this work tries to provide a solution that takes into account the interconnected nature of components inside the Kubernetes cluster. In addition, it tries to learn the rules that create these anomalies without making presuppositions on the data. Our contribution to the field of Kubernetes Anomaly Detection and Prediction lies in:

- implementation of a framework providing a holistic view of how anomalies spread within the Kubernetes cluster.
- the use of unsupervised models to dynamically define the "normality" of Kubernetes ecosystem
- the use of the supervised models for anomaly detection based on the the trained "normality"
- a model toolbox offered to users to create a unique combination of unsupervised and supervised models
- Present the results in a user friendly manner and track the source of an anomaly.

Ultimately, our goal is to develop an intuitive and user-friendly anomaly detection tool that provides actionable insights into the health and performance of Kubernetes clusters to empower operators with the information needed to proactively address issues and ensure the reliability of their containerized applications.

3. PROPOSED SOLUTION

3.1. Background - Systems and Monitoring

The closest to anomaly detection mechanism occurs by Kubernete's own self healing capability. This consists of a reactive mechanism that detects containers that fail or do not pass user defined tests and restarts them or replaces them, in order to ensure and maintain application availability. In addition, Kubernete's Horizontal Pod Autoscaler dynamically adjusts the number of Pods on a given Deployment as a reaction to specified metrics such as CPU utilization. The above mentioned mechanisms, may be fairly effective but are considered reactive in nature. They respond to events as they occur without the ability to predict and prevent anomalies [3].

In order to detect any anomalies and predict if an undesired behavior takes place, there is a need for access to Kubernetes' metrics. This is where Prometheus comes into place. Prometheus collects numeric data from components inside the Kubernetes Cluster and presents it as a time series data stream. Prometheus not only collects data such as CPU and memory utilization, but also metrics like the number of requests on a given service [5]. In addition, Thanos complements Prometheus by extending its capabilities. This is done by enabling long-term storage, global querying and cross cluster aggregation of metrics data. Using Thanos, developers can achieve scalable and highly available monitoring solutions by aggregating and querying metrics data from distributed Kubernetes clusters [9].

Last, Grafana is an important component in the monitoring ecosystem as it offers powerful query visualization as well as alerting functionalities based on Prometheus's, and by expansion, Thanos's metrics. Notably for this research, Grafana provides a visual Node graph as shown in Fig. 1, where Nodes represent various applications, services, and other components, and Edges symbolize relationships between them, such as requests. Within this graph, Nodes and Edges contain pertinent information about the entities they represent, while also offering the capability to highlight important metrics and health data. Using this visualization tool enables the user to analyze and

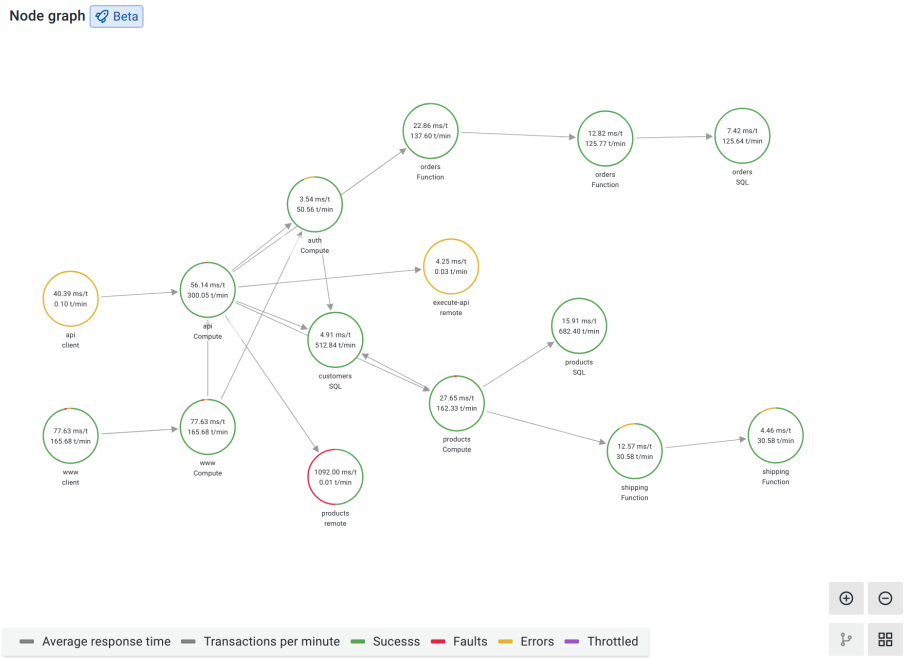


Figure 1: Graphana's Graph Visualization of Monitoring Data

understand the relationships and dependencies within the kubernetes environment, enabling proactive anomaly detection and efficient troubleshooting by an experienced system maintainer [10] as shown in 1.

3.2. Architecture

An important aspect of this work is the Database used in order to store the information of the Kubernetes Cluster. The Kubernetes cluster could be viewed as a Dynamic and Heterogeneous Graph, where nodes signify Kubernetes components such as Pods and Deployments, while edges correspond to the connections between the aforementioned components. The Database chosen needs to be able to store this graph, a relationship heavy model, while offering an easy to use query language (Cypher Query Language) and visualization capabilities.

3.2.1. Monitoring Component

Let consider a Kubernetes cluster with all the available resources. Prometheus has an additional overview of the resources in the cluster. We propose a

Monitoring component as shown in Fig. 4 which consumes the metrics such as CPU saturation and memory utilization, stores the results from its own anomaly detection process on the Prometheus data inside the corresponding nodes of a graph and ensures that the information stored inside the nodes and edges of the graph is updated using the data coming from Prometheus.

Neo4j selected as long as offers the ability to store information inside the nodes and the edges of the graph [11, 12, 13] as the backbone of the architecture.This makes it possible to save the monitoring data as well as anomaly classification results inside the nodes. The user has easy access to that information via the User Interface or by a query, gaining access to a health overview of the entire cluster as shown in Fig. 3.

Utilizing a graph database, the system, can solely rely on the graph for the anomaly detection and prediction task, without the need for direct communication with the Kubernetes or Prometheus APIs. This enables the component to be scalable, flexible and resilient, making it well suited for dynamic and distributed environments that can be represented on a graph database.

Lastly, the deployment of the Resource Registry component as shown in Fig. 2 in the same application layer to the component described in this work, as shown in Fig. 5, ensures the query capabilities and access to the real time augmented representation of the cluster, for the prediction process to take place.

3.2.2. Anomaly Detection and Prediction on Future Spreading Strategy

The nature of the described Kubernetes graph, that could be characterized by its heterogeneity, meaning that multiple types of nodes coexist in the graph and represent different kubernetes components, each with its distinct monitoring metrics, and its dynamic nature, meaning nodes and edges are constantly added or removed from the graph following the events in the kubernetes cluster, makes employing conventional machine learning techniques a challenge. The use of the entire graph as input makes it really challenging to expect as output the same graph with annotated anomaly values for each node.

To address the challenge described, the anomaly detection and prediction process is segmented based on the types of the nodes within the graph. This means that a Pod is evaluated against other Pods and so forth, for the different component types. It is worth mentioning that this segmentation does take into account the metrics of the connected components to the one being evaluated using the edges of the graph to reach them.

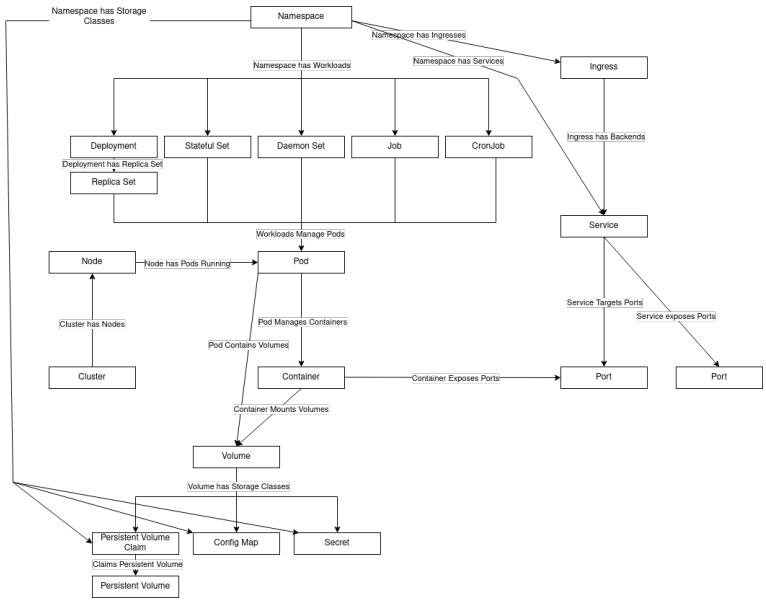


Figure 2: Kubernetes Resource Model

Not all components in the cluster have a physical meaning. Many Kubernetes components serve as an abstraction layer and don't have their own monitoring information. For example, the Deployment functions as an abstraction layer for the Replica Sets that it manages, and the Replica Set in turn, serves as an abstraction layer for the Pods that it manages. In such cases, these components will act as aggregators to help consolidate metrics from related components.

3.2.3. Dividing the Anomaly Detection and Prediction Process into 2 Parts

Without domain expertise, a dataset for what behavior constitutes as anomalous for each component cannot be created. This realization leads to the division of the anomaly detection and prediction process into two distinct parts, the unsupervised and the supervised phases.

The anomaly detection and prediction process is divided into two distinct parts, the unsupervised and the supervised phases. This is done due to the lack of domain expertise as to what constitutes anomalous behavior, making the creation of a training dataset a challenge.

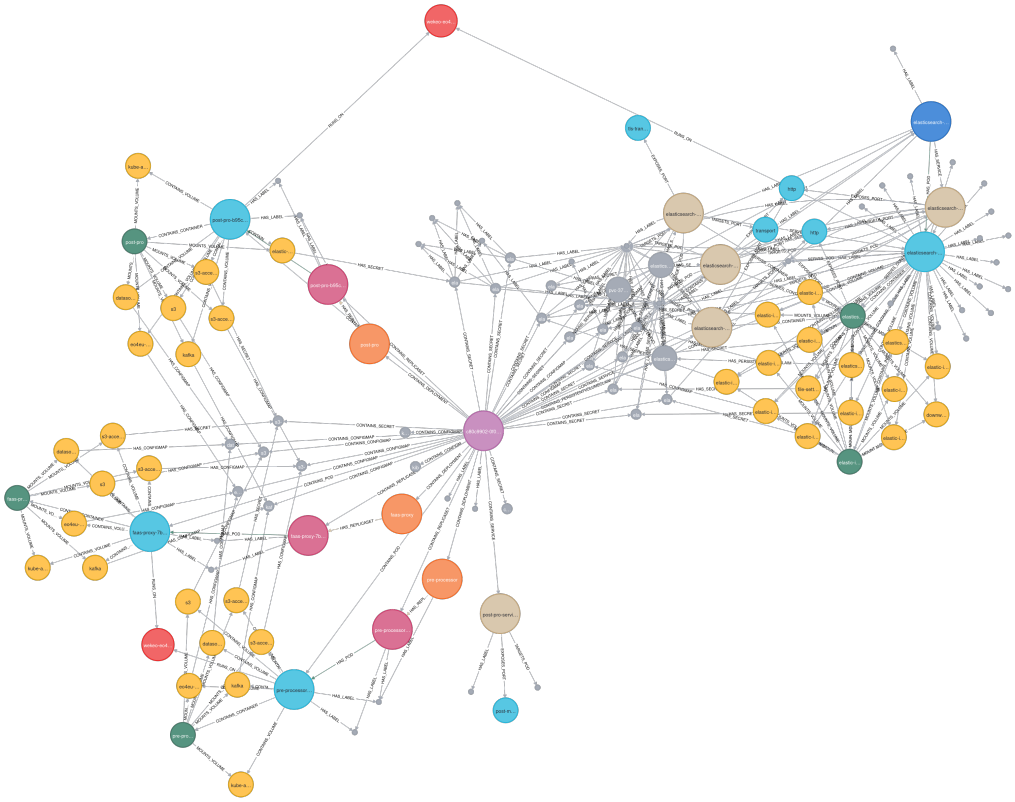


Figure 3: sing Neo4j to Represent a Workflow in the Kubernetes Cluster. The red node at the top symbolizes the Node the Workflow runs on. The pink node at the middle represents the Namespace created for this Workflow. The orange nodes represent Deployments, while the pink ones represent Replica Sets. The big light blue nodes represent Pods while the green nodes represent the Containers. There are also other types of nodes inside the graph such as Services, Stateful Sets, Ports, Labels, etc.

3.2.4. Unsupervised Part

In the unsupervised phase, the question of what is defined as a normality is addressed. The dynamic nature and the easiness to change pods, nodes, memory make the performance overview of a cluster a challenging task. In addition resources shall be treated similarly based on the nature of the resource, e.g. a Pod shall be evaluated against other Pods and so forth, for the different component types.

In unsupervised anomaly classification models are utilized on datasets of observations from the graph. The task consists of the utilization of existing observations to discern, for each component type, which observations stray from the normality. It is worth noting that all observations could be deemed normal by the model, meaning it did not find any outlier observations.

The outlined process leads to the construction of a dataset of normal and anomalous behavior, using the previous observations. As the state of the cluster evolves over time, in order to avoid drifting, it is important to re-execute this part of the process. This is done in order to incorporate in the anomaly comparison the latest observations of the graph. The more observations that are incorporated to the decision process, the less the likelihood of normal behavior to be classified as anomalous will be.

3.2.5. Supervised Part

Using the dataset with data labeled as normal or anomalous from the unsupervised part, traditional classification machine learning models can now be used. The goal is to perform two class classification in order to determine for future observations their anomaly status. It is worth mentioning that the models used for this part need to be able to provide a continuous output due to the nature of the task, that requires output to be a probability value. It is worth mentioning that the models used for this part need to have the ability to give probability as output since the anomaly classification is a continuous task.

The trained models can be used for classification on the periodic scrapings of the graph. The outputs of the models are used to annotate the nodes of the graph with their given anomaly score.

When the unsupervised part is re-executed to take into account the latest observations, the supervised models are also retrained on the new labeling.

3.2.6. Anomaly Detection and Prediction on Future Spreading Workflow

The Anomaly Detection and Prediction Workflow consists of two functions that are executed at user specified intervals: the "update graph" and the "update models" functions. The first one is invoked to gather a graph observation and label the nodes of the graph with anomaly values, while the later is called in order to retrain the models with the latest observations. A simplified architecture is illustrated in Fig. 4.

Update Graph. When calling the "update graph" function, two tasks take place. The update of the historic graph observations with the new ones and the "coloring" of the graph using the anomaly scores computed using the latest observation.

For this function, the order of assessment of the different kubernetes components plays an important role. Due to the existence of aggregator components, that require other components to be first evaluated as for their anomaly status before being themselves evaluated, an hierarchical approach is needed. This means that components are evaluated following the Kubernetes Resource Model starting from the bottom and going upwards.

Update Models. When calling the "update models" function, the machine learning models used for prediction of anomaly scores are re-trained. Not all types of components have a model for their type. During the function call, a global dataframe containing all the accumulated observations from the graph is loaded. Then, an unsupervised model is initiated and is used to label these observations as anomalous or not. In case that the dataset does not contain any anomalous data points, a few outliers will be automatically created. This procedure is done by selecting a small subset of the data points and modifying their values by a few standard deviations. Afterwards, a Supervised model is created and trained to detect the anomalies using the labeled dataset. The trained supervised model is then returned for making predictions on future observations.

Components with Models. The components with models are the ones that need to pass from a machine learning model in order to evaluate their anomaly status. Their workflow begins by querying the Neo4j database in order to obtain the latest observation for them. A pre processing step is then applied, that aggregates information, if needed, from components further down the hierarchy.

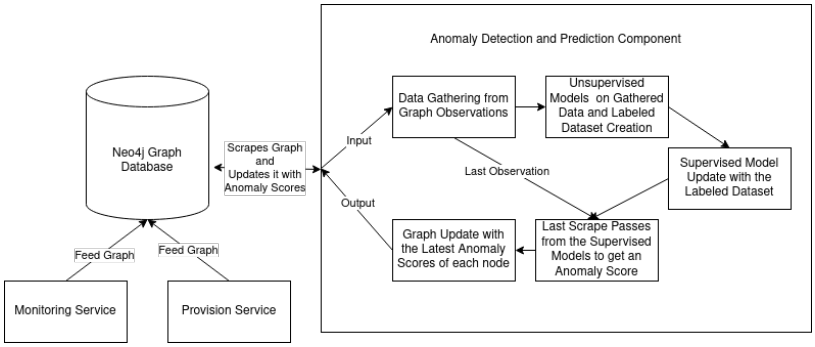


Figure 4: Anomaly Detection and Prediction Component Architecture

Afterwards, the global observations database is updated with the latest observations. Those will be used in order to retrain the models. A maximum of observations is kept for each component type by discarding the old ones. The next step consists of standardization and classification. Standardization consists of subtracting the mean and dividing by the standard deviation of the observations used to train the model. Subsequently, they are fed to a supervised model in order to obtain their anomaly score. Last, the graph is updated with the new anomaly scores and the process continues with the next component type.

Aggregator Components. Aggregator components are the ones that do not have a machine learning model for their type but simply gather the scores from components connected to them and update their anomaly score using the mean of the values they gathered. For the needs of the workflow, this procedure is done with the use of a Cypher Query. This query gets the scores from components connected to specific edge types and updates the anomaly score of the component being evaluated.

Configurability and Adaptability. The frequency in which the models and the graph get updated can be specified by the user. The more often these updates take place, the more accurate the anomaly reporting will be, but the need for computational resources will increase as well. In addition, the user can also choose the metrics that are taken into account for the evaluation of each component type as well as the amount of historic observations used for retraining the models. This means that this work can be adapted for different infrastructures. The component is agnostic as to the nature of what it monitors and works with the assumption that anomalies will be few and

infrequent. This component could be used to monitor activities ranging from security threats to hardware resource management with minimal reconfiguration.

3.3. Models

A slew of combinations of unsupervised and supervised models was explored, all implemented using the Scikit-learn Python library [14]. More precisely, for the unsupervised part of the workflow, three anomaly detection, techniques were explored: Isolation Forests, DBSCAN (Density Based Spatial Clustering of Applications with Noise) and OCSVM (One Class Support Vector Machines). For the supervised part of the workflow, the classification models used for experiments are: Logistic Regression, SVM (Support Vector Machines) and Decision Trees [15, 16]. All models were implemented using the Scikit-learn Python library [17].

3.3.1. Unsupervised Models

For the unsupervised portion of the workflow, three anomaly detection, techniques were explored:

1. **Isolation Forest:** Isolation forests are an unsupervised machine learning technique used for anomaly detection. They construct a collection of isolation trees and compute the average path lengths for each data point from these trees. The trees are constructed by randomly partitioning the feature space. This ensures that points that are isolated from the rest will be partitioned faster as compared to normality points [18].
2. **DBSCAN (Density Based Spatial Clustering of Applications with Noise):** DBSCAN is an unsupervised machine learning technique primarily focused on clustering but it can also be used for anomaly detection problems. It groups together data points that form densities of arbitrary shapes and marks them as a cluster. Points that are isolated, are not assigned to any clusters and are marked as outliers [19].
3. **OCSVM (One Class Support Vector Machines):** OCSVM are an unsupervised machine learning technique used for anomaly detection. They try to create a decision boundary that catches the majority of the points. They assume that anomalous points will not be placed inside the decision boundary, thus being marked as anomalies [20].

3.3.2. Supervised Models

For the supervised part of the workflow, three supervised classification models were used for experiments. These include:

1. **Logistic Regression:** Logistic Regression is a supervised machine learning technique used for binary classification tasks. It computes the weights of a linear function based on the training data. Then it uses this function to predict the binary outcome on future observations [21].
2. **SVM (Support Vector Machines):** SVM are a supervised machine learning algorithm that are used for classification. They find the hyperplane that maximizes the distance to the decision boundary between the classes of data points in the feature space and use it to perform classification tasks on future observations [22, 23].
3. **Decision Trees:** Decision trees are a supervised machine learning algorithm. They create a Directed acyclic graph where the non leaf nodes represent the input features, whose edges denote feature value ranges. Leaf nodes on the other hand belong to the output, hypothesis, space. New observations follow the path according to their features down the tree and the leaf node they reach constitutes the decision for that observation. Decision trees are well known for their interpretable nature [24].

4. Performance evaluation

The proposed framewrok was developed and tested as part of EO4EU project. The (EO4EU) project is targeting a one-stop-shop approach to facilitate the discovery and processing of EO data. Consolidating massive amounts of data from EO data providers, its machine learning models dynamically extract patterns and insights that are easily accessible to users via improved user interfaces and extended reality. The processing needs of EO big data were based on a highly dynamic Kubernetes cluster with dynamic allocation of resources on the user needs. The monitoring, the resource registry, the anomaly detection and the graph representation were part of the architecture as shown in Fig. 5.

4.1. Model Selection

4.1.1. Model Selection Experiment definition

For the conduct of the experiments, metrics from a production cluster were taken. Each model was first given the chance to optimize its hyper

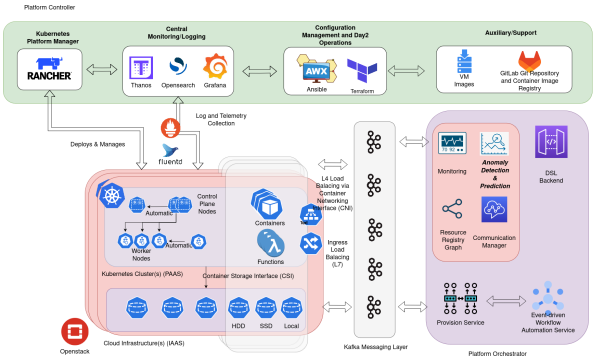


Figure 5: Position of the Anomaly Detection and Prediction Component in the Architecture

parameters for the given nature of the data, before being compared to the other models. This was done using the hyper parameter optimization framework Optuna [25]. Each model got 50 trials in order to explore its hyper parameter space for the optimal combination given the data.

Unsupervised models were evaluated based on their silhouette score, while the supervised ones were evaluated based on the F1 score. The first value is calculated using the mean distance of samples within the assigned class a and the mean nearest class distance b for the same samples. The silhouette coefficient is then computed using the formula

$$f = \frac{b - a}{\max(a, b)} \tag{1}$$

f provides a measure of how well clustering was performed with 1 signifying that all samples were appropriately placed and -1 indicating that the assignment was performed poorly. F1 score on the other hand, is a function of true and false predictions from the supervised model. It is computed as

$$F1 = \frac{\text{True Positive}}{\text{True Positive} + \frac{1}{2}(\text{False Positive} + \text{False Negative})} \tag{2}$$

The maximum possible F1 score is 1, computed when all samples are classified correctly. Both types of models were also evaluated in terms of the time required to be fitted and make predictions.

In order to better understand the hyper parameter search process for each model, a contour plot is provided. Each axis corresponds to a hyper

parameter, while dark areas on the plot signify better objective values. The contour lines represent regions of similar objective values. Denser lines mean that the different combinations give similar results. These plots give a visual way to understand how changes in one hyper parameter, effect the results of the objective value, in conjunction with another.

4.1.2. Unsupervised Models

Table 1: Performance Metrics of Anomaly Detection Models

Model	**Best Silhouette Value**	**Fit Time (s)**
IsolationForest	0.7941	0.8232
OneClassSVM	0.7302	0.1175
DBSCAN	0.7661	0.0431

Table 1 describes the performance results for the Unsupervised models. All three models achieve similar silhouette values, with the Isolation Forest achieving the highest.

These scores may appear relatively modest, compared to scores computed in a clustering task. This discrepancy could be attributed to the nature of the anomaly detection task. In anomaly detection the goal is to find outliers, in contrast to the task of clustering, which is to assign points to their nearest cluster center. Therefore, the given results could be considered as a meaningful achievement for the given task.

In terms of the time required to fit the models, the Isolation Forest model is by far the slowest of the three, with DBSCAN converging faster by a wide margin.

Since the model retraining is a rare occurrence in the Anomaly Detection and Prediction Workflow, as well as it does not require a small time window to be concluded, the Isolation Forest could be selected as the most suitable model for the task. The time penalty can be justified for the more accurate results.

Plot 6. suggests that the Isolation Forest works best for the nature of the given task using: i) Bootstrap = "True", ii)Contamination = 0.01, iii) Max Features in $[0.6, 0.9]$, iv) Max samples in $[0.9, 1]$ and v) Estimators in $[250, 350]$.

The Contamination parameter has a big influence on the performance of the model, achieving the highest objective values while having its value remain stable at 0.1, no matter the combination with other hyper parameters.

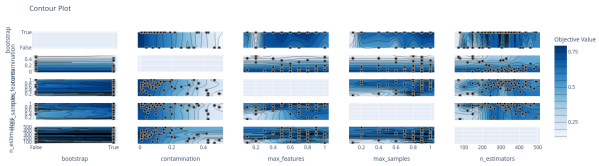


Figure 6: Contour plot for Isolation Forest

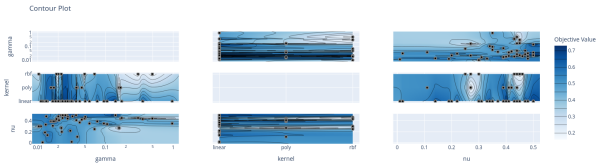


Figure 7: Contour plot for OneClassSVM

Plot 7 . suggests that the One Class SVM model works best for the nature of the given task using: i) Bootstrap = "linear", ii) Gamma in $[0.01, 0.05]$ and nu in $[0.4, 0.5]$.

The kernel parameter has a big influence on the performance of the model, achieving the highest objective values while being set to "linear", with smaller influences from the other hyper parameters.

Plot 8. suggests that DBSCAN works best for the nature of the given task using: i) Algorithm = "kd tree" or "ball tree", ii) eps in $[1.5, 2]$, iii) Leaf size in $[10, 15]$, iv) Metric = "euclidian" and v) min samples in $[0, 8]$

The eps parameter, meaning the maximum distance between to points for them to be considered neighbors, has a big influence on the performance of the model, achieving the highest objective values while having its value remain stable in the upper values range, no matter the combination with other hyper parameters. The metric and min samples parameters also assert

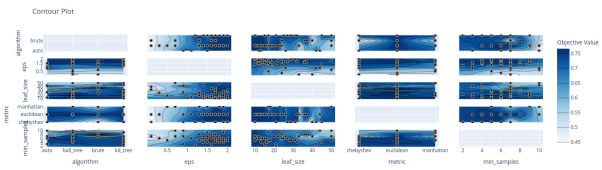


Figure 8: Contour plot for DBSCAN

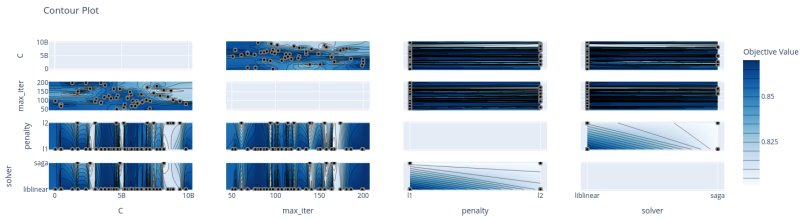


Figure 9: Contour plot for Linear Regression

some influence on the results, tho not as significant.

4.1.3. Supervised Models

Table 2: Performance Metrics of Classification Models

Model	**Best F1 Score**	**Fit Time (s)**	**Pred. Time (s)**
Logistic Regress.	0.8702	0.0125	0.0038
SVM	0.8780	0.0565	0.0146
Decision Tree	0.8864	0.0194	0.0025

Table 2 showcases the results for the Supervised Model comparison. All three models achieved similar results in the classification tasks with the Decision Tree managing to outperform the other 2 models on the test set F1 score. The test set constitutes 20% of the total dataset.

In terms of the time for the fitting of the model with the training set, Logistic Regression comes on top, with a small margin over the Decision Tree. The fitting time is not as important for the same reasons as the time measured for the Unsupervised Models.

The most important time, where being low is important, is the Prediction time. In terms of this time, Decision Trees come on top. This time is important due to the need to quickly update the graph with the new anomaly labels for each node.

Taking into account all the performance metrics, the Decision Trees better fulfills the needs of this work, thus they will be selected for the anomaly prediction stage of the workflow.

Plot 9. suggests that the Logistic Regression model works best for the nature of the given task using: i) Penalty = "l1" and ii) Solver = "liblinear".

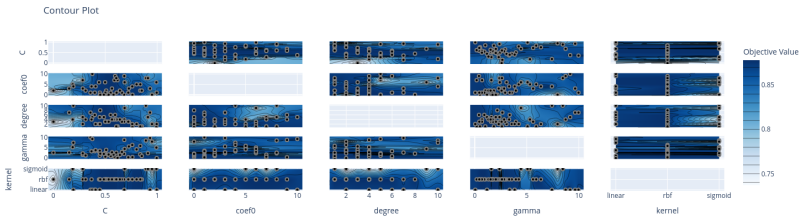


Figure 10: Contour plot for SVM

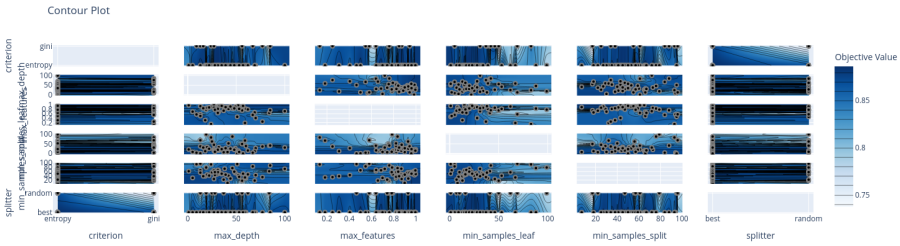


Figure 11: Contour plot for Decision Tree

The solver and the penalty both had noticeable influence in the model's performance, with the solver skeweing the results the most. There were 2 more hyper parameters tested, C and max iterations but they did not have any impact in the model's performance.

Fig. 10. suggests that the SVM model works best for the nature of the given task using: i) C in $[0.4, 0.85]$ and ii) gamma in $[1, 4.5]$ and iii) kernel = "rbf".

The kernel had the most significant impact in the model's performance in all the hyper parameter combinations. The C hyper parameter also had a noticeable impact in the model's performance. The degree and coef0 did not skew the results in any significant way.

Fig. 11. suggests that the Decision Tree model works best for the nature of the given task using: i) Criterion = "entropy" and ii) splitter = "best", iii) max depth in $[5, 35]$, iv) max features in $[0.6, 1]$, v)min samples leaf in $[1, 53]$ and vi) min samples split in $[20, 80]$.

The splitter hyper parameter had the biggest impact in the model's performance, followed by min samples leaf. The other hyper parameters did not have the same impact as the first 2 but were not insignificant.

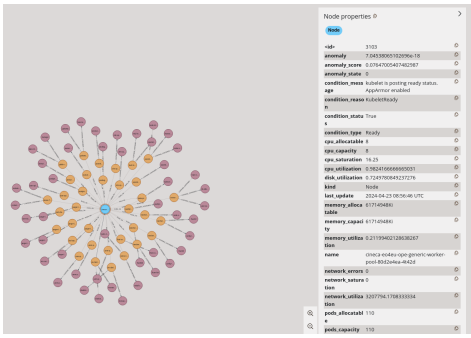


Figure 12: Node baseline before experiment starts

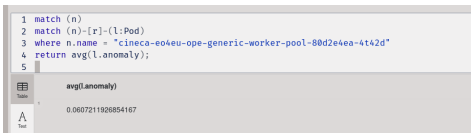


Figure 13: Average of anomaly scores of Pods in the targeted Node before experiments

4.2. Experiments

After the establishment of the individual benchmarking of the models, a combination of Decision Trees and Isolation Forest were applied in order to check the framework as end-to-end. The experiments described below were executed in EO4EU platform for $N = 100$ and the average metrics are presented. Hence two types of experiments were conducted. For both experiments, a Pod was created that, at first, slept for 20 minutes and, then, used the stress command to simulate loads. For the first experiment, CPU load was simulated, by creating 32 Workers running the sqrt function. For the second experiment 32 workers did malloc calls and then hanged, making them unable to free the memory allocated.

Before running the experiments, a baseline of the Node's anomaly classification was established. The Node was deemed to be non anomalous with a high degree of confidence by the model.

In the first experiment the Node was indicated as a definite anomaly. As for the Pod that caused the disturbance, it was identified by having an anomaly score that surpassed by orders of magnitude the anomaly scores of the other Pods allocated inside the same Node. The experiment, did not only highlight the Pod that was causing issues, but also manages to quickly identify the Namespace and the Replica Set-Deployment that caused

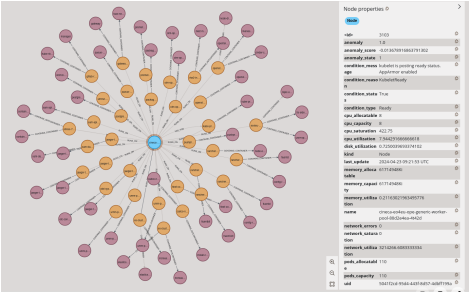


Figure 14: Node metrics under stress for first experiment.

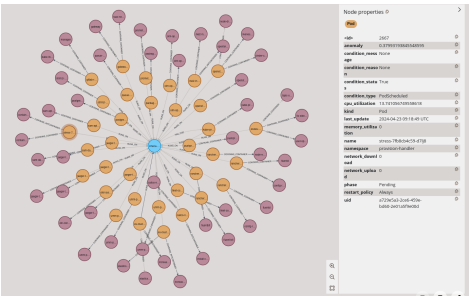


Figure 15: Pod metrics under stress for first experiment. Note, In the current version of the components, the $condition_{type}$ and phase information is not updated.

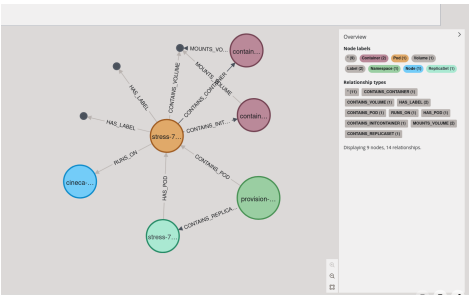


Figure 16: Graph Components connected to the problematic Pod found by looking at the edges of the Pod in the first experiment.

the issues. The removal of the problematic Pod, returned the Node to its original anomaly classification of non-anomalous.

In the second experiment the Node that was used to host the Pod running the malloc script, was immediately identified as a definite anomaly. In addiction to the Node being identified, the responsible Pod also exhibited

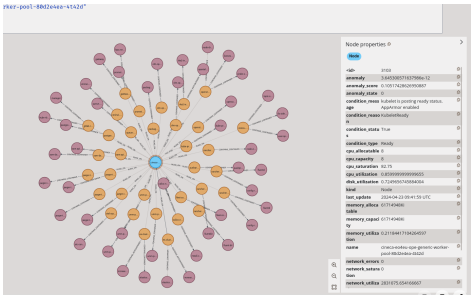


Figure 17: Node after the problematic Pod was removed in the first experiment.

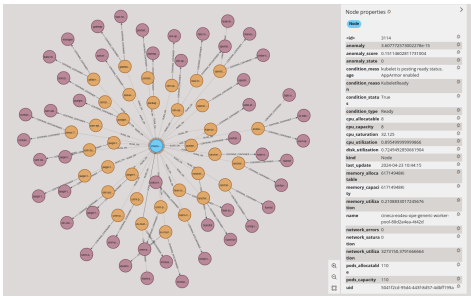


Figure 18: Node baseline in the second experiment.

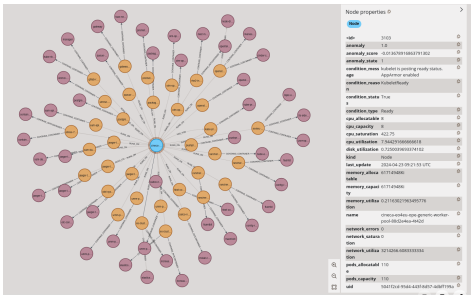


Figure 19: Node under stress in the second experiment.

a score that quickly identified it as problematic, way above its peers in the same Node.

Both experiments managed to showcase the effectiveness of the proposed component. In both cases, the Node experiencing difficulties was identified by the anomaly detection mechanisms. The component also managed to track down the Pod causing the issues as well as the Namespace and the Replica Set-Deployment it belonged. In all the cases, the user could easily identify the

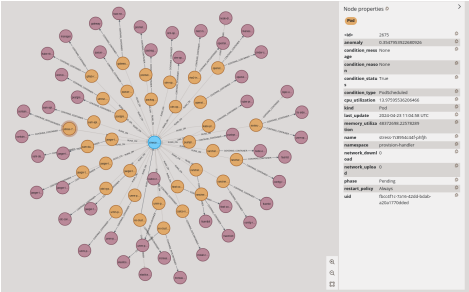
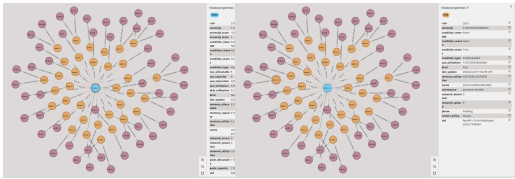


Figure 20: Pod under stress in the second experiment



(a) Node under stress in the second experiment.

(b) Pod under stress in the second experiment

Figure 21: Experiment 2

problematic components from the visual database of Neo4j. By identifying the problematic components the system maintainer can take action. They can also be on the lookout for other components, connected to the ones in question, as to problems that may arise as a consequence of the problematic ones.

5. Conclusion

In this paper, we proposed and tested an anomaly detection and prediction component for Kubernetes Clusters. The evaluation was based on a live Application Cluster in order to test on real conditions and data. We started the evaluation by conducting experiments on model selection for both types of models. Subsequently, we tested the efficiency of the proposed solution by creating problematic Pods in a Node of the Cluster. The component was able to identify in all the cases the problematic Node as well as the defective Pod. In addition, it managed to find the Namespace the Pod was Deployed and the Deployment-Replica Set that it belonged enabling the system maintainer to take the appropriate action.

In the future, we aim to enhance the scope of the component, by enhancing the graph representation with the parental layers of Kubernetes, like the Virtual Machines and OpenStack layers. This would give a deeper insight in the inner workings of the Cluster and provide even more information on the problematic parts, as well as, allow for better resource management by controlling the location of the deployments, in an effort to maximize efficiency. We also aim to enhance the anomaly prediction component with LSTM (Long Short Term Memory) neural networks. By creating a time series of graph observations, it is possible to predict future values and take appropriate action at an even earlier stage than it is possible now, with the current approach.

6. Acknowledgments

This work was supported by the EO4EU (AI-augmented ecosystem for Earth Observation data accessibility with Extended reality User Interfaces forService and data exploitation) project, funded by the EU's Horizon Europe Research and Innovation Programme under Grant Agreement No. 101060784.

References

[1] A. R. N. Pratik Narendra Gulhane1, A review paper on cloud computing, International Advanced Research Journal in Science, Engineering and Technology (2022).

[2] G. Liu, B. Huang, Z. Liang, M. Qin, H. Zhou, Z. Li, Microservices: architecture, container, and challenges, in: 2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C), 2020, pp. 629–635. doi:10.1109/QRS-C51114.2020.00107.

[3] Kubernetes, `Available:https://kubernetes.io/docs/concepts/architecture/`, [Online; accessed 03/01/2024] (2023).

[4] J. Shah, D. Dubaria, Building modern clouds: Using docker, kubernetes and google cloud platform, in: 2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC), 2019, pp. 0184–0189. doi:10.1109/CCWC.2019.8666479.

[5] Prometheus documentation, `https://prometheus.io/docs/introduction/overview/`, accessed: (15/03/2024).

[6] J. Kosińska, M. Tobiasz, Detection of cluster anomalies with ml techniques, IEEE Access 10 (2022) 110742–110753. doi:10.1109/ACCESS.2022.3216080.

[7] Q. Du, T. Xie, Y. He, Anomaly detection and diagnosis for container-based microservices with performance monitoring, EasyChair Preprint no. 468 (EasyChair, 2018). doi:10.29007/43km.

[8] C.-W. Tien, T.-Y. Huang, C.-W. Tien, T.-C. Huang, S.-Y. Kuo, Kubanomaly: Anomaly detection for the docker orchestration platform with neural network approaches, Engineering Reports 1 (5) (2019) e12080. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/eng2.12080, doi:https://doi.org/10.1002/eng2.12080.
URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/eng2.12080`

[9] Thanos, `https://thanos.io/`, accessed: (15/03/2024).

[10] Grafana, `https://grafana.com/`, accessed: (05/05/2024).

[11] F. López, E. Cruz, Literature review about neo4j graph database as a feasible alternative for replacing rdbms, Industrial Data 18 (2015) 135. doi:10.15381/idata.v18i2.12106.

[12] M. A. Rodriguez, P. Neubauer, The graph traversal pattern (2010). arXiv:1004.1001.

[13] B. Regmi, Neo4j graph database (01 2021). doi:10.13140/RG.2.2.35357.33764.

[14] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in python, J. Mach. Learn. Res. 12 (null) (2011) 2825–2830.

[15] S. Theodoridis, Machine Learning: A Bayesian and Optimization Perspective, Elsevier, 2020. doi:DOI: 10.1016/C2019-0-03772-7.

[16] E. Alpaydin, Introduction to Machine Learning, MIT Press, 2020.

[17] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, G. Varoquaux, API design for machine learning software: experiences from the scikit-learn project, in: ECML PKDD Workshop: Languages for Data Mining and Machine Learning, 2013, pp. 108–122.

[18] F. T. Liu, K. M. Ting, Z.-H. Zhou, Isolation forest, in: 2008 Eighth IEEE International Conference on Data Mining, 2008, pp. 413–422. doi:10.1109/ICDM.2008.17.

[19] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, A density-based algorithm for discovering clusters in large spatial databases with noise, in: Knowledge Discovery and Data Mining, 1996.
URL `https://api.semanticscholar.org/CorpusID:355163`

[20] B. Schölkopf, R. C. Williamson, A. Smola, J. Shawe-Taylor, J. Platt, Support vector method for novelty detection, in: S. Solla, T. Leen, K. Müller (Eds.), Advances in Neural Information Processing Systems, Vol. 12, MIT Press, 1999.
URL `https://proceedings.neurips.cc/paper_files/paper/1999/file/8725fb777f25776ffa9076e44fcfd776-Paper.pdf`

[21] D. Jurafsky, J. H. Martin, Speech and language processing, draft of January 7, 2023 (2023).
URL `https://web.stanford.edu/~jurafsky/slp3/5.pdf`

[22] Y. Liang, Machine learning basics - lecture 4: Svm i, `https://www.cs.princeton.edu/courses/archive/spring16/cos495/slides/ML_basics_lecture4_SVM_I.pdf` (2016).

[23] Y. Liang, Machine learning basics - lecture 5: Svm ii, `https://www.cs.princeton.edu/courses/archive/spring16/cos495/slides/ML_basics_lecture5_SVM_II.pdf` (2016).

[24] Y. Izza, A. Ignatiev, J. Marques-Silva, On explaining decision trees (2020). arXiv:2010.11034.

[25] T. Akiba, S. Sano, T. Yanase, T. Ohta, M. Koyama, Optuna: A next-generation hyperparameter optimization framework (2019).

arXiv:1907.10902.
URL https://arxiv.org/abs/1907.10902

arXiv:2410.18825v1 [cs.RO] 24 Oct 2024

A generic approach for reactive stateful mitigation of application failures in distributed robotics systems deployed with Kubernetes

Florian Mirus[1,*], Frederik Pasch[1,*], Nikhil Singhal[1], and Kay-Ulrich Scholl [1]

***Abstract*— Offloading computationally expensive algorithms to the edge or even cloud offers an attractive option to tackle limitations regarding on-board computational and energy resources of robotic systems. In cloud-native applications deployed with the container management system Kubernetes (K8s), one key problem is ensuring resilience against various types of failures. However, complex robotic systems interacting with the physical world pose a very specific set of challenges and requirements that are not yet covered by failure mitigation approaches from the cloud-native domain. In this paper, we therefore propose a novel approach for robotic system monitoring and stateful, reactive failure mitigation for distributed robotic systems deployed using Kubernetes (K8s) and the Robot Operating System (ROS2). By employing the generic substrate of Behaviour Trees, our approach can be applied to any robotic workload and supports arbitrarily complex monitoring and failure mitigation strategies. We demonstrate the effectiveness and application-agnosticism of our approach on two example applications, namely Autonomous Mobile Robot (AMR) navigation and robotic manipulation in a simulated environment.**

I. INTRODUCTION

Modern algorithms, particularly those employing cutting-edge AI, allow robots to reach a greater level of autonomy and fulfill more challenging tasks. However, on-board limitations regarding computational and energy resources are hindering factors regarding the deployment of such resource-hungry algorithms, particularly on mobile robots. On the other hand, the number and diversity of robots in a fleet for industrial automation will increase in the future. One attractive option to tackle both challenges is offloading most of the algorithmic workloads to the edge or even cloud to leverage massive computing power for robotics applications. However, given a large, heterogeneous fleet of robots with a diverse, challenging set of tasks and a complex stack of software deployed with a container management system such as K8s, one key problem is how to mitigate failures and thereby minimize their impact on the robots' and fleets' task performance. Potential failures in such a system range from failures of the compute nodes, over failures in the communication network to failures of the containerized applications themselves [1]. In previous work, we already tackled temporary communication failures for a specific application use-case, namely AMR navigation [2].

In this paper, we focus on mitigation of application failures and investigate robotics-specific requirements, that are not yet covered by failure mitigation approaches from the cloud-native domain. One core aspect of cloud-native applications is to ensure that failing micro-services are timely monitored and restarted such that, ideally, the end-user does not experience notable down-times. However, in a cloud-native deployment, containers are mostly running isolated from the physical world whereas complex robotic systems interact with the physical world in real-time and stateful failure mitigation approaches from the cloud-native domain are not directly applicable.

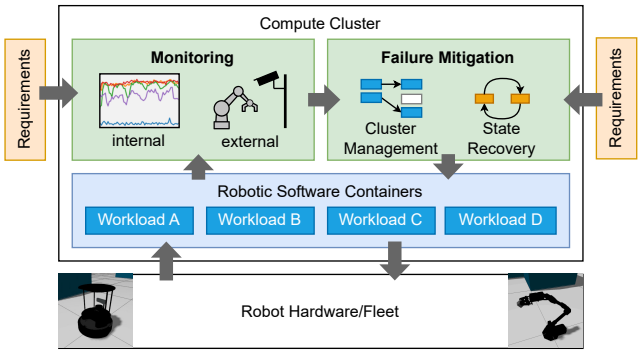


Fig. 1. High-level overview of the monitoring and failure mitigation system.

In this paper, we therefore propose a novel approach for robotic system monitoring and reactive failure mitigation for distributed robotic systems deployed using Kubernetes (K8s) and ROS2. Our approach is agnostic to the application and considers the unique challenges of distributed robotics applications by introspectively monitoring robotics-specific metrics or, monitoring the overall system's behaviour through external sensors and applying robotic-specific failure mitigation. One crucially important aspect is that our approach allows to preserve the last healthy state of the robotic system before the failure occurred and transfer it once the failure is resolved. This enables the robotic application to continue its task at the point where the failure of the workload occurred. Furthermore, our approach offers different methods for handling application failures varying in the time necessary for bringing the workload back to a functional state as well as the computational resources necessary. Finally, our approach employs Behaviour Trees [3] to encapsulate the control of the failure mitigation and thus allows arbitrarily complex failure mitigation strategies. We demonstrate the effectiveness and application-agnosticism of our approach on two example applications, namely AMR navigation and robotic manipulation, in a simulated environment.

In summary, our three main contributions are: 1) an

* Equal contribution

[1]Florian Mirus, Frederik Pasch, Nikhil Singhal and Kay-Ulrich Scholl are with Intel Labs, Karlsruhe, Baden-Würrtemberg, Germany {florian.mirus, frederik.pasch, nikhil.singhal, kay-ulrich.scholl}@intel.com

application-agnostic, reactive failure mitigation system based on the generic substrate of Behaviour Trees for distributed robotic systems deployed using Kubernetes and ROS2 allowing to preserve the last healthy state prior to the failure. 2) a robotics-specific workload monitoring system to detect failures either trough introspection, i.e., monitoring system diagnostics and Key Performance Indicators (KPIs), and/or through external supervision, i.e., observing the overall system's behaviour with external sensors and comparing the expected with actually observed behaviour. 3) several recovery strategies resulting in a trade-off between system-downtime and demand for computational resources.

II. RELATED WORK

Offloading robotic software to the edge/cloud offers a powerful enhancement to various robotic tasks [4] by leveraging high computing power and large storage spaces in the robotics domain. In recent years, researchers proposed a plethora of innovative approaches and concepts for employing edge-computing in the robotics domain ranging from applications such as perception [5], [6], grasping [7], [8], motion planning [9] and mobile navigation [10], [11]. Offloading workloads to the edge or even cloud is particularly useful for resource-hungry algorithms [12], for instance, based on modern machine learning approaches [6]. Chen et. al. [13] proposed a strategy for optimal service deployment in the cloud to ensure that the strict requirements of latency-critical robotics applications are met.

These scientific advancements are complemented from a system perspective by several architectural solutions, which aim to simplify the access to cloud computing for roboticists. For instance, FogROS2 [14] automatically provisions a cloud computer to deploy and launch ROS2 nodes while KubeROS [15] uses K8s to facilitate the deployment of ROS2-based applications across robot, edge, and cloud.

One key problem when deploying containerized applications using a container management system such as K8s is ensuring resilience against various types of failures [1], which can range from communication network failures, over compute node failures to failure of the applications or pod processes themselves. In the context of AMR navigation, a strategy to tackle temporary communication network failures by leaving minimal fallback workloads on the robot's onboard compute has been proposed in [2]. In cloud-native applications using K8s to deploy containers, failure mitigation approaches are typically separated in two categories, namely reactive and proactive [16]. Reactive approaches re-launch a failed service after a fault has occurred and was detected by the system. Proactive approaches on the other hand aim to predict potential failures before they happen employing, e.g., learning approaches ranging from forecasting neural networks such as Long Short-Term Memorys (LSTMs), k-means, or reinforcement learning and take countermeasures accordingly. Another important aspect is preserving the state of the failed application. In cloud-native applications, there is a large variety of approaches for stateful container migration [17] and stateful failure mitigation [18]. Other approaches

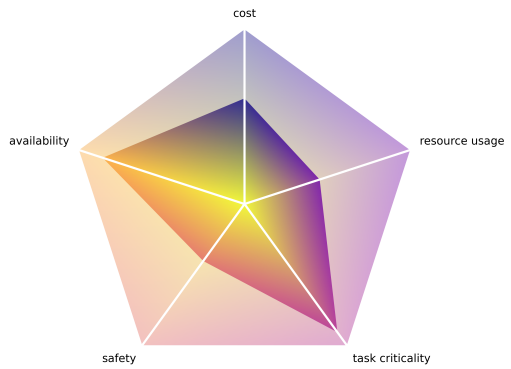


Fig. 2. Factors and possible weights for selection of Failure Monitoring and Mitigation Strategy.

focus on mitigating failures occurring in the container management system K8s itself [19].

In this paper, we aim to close the gap for a robotics-specific stateful failure mitigation for distributed robotics applications deployed with K8s. Our focus lies on a reactive approach detecting application failures either through introspection of relevant KPIs or external supervision, and a generic failure mitigation method based on Behaviour Trees. The introspection monitoring draws inspiration from monitor templates for robotic systems proposed in [20] but, for simplicity, only uses user-defined monitors instead of automatically generated ones as proposed in [20]. To the best of our knowledge, our work is the first to consider robotics-specific requirements for stateful mitigation of application failures within robotic systems deployed with the container management system Kubernetes.

III. STATEFUL FAILURE MITIGATION

Our reactive, stateful failure mitigation system consists of two main components: the monitoring system for failure detection as well as the actual failure mitigation for bringing the workload back to a functional state while preserving the last healthy state prior to the failure (see Fig 1). Both components rely on Behaviour Trees [3] to describe and apply the monitoring and failure mitigation procedures during execution time. Behavior trees are a powerful and generic substrate, that provides a formal and extensible structure for the monitoring and failure mitigation logic, which can be used to create arbitrarily complex monitoring systems and failure mitigation procedures while being human-readable and verifiable. Importantly, our failure mitigation approach is executed next to the main application without requiring to change the application's source code.

1) Taxonomy: Monitoring advanced robotic systems can be very difficult due to the systems' inherent complexity [20]. In general, there is plethora of levels at which robotics systems can and should be monitored to detect failures in the hardware, the software, at application level, at behaviour level or from a safety perspective. The choice of both, the monitoring requirements and the actual failure mitigation strategy could be tailored depending on the requirements of the application or task at hand. For instance, if the robot

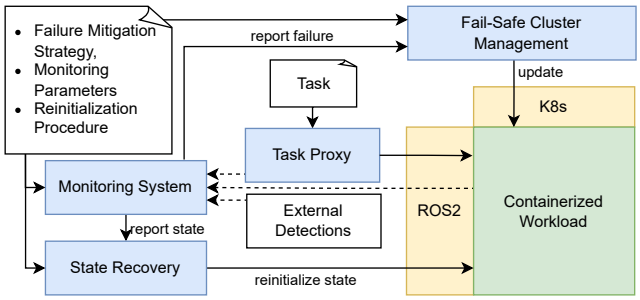


Fig. 3. System Architecture

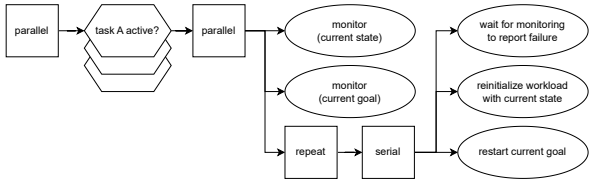


Fig. 4. Example Behaviour Tree to apply basic failure mitigation for a task and an automatically restarting workload (e.g., using Kubernetes deployment)

is performing a difficult task handling expensive materials that could be damaged while a failure occurs, it is more important to keep the system's down-time to a minimum at the cost of additional computational resources. For other cases, some level of system-downtime could be acceptable while stricter requirements regarding computational resources apply. Therefore, we propose to select the focus of the monitoring and failure mitigation system according to a taxonomy as depicted in Fig. 2 considering the requirements (such as safety, availability, resource usage, safety, task criticality just to name a few options) of tasks the robotic system is performing.

A. Monitoring System

Application- and workload-monitoring is one key ingredient to ensure meaningful and functional operation of edge- and/or cloud-systems. However, monitoring a complex robotic system interacting with the physical world poses unique challenges compared to cloud-native applications. In this paper, we focus concretely on two main monitoring approaches, namely introspective monitoring of software workloads and external supervision of the robot's behaviour and task performance.

1) Introspective monitoring: Here, we mainly apply *time-related monitors* according to the templates proposed in [20], which are characterized by a ROS2 topic's frequency. However, availability and frequency of a certain topic depend on the robot's current task and therefore a more generic monitoring solution on top of the topic monitors is necessary. For instance, consider an AMR equipped with a robotic arm, i.e., a mobile manipulator: if the frequency of the velocity command sent to the actuators of the mobile base is not consistently meeting a desired level while the robot is supposed to be navigating towards a desired goal position, the system is most likely in a failure state. In contrast, if there are no velocity commands sent to the mobile base, while the robot is standing to manipulate objects with its arm and/or end-effector, the system is considered healthy if the joint-states of the arm are delivering their data at the expected frequency. At the same time, the availability of meaningful data from various sensor sources such as Light Detection and Ranging (LIDAR), Inertial Measurement Unit (IMU), Cameras or torque sensors, needs to be monitored permanently in parallel to the task-dependent monitoring objectives, which are only active if certain conditions are fulfilled. For being able to model such hierarchical and situational monitoring requirements, we employ the generic substrate of Behaviour Trees [3]. Fig. 4 visualizes one possible Behaviour Tree for the aforementioned mobile manipulator example.

2) External supervision: Our second monitoring approach is unique for Cyber-Physical Systems (CPSs). As robotic systems interact with the physical (or simulated) world, the resulting behaviours (e.g., movement of the manipulator or the mobile base driving along path) can be monitored through external sensors such as cameras or LIDARs and state-of-the-art object detection and tracking approaches. The essential part of this monitoring approach is to compare the behaviour observed with the external sensor(s) with the expected behaviour, which is known to the system through introspection. If this comparison yields a significant discrepancy between observed and expected behaviour, the failure mitigation component of our system needs to apply countermeasures to bring the system back to a functional state.

B. Failure Mitigation

In case the monitoring system as described in Sec. III-A reports a critical failure of a workload, the failure mitigation is responsible for bringing the workload back to a functional, healthy state. Our system architecture, the interplay between the two main components and how they are connected to the monitored workload is visualized in Fig. 3. Similar to our monitoring approach, the failure mitigation procedure is encapsulated in the generic substrate of Behaviour Trees to, in general, allow the system to perform any failure mitigation procedure. Concretely, we focus on four classes of failure mitigation strategies, which result in a trade-off between down-time of the overall system after the failure occurred and the additional computational resources required for the mitigation as depicted in Fig. 5.

1) Failure mitigation strategies: The simplest way is to restart the failing workload from scratch. However, this is the slowest recovery strategy as it requires both failure mitigation at the level of the Kubernetes cluster as well as re-initialization of the failed application and thus results in some down-time of the workload. On the other hand, this recovery strategy does not require additional computational resources except for the monitoring during operation. Another option is to run a fallback instance in parallel to the main workload to speed up the recovery time. The total time necessary for recovery depends on the state the fallback workload is in, which could be either of 1) uninitialized 2) initialized and 3) in execution while receiving external data but not sending data to the remainder of the system (see Fig. 5).

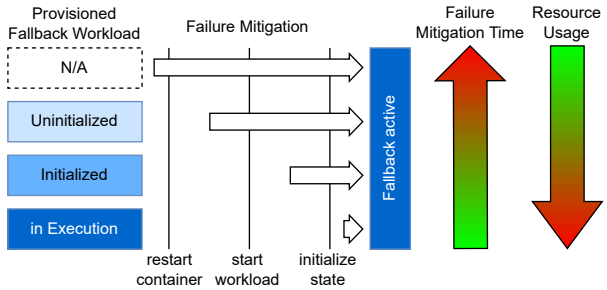


Fig. 5. Required steps for the different failure mitigation strategies and their trade-off in terms of failure mitigation time and resource usage. Depending on the provisioned fallback workload, the steps of restarting the container, workload startup and state initialization need to be executed.

All four strategies take some time at K8s cluster level either for restarting the failing workload or connecting the rest of the system to the fallback workload. Furthermore, the time for recovery at application level varies depending on the state of the fallback instance: if the fallback workload is only started, it needs to be initialized, possibly handed over the last healthy state of the failed main workload and pick up the task where the main workload failed. If the fallback workload is already initialized or even in execution mode, the necessary time for recovery reduces accordingly (see also Fig. 5 and 6 and Eq. 1).

C. State Recovery

In addition to re-launching the failing workload or connecting the running system to the fallback workload, it is crucially important that the new instance (either fallback or restarted) properly picks up where the main workload failed, i.e., the last healthy state needs to be stored during operation and handed over from the failing workload. Depending on the recovery strategy employed as described in Sec. III-B.1, the steps performed during the recovery procedure potentially differ. Therefore, the proposed failure mitigation is able to run arbitrarily complex recovery procedures encoded as Behaviour Tree, which it receives as a user-defined description (marked as *requirements* in Fig. 1). Additionally, it is possible that the failing workload critically depends on another (healthy) workload. In such a case, it could even be necessary to restart/recover the healthy workload as well. The proposed recovery approach based on Behaviour Trees also supports such complex recovery scenarios by encapsulating dependencies within the Behaviour Tree description.

D. Implementation Details

To realize the failure mitigation strategies described in Sec. III-B.1, we rely on off-the-shelf Kubernetes (K8s) components: the restart from scratch is realized through a K8s deployment [21] while rewiring the system to connect to the fallback workloads instead of the failed workloads is based on dynamically changing K8s network policies [22]. For the realizing the Behaviour Tree representation and execution of the workload monitoring and failure mitigation, we employ Scenario Execution for Robotics [23], [24], a software library that translates scenarios or, in our case and more generally, Behaviour Tree description files written in the OpenSCENARIO 2 [25] language to a Python Behaviour Tree [26] and executes it. To keep the high-level task request active, we establish a task proxy, that is capable of handling a possible workload re-initialization.

IV. EXPERIMENTS

A. Experimental Setup

We demonstrate the effectiveness of our reactive failure mitigation system for robotics applications on two example applications, namely AMR navigation and robotic manipulation in the Gazebo simulator [27], [28]. We use two different domains to demonstrate that our approach is agnostic of the application or workload it is monitoring and protecting. For AMR navigation, we use the Turlebot 4 [29] as robot platform and the WidowX-200 [30] robotic arm for manipulation.We containerize the ROS2 frameworks Nav2 [31], [32] and MoveIt2 [33], [34] for navigation and robotic manipulation respectively and execute one simple task per example application. For both applications, we execute a basic task, i.e., move the robot platform or arm to a series of user-defined goal positions, and after a user-defined time time $t_{failure}$, we delete the K8s pod containing the main workload (either Nav2 or MoveIt2) to artificially inject a failure. Again, we use Scenario Execution for Robotics [23], [24] to define and control the high-level scenarios. In both cases, our monitor checks the frequency of specific topics, i.e., the velocity commands sent to mobile base for Nav2 or the joint states for the robotic arm, and reports a failure once the stream of messages gets interrupted due to the pod containing the main workload being deleted. To demonstrate the external behaviour monitoring, we only consider the AMR navigation use-case: here, our monitor compares the expected velocity command received from introspection with the velocity command calculated from external observations. These observations are obtained by placing a camera nearby and detecting the poses of ArUco markers attached to the robot over time. To inject a failure, we remap the velocity commands such that the system assumes the mobile base receives meaningful velocity commands, whereas it actually stops moving.

1) Metrics: The first metric to consider is the time necessary to recover from an application failure and to bring the overall robotic system back to a functional state. This recovery time $t_{recovery}$ is the sum of 1) the time $t_{detection}$ necessary to detect the failure 2) failure mitigation time at cluster level (microservice/pod restart and/or time for adjusting network connections) and 3) restart and 4) re-initialization time at application level, i.e.,

$$t_{recovery} = t_{detection} + t_{cluster} + t_{startup} + t_{re-initialization}. \quad (1)$$

The second metric is the usage of computational resources. We measure the CPU usage γ_{c,t_i} at timestamp t_i for $i \in \{1,\ldots,n\}$ for each container c separately over the duration of an entire experimental run. Therefore, we use CAdvisor [35], a tool for analyzing and exposing resource usage and

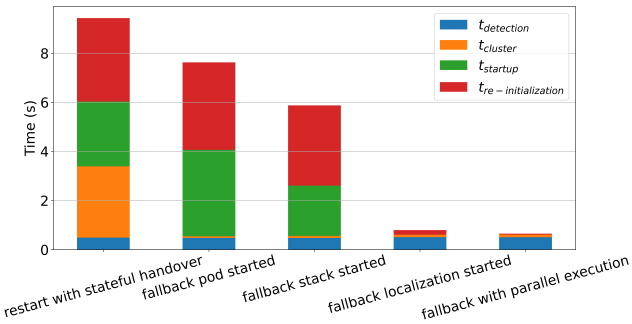


Fig. 6. Evaluation of failure recovery approaches regarding necessary time for failure mitigation.

performance data from running containers. The unit for measuring CPU usage is the K8s standard milliCPU [36] per second. The final metric is the total average CPU usage of the cluster $\sigma(CPU)$, i.e., the sum of the mean CPU usage $\sigma(c)$ per container c for all containers in the cluster $\mathfrak{C}$:

$$\sigma(CPU) = \sum_{c \in \mathfrak{C}} \underbrace{\frac{1}{n} \sum_{i=1}^{n} \gamma_{c,t_i}}_{=\sigma(c)} . \tag{2}$$

B. Evaluation of recovery strategies

1) AMR navigation use-case: Fig. 6 visualizes the individual parts of the recovery time $t_{recovery}$ according to Eq. 1 for each recovery strategy. Note that we only compare our recovery strategies against each other without a baseline, as there is - to the best of our knowledge - no other framework (e.g., FogROS2 [14] and KubeROS [15]), that offers the functionality we are proposing in this paper. The closest off-the-shelf Kubernetes counterpart is Stateful Sets [37], which however would require adaptations to the application itself, whereas our solution just uses off-the-shelf ROS2 and K8s tools without requiring application changes.

As Nav2 consists of several components, we investigate different levels of initialization for the recovery strategies involving a fallback workload ranging from just the K8s pod being started, over the Nav2 stack being started, to the fallback container running the localization workload and even the full Navigation stack in parallel. The necessary time $t_{detection}$ to detect the failure depends on the requirements specified by the user and is set to 500 ms in our experiments for all recovery strategies. The failure mitigation time at cluster level $t_{cluster}$ shows significant differences among recovery strategies: restarting a container from scratch takes K8s significantly longer (on average 32 times longer) than patching network policies to connect the system to the fallback variant of the failing workload (2.9 s vs. 0.1 s on average). Furthermore, with increasing initialization of the fallback workload, the startup time $t_{startup}$ and re-initialization time $t_{reinitialization}$ (green and red bars in Fig. 6 respectively) decrease proportionally. For the fallback workload in full execution mode, application startup is not needed and hence, the startup time collapses to $t_{startup} = 0$.

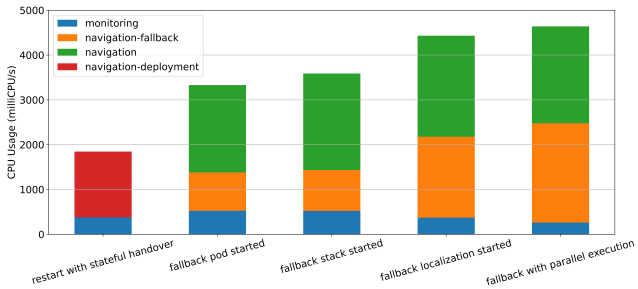


Fig. 7. Experimental evaluation of failure recovery approaches regarding CPU load.

Fig. 7 visualizes the total average CPU usage $\sigma(CPU)$ for each recovery strategy with the stacked colored bars representing the mean CPU usage of individual K8s pods containing either the workload instances, i.e., Nav2, or our monitoring and failure mitigation system. As expected, the failure mitigation strategy starting the failing workload from scratch (leftmost bar in Fig. 7) consumes the least resources on average, as there is no fallback container running in parallel to the main workload. The recovery strategies that hold an uninitialized fallback variant of the navigation stack in parallel to the main system consume at least the same resources as the restart from scratch in addition to the resources needed for running the uninitialized fallback workload in parallel. The uninitialized container consumes around 60 % of the resources necessary for the main navigation container. Similarly, the fully initialized fallback navigation stack running in execution mode in parallel consumes the same resources as the main workload. Hence, these recovery strategies are the most-resource hungry among all recovery strategies and additionally require one fallback instance for each monitored workload whereas an uninitialized workload can serve as a fallback for several main workload instances (e.g., in a robot fleet). The additional resources necessary for the monitoring component of our system is (almost) identical for all recovery strategies.

2) Manipulation use-case: Fig. 8 visualizes the recovery time for each failure mitigation strategy. Due to application differences, only a subset of the mitigation strategies from the navigation use case is applicable for manipulation, namely restart from scratch, a fallback workload with just the K8s being started and a fallback workload in full execution mode. As expected, we observe tendencies similar to the AMR navigation use-case: starting the failing workload from scratch is the slowest whereas parallel execution offers the fastest recovery. The resource usage of the recovery strategies for manipulation is similar to the navigation example, hence we omitted a dedicated figure as it offers little additional information over Fig. 7.

C. Scaling considerations

In Sec. IV-B, we have seen that there is indeed a trade-off between recovery time and necessary computational resources regarding the choice of the failure mitigation strategy. Depending on the task and application of the robotic system, the choice for a suitable recovery strategy might

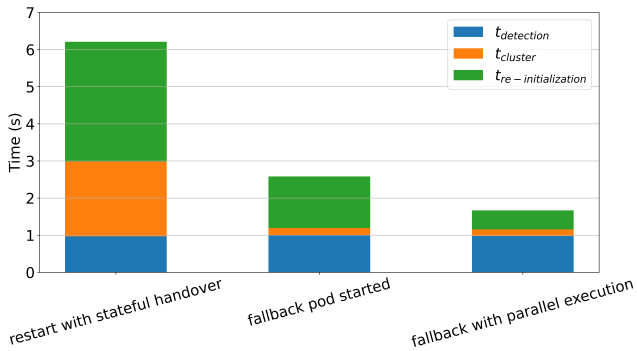


Fig. 8. Navigation: Experimental evaluation of failure recovery approaches regarding necessary time for failure mitigation.

vary. For instance, time-critical tasks involving expensive goods that call for as little down-time as possible and hence are more relaxed regarding additional resource usage will employ a recovery strategy with a fallback workload running in execution mode in parallel to the main workload. On the other extreme, workloads running in a resource-constrained cluster with less strict requirements regarding system-downtime will employ the recovery strategy starting the failing workload from scratch.

Consider a fleet of N robots and, for simplicity, only one workload per robot to monitor (e.g., navigation) with a failure of this workload causing a system downtime of t_S depending on the recovery strategy S. Additionally, let $I = \{t_1, \ldots, t_N\}$ be a discrete time interval of interest and X the number of expected failures of the entire fleet within an interval of length N. To ensure that the expected down-time of the system does not exceed the down-time t_S of the selected recovery strategy S, we need to run a sufficient number of fallback workloads. Sufficient means that the probability that more failures than available fallback workloads occur within a time-window $D_k \subset I$ of length t_S for $k = 1, \ldots, N - t_S + 1$ is sufficiently low. Assuming that the probability of a failure occurring at time-step t_i within the interval I is equal for all $i = 1, \ldots, N$, the number of possible failure occurrences $f_1, \ldots f_X$ during the time interval I is lower than

$$\binom{N+X-1}{N-1}, \tag{3}$$

because we only count instances with $f_j \in D_k$ for all $j = 1, \ldots, X$ for one time-window D_k. For example, let's assume a (fairly high) failure rate of 1 failure per hour per robot for a fleet of $N = 1000$ robots, i.e., a total of 1000 failures per hour. Hence, in a time interval I of 30 s, $\frac{1000}{120} \approx 8.3$ failures occur. By employing 4 fallback instances, the probability of the $8.3 - 4 \approx 5$ remaining failures to occur within a time-window D_k of length $t_S = 6\,\text{s}$ (which is roughly the down-time we measured for navigation for the recovery strategy holding an uninitialized fallback workload), i.e., leading to a longer system-downtime is at 1.2 %. To reduce the down-time by factor 10 compared to the restart from scratch by having one fallback workload in parallel execution mode for all 1000 robots, the CPU usage in turn would be double whereas 4 additional fallback workloads would reduce the down-time by a factor of 2 and only add 0.4 % resource usage compared to the restart from scratch. Hence, a recovery strategy holding an uninitialized fallback workload in parallel offers a good balance between system down-time and additional resource usage. Particularly, for larger robot fleets, the additional compute resources necessary in such a setting is neglectable.

V. DISCUSSION

A. Conclusion

In this paper, we presented an application-agnostic, reactive failure mitigation system based on the generic substrate of Behaviour Trees for distributed robotic systems deployed using Kubernetes and ROS2 allowing to preserve the last healthy state prior to the failure. Our approach consists of a robotics-specific workload monitoring system to detect failures either trough introspection, i.e., monitoring system diagnostics and KPIs, and/or through external supervision, i.e., observing the overall system's behaviour with external sensors. Finally, we presented different recovery strategies resulting in a trade-off between system-downtime and demand for computational resources and demonstrated the effectiveness of our approach at two example applications, namely AMR navigation and robot manipulation.

B. Future Work

Although we believe that our approach is an important step towards making distributed robotic systems deployed with a container management system such as K8s resilient against application failures, there are several options for future work. In this paper, we assumed that the workload to be monitored runs entirely in one container inside K8s. However, the navigation stack for instance, consists of multiple individual workloads such as localization, path planning and trajectory execution that could run in separate containers following a more micro-service-oriented architecture. At the same time, these workloads critically depend on each other in the sense that a failure in the localization module directly affects the functionality of the path planning module. Hence, in a micro-service-oriented software architecture, the monitoring and failure mitigation system needs to take such dependencies into account and act accordingly when mitigating failures. Although our proposed monitoring system is, in principle, able to encode such a hierarchical monitoring system in the substrate of Behaviour Trees, more research and evaluation is necessary to realize it. Additionally, we aim to add monitoring vectors along the lines of our taxonomy as described in Sec. III-.1, for instance, safety-focused monitoring. Finally, we aim to incorporate our monitoring and failure mitigation system in a larger orchestration solution, which is not only able to mitigate failures but also reschedule containers and compute resources in case the monitoring solution detects sub-optimal behaviour of individual workloads above failure-level.

REFERENCES

[1] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Microservice Based Architecture: Towards High-Availability for Stateful Applications with Kubernetes," in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, 2019, pp. 176–185.

[2] F. Mirus, F. Pasch, and K.-U. Scholl, "Towards fault-tolerant deployment of mobile robot navigation in the edge: An experimental study," in *41st IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2024, pp. 6791–6797.

[3] M. Colledanchise and P. Ögren, *Behavior Trees in Robotics and AI: An Introduction*, 1st ed. CRC Press, 2018.

[4] B. Kehoe, S. Patil, P. Abbeel, and K. Goldberg, "A Survey of Research on Cloud Robotics and Automation," *IEEE Transactions on Automation Science and Engineering*, vol. 12, no. 2, pp. 398–409, Apr. 2015.

[5] W. J. Beksi, J. Spruth, and N. Papanikolopoulos, "Core: A cloud-based object recognition engine for robotics," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2015, pp. 4512–4517.

[6] A. Rashid, C. M. Kim, J. Kerr, L. Fu, K. Hari, A. Ahmad, K. Chen, H. Huang, M. Gualtieri, M. Wang, C. Juette, N. Tian, L. Ren, and K. Goldberg, "Lifelong LERF: Local 3D Semantic Inventory Monitoring Using FogROS2," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, 2024, pp. 7740–7747.

[7] B. Kehoe, D. Warrier, S. Patil, and K. Goldberg, "Cloud-based grasp analysis and planning for toleranced parts using parallelized Monte Carlo sampling," *IEEE Transactions on Automation Science and Engineering*, vol. 12, no. 2, pp. 455–470, 2014.

[8] M. Zahid and F. T. Pokorny, "CloudGripper: An Open Source Cloud Robotics Testbed for Robotic Manipulation Research, Benchmarking and Data Collection at Scale," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, 2024, pp. 12 076–12 082.

[9] J. Ichnowski, W. Lee, V. Murta, S. Paradis, R. Alterovitz, J. E. Gonzalez, I. Stoica, and K. Goldberg, "Fog robotics algorithms for distributed motion planning using lambda serverless computing," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020, pp. 4232–4238.

[10] M. Groshev, G. Baldoni, L. Cominardi, A. de la Oliva, and R. Gazda, "Edge robotics: are we ready? an experimental evaluation of current vision and future directions," *Digital Communications and Networks*, May 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352864822000888

[11] M. Balogh, A. Vidács, G. Fehér, M. Maliosz, M. Á. Horváth, N. Reider, and S. Rácz, "Cloud-Controlled Autonomous Mobile Robot Platform," in *2021 IEEE 32nd Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, 2021, pp. 1–6.

[12] S. Chinchali, A. Sharma, J. Harrison, A. Elhafsi, D. Kang, E. Pergament, E. Cidon, S. Katti, and M. Pavone, "Network offloading policies for cloud robotics: a learning-based approach," *Autonomous Robots*, vol. 45, no. 7, pp. 997–1012, July 2021.

[13] K. Chen, M. Wang, M. Gualtieri, N. Tian, C. Juette, L. Ren, J. Ichnowski, J. Kubiatowicz, and K. Goldberg, "FogROS2-LS: A Location-Independent Fog Robotics Framework for Latency Sensitive ROS2 Applications," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, 2024, pp. 10 581–10 587.

[14] J. Ichnowski, K. Chen, K. Dharmarajan, S. Adebola, M. Danielczuk, V. Mayoral-Vilches, N. Jha, H. Zhan, E. Llontop, D. Xu, C. Buscaron, J. Kubiatowicz, I. Stoica, J. Gonzalez, and K. Goldberg, "FogROS2: An Adaptive Platform for Cloud and Fog Robotics Using ROS 2," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 2023, pp. 5493–5500.

[15] Y. Zhang, C. Wurll, and B. Hein, "KubeROS: A Unified Platform for Automated and Scalable Deployment of ROS2-based Multi-Robot Applications," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 2023, pp. 9097–9103.

[16] Z. Huang and H. Huang, "Proactive Failure Recovery for Stateful NFV," in *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*, 2020, pp. 536–543.

[17] P. S. Junior, D. Miorandi, and G. Pierre, "Stateful Container Migration in Geo-Distributed Environments," in *2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2020, pp. 49–56.

[18] M.-N. Tran, X. T. Vu, and Y. Kim, "Proactive Stateful Fault-Tolerant System for Kubernetes Containerized Services," *IEEE Access*, vol. 10, pp. 102 181–102 194, 2022.

[19] M. Barletta, M. Cinque, C. D. Martino, Z. T. Kalbarczyk, and R. K. Iyer, "Mutiny! How does Kubernetes fail, and what can we do about it?" 2024. [Online]. Available: https://arxiv.org/abs/2404.11169

[20] H. Jiang, S. Elbaum, and C. Detweiler, "Inferring and monitoring invariants in robotic systems," *Autonomous Robots*, vol. 41, no. 4, pp. 1027–1046, 2017.

[21] Kubernetes. (2024) K8s deployment. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

[22] ——. (2024) K8s Network Policies. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/network-policies/

[23] F. Pasch, F. Mirus, Y. Zhang, and K.-U. Scholl, "Scenario Execution for Robotics: A generic, backend-agnostic library for running reproducible robotics experiments and tests," 2024. [Online]. Available: https://arxiv.org/abs/2409.07080

[24] Frederik Pasch and Florian Mirus. (2024) Scenario Execution for Robotics. [Online]. Available: https://github.com/IntelLabs/scenario_execution

[25] Association for Standardization of Automation and Measurement Systems (ASAM). (2024) OpenScenario V2.0. [Online]. Available: https://www.asam.net/project-detail/asam-openscenario-v20-1/

[26] Py-trees. (2024) Py-trees. [Online]. Available: https://py-trees.readthedocs.io/en/devel/introduction.html

[27] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, vol. 3, 2004, pp. 2149–2154 vol.3.

[28] Open Robotics. (2024) Gazebo. [Online]. Available: http://gazebosim.org/home

[29] C. Robotics. (2024) Turtlebot 4. [Online]. Available: https://clearpathrobotics.com/turtlebot-4/

[30] Interbotix. (2024) WidoxX-200. [Online]. Available: https://docs.trossenrobotics.com/interbotix_xsarms_docs/specifications/wx200.html

[31] S. Macenski, T. Moore, D. V. Lu, A. Merzlyakov, and M. Ferguson, "From the desks of ROS maintainers: A survey of modern & capable mobile robotics algorithms in the robot operating system 2," *Robotics and Autonomous Systems*, p. 104493, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S092188902300132X

[32] S. Macenski. (2024) Nav2. [Online]. Available: https://docs.nav2.org/

[33] D. Coleman, I. A. Sucan, S. Chitta, and N. Correll, "Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study," *Journal of Software Engineering for Robotics*, vol. 5, no. 1, pp. 3–16, 2014.

[34] I. A. Sucan and S. Chitta. (2024) MoveIt2. [Online]. Available: https://moveit.ros.org/

[35] Cadvisor. (2024) CAdvisor. [Online]. Available: https://prometheus.io/docs/guides/cadvisor/

[36] Kubernetes. (2024) K8s, the meaning of CPU. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#meaning-of-cpu

[37] ——. (2024) K8s Stateful Sets. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/

KGSecConfig: A Knowledge Graph Based Approach for Secured Container Orchestrator Configuration

Mubin Ul Haque, M. Mehdi Kholoosi, and M. Ali Babar
Centre for Research on Engineering Software Technologies (CREST)
School of Computer Science, and Engineering, The University of Adelaide, Adelaide, Australia
Cyber Security Cooperative Research Centre

arXiv:2112.12595v1 [cs.CR] 21 Dec 2021

***Abstract*—Container Orchestrator (CO) is a vital technology for managing clusters of containers, which may form a virtualized infrastructure for developing and operating software systems. Like any other software system, securing CO is critical, but can be quite challenging task due to large number of configurable options. Manual configuration is not only knowledge intensive and time consuming, but also is error prone. For automating security configuration of CO, we propose a novel Knowledge Graph based Security Configuration, KGSecConfig, approach. Our solution leverages keyword and learning models to systematically capture, link, and correlate heterogeneous and multi-vendor configuration space in a unified structure for supporting automation of security configuration of CO. We implement KGSecConfig on Kubernetes, Docker, Azure, and VMWare to build secured configuration knowledge graph. Our evaluation results show 0.98 and 0.94 accuracy for keyword and learning-based secured configuration option and concept extraction, respectively. We also demonstrate the utilization of the knowledge graph for automated misconfiguration mitigation in a Kubernetes cluster. We assert that our knowledge graph based approach can help in addressing several challenges, e.g., misconfiguration of security, associated with manually configuring the security of CO.**

Keywords- Container, Configuration, Security, Knowledge Graph

I. Introduction

Container Orchestrator (CO), such as Kubernetes [1], Docker-Swarm [2], Nomad [3] enable system administrators to perform multiple tasks, e.g., scaling and interconnecting a large number of containers seamlessly [4]. Furthermore, organizations gain a significant benefit in deploying containers in clouds across different production environments [5] for rapid software delivery using continuous software engineering.

Despite the reported benefits of using CO, security is one of the key concerns while deploying containers in CO [6], [7]. CO is typically used to manage thousands of containers hosting business critical services, whose security can be compromized if CO is not secured.

That is why CO security misconfigurations have emerged as one of the biggest concerns of system administrator and software professionals [8]. In a recent survey [9], CO security misconfiguration has been mentioned as the top concern (59%). By exploiting a misconfigured identity management roles of a CO in a Capital One system in 2019 [10], hackers accessed 30 GB of application data containing valuable financial and personal information of 106 million people.

If the security of a CO is not appropriately configured, it cannot enforce security policies (e.g., restriction of capabilities, accessibility of root files, privileges to run containers) for the underlying clusters on its own as the default options are not always security-focused [11]. Therefore, system administrators need to manually perform security configuration tasks, such as identifying and updating the default options, implementing configuration arguments and the options involved in the security policy [12]. Moreover, when deploying containers in a cluster, system administrators may need to maintain diverse configuration files for different purposes, such as deployment (e.g., replica, namespaces), managing credentials (e.g., roles and resources), and observing performances (e.g., audit and log). Each of such configuration files consists of several arguments. For example, a deployment configuration file in Kubernetes may consist of 40 configuration arguments [13] and more than 100 options. Besides, CO has a distributed architecture constituting different components with a vast configuration space, e.g., arguments, options, default values and types. For example, Kubernetes consists of eight different components, more than 1K configuration arguments across all components, which require manual intervention for configuration [12]. Thus, the process of manually identifying and configuring a vast configuration space and different configuration files from the security perspective in CO is effort-intensive, time-consuming, and error-prone [11], [12].

Therefore, automation is highly desirable for securing the configuration of CO to reduce the required effort and minimize the risk of misconfiguration [7], [14]. However, automation support may need to overcome several challenges, such as configuration data *scatteredness*, *dynamicity*, and *overload* for automating the security configuration task of CO. We provide brief explanation of each of these challenges below to help better understand our research's motivators.

Automation of CO security configuration is a knowledge-intensive task; it is important to know 'what' options are required for 'which' arguments for each component of a CO. This task is not trivial due to a large number of available configuration options and arguments. In addition, CO is typically integrated with multi-vendor software systems [9], such as Continuous Integration and Development or Deployment (CI/CD) tools, e.g., Jenkins; Development and Operations (DevOps) platforms, e.g., Docker; and cloud resource providers, e.g., Azure. Therefore, administrators need to investigate each of the *scattered* tools, platforms, or providers' configuration space for identifying and capturing the secured configuration

arguments and options.

CO security configuration options data may *dynamically* change due to new vulnerabilities, malware, security flaws, and patch releases. For example, the configuration option for the argument *'imagePullPolicy'* in Kubernetes was expected to change after Distributed Denial of Service (DDoS) attack in DataDog [11]. Hence, to mitigate the rapidly changing threat landscape, configuration space needs to be updated continuously, which requires a frequent manual monitor and search of diverse data sources (e.g., HTML, JSON or XML).

Cyber security information *overload* as the number and the sources of threat advisories are increasing. Whilst threat advisories contain in-depth information on how attackers target the existing configuration to compromise software systems, it is a cognitively challenging task to separate relevant from the irrelevant information. For example, to obtain the secured configuration options for kube api server [15] (one of the eight constituent components of Kubernetes), relevant documents contain 400 sentences on average, whereas the concepts, i.e., relevant sentences, such as reason, implementation details, are 30 sentences on average.

There is a need for a unified solution for secured CO configuration, constituting configuration space associated with their secured arguments, options, and concepts of heterogeneous, multi-vendor, and diverse tools, platforms, providers for enabling automation of secured CO configuration.

There are a few studies that have discussed the best security practices [7] and defects [14] for Kubernetes. Moreover, Kubernetes usage for monitoring resources and performances had also been reported in [16], [17]. However, there is a general lack of solutions for automating CO security configuration.

To address the above identified problem, we propose to build a KG for Secured Configuration (KGSecConfig) that provides a fundamental support for automatic security configuration of CO. KG can provide the unification of configuration space associated with their secured arguments, options, and concepts in terms of entities and relations. For example, configuration options, arguments, types, default values, code snippets (e.g., Infrastructure-as-Code (IaC)) represent entities. Relations connect entities using phrases that describe relationship among entities, such as *'is'* or *'has'* relationship. Besides, we propose a keyword and learning-based model to address the *dynamicity* and *overload* barriers, which can automatically extract secured configuration knowledge in terms of secured arguments, options, and concepts hidden in large documents. These types of knowledge are populated in the KG for organized structure, so that the encapsulated knowledge can be utilized for automating the security configuration of CO.

Our KGSecConfig can capture and link the configuration knowledge through real-time monitoring of multiple, diverse, and heterogeneous configuration space from different data sources to reflect the intricate and evolving security threats. Our KGSecConfig can be utilized for various downstream configuration tasks, such as automated misconfiguration mitigation and interpretation of secured configuration options which can increase the understanding of the reason of enabling/disabling particular arguments. Furthermore, our KGSecConfig can provide a way for visualizing configuration arguments where default values are not security-focused and can help administrators to prioritize their efforts in configuration task.

Our paper makes the following three main contributions:

- We are the first, to the best of our knowledge, to propose a keyword and learning-based approach for CO to capture, store and correlate configuration knowledge automatically.
- We build a secured configuration knowledge graph using KGSecConfig on Kubernetes [1], Docker [18], Azure [19], and VMWare [20].
- Our evaluation shows high accuracy to build the KGSecConfig. Moreover, we demonstrated the effectiveness of KGSecConfig in automated mitigation of misconfiguration of a Kubernetes cluster.

The remainder of our paper proceeds as follows. Section II describes the related work. Section III and IV discuss the approach and implementation. The process of the evaluation and results are discussed in Section V. We report the implications and limitations in Sections VI and VII. Finally, our paper concludes in Section VIII with some future directions.

II. Related Work

Our research is related to the prior studies that have investigated the security of CO, the configuration studies in software engineering, and cyber security KG.

A. Security in Container Orchestrator

Shamim et al. [7] reported a study of grey literature, e.g., blog posts and tutorials to identify security best practices for Kubernetes. They stated secure usage of Kubernetes requires the implementation of security practices applicable to multiple components within Kubernetes, such as containers [21] and pods [22]. They also advocated for a need of a deep understanding of Kubernetes configurations to implement security practices. Bose et al. [14] conducted a study on open-source software repositories, e.g., GitLab, and identified commits for updating security-related defects in Kubernetes manifests [7]. They applied closed coding [23], a qualitative analysis technique, on the collected commits to determine commits related to a security defect. Moreover, there were studies on the usage of Kubernetes in creating monitoring systems [16] and comparison of performance and resource management [17]. However, our research is different to these studies as we aim to develop an approach which can automatically capture, link, and encapsulate the configuration knowledge of diverse configuration space in a unified KG. which can be used for secured configuration of CO.

B. Configuration in Software Engineering

Sayagh and Hassan [24] identified the appropriate options to configuration related user questions by mining already answered configuration questions on online forums. Jin et al. [25] and Wen et al. [26] identified the appropriate options to a configuration-related question by extracting options whose option names are textually similar to the new user question. Xia

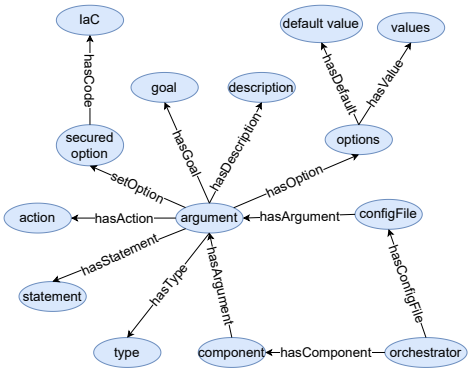


Fig. 1: Conceptual schema for configuration knowledge

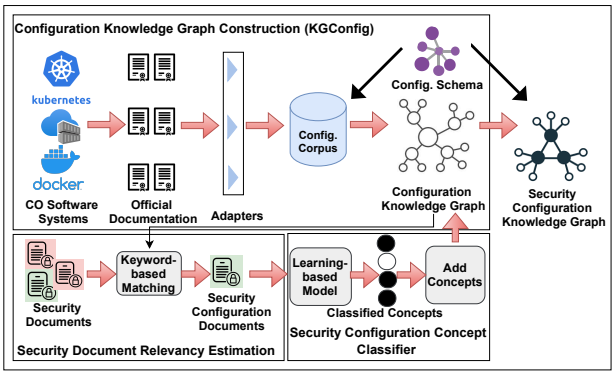


Fig. 2: Approach for constructing KGSecConfig

et al. [27] predicted if a bug report is related to configuration or not. However, the study [24] focused on the already answered question; thus, there would be a long trail of less discussed configuration options. Moreover, all the configuration options might not be discussed on online forums, and the answered options might not be reliable, as shown by prior studies [28], [29]. Besides, given a bug report, the study [27] determined whether the bug report was a configuration or not, and the studies [25], [26] extracted configuration options from the bug reports. However, our research goal is different as our motivation is not only to extract configuration options but also realize and extract configuration concepts (e.g., reason, implementation details). We specifically intend to encapsulate such extracted configuration knowledge in a structured way in the unified knowledge graph. This unified knowledge graph allows us to correlate the vast configuration space of diverse CO software systems and, afterward, enables us to automate the secured configuration and compliance.

C. Cyber Security Knowledge Graph

Rastogi et al. [30] proposed a KG for predicting missing information for malware. Pingle et al. [31] proposed a system, RelExt, that would detect relationships and create semantic triples over cyber security text using a deep-learning approach. Piplai et al. [32] extracted the information from malware After Action Reports (AAR) that can be merged to create a KG using RelExt. Mendsaikhan et al. [33] identified the significance of the cyber security text using the deep-learning approach. Whilst the prior studies focused on detecting cyber security text, e.g., attack patterns, vulnerabilities, malware, our study, in contrast, aims to build a KG automatically for secured configuration knowledge for CO.

In summary, our research, to the best of our knowledge, has provided a unique solution, KGSecConfig, for building a secured configuration knowledge graph automatically for diverse software systems in CO. Our KGSecConfig is expected to minimize the manual efforts required to extract and utilize configuration knowledge for securing CO configuration.

III. Approach

We propose a Knowledge Graph (KG) based approach to overcome the above-mentioned barrier, i.e., data *scatteredness*, *dynamicity* and *overload*, to automating the configuration of CO. The conceptual schema of our KGSecConfig is shown in Fig. 1; and Fig. 2 shows the different stages of envisioned process for automatically constructing KGSecConfig. In particular, we present how KGSecConfig can be constructed from the natural language text sources by retrieving the entities and relations to form the triples [31], [32] of KGSecConfig. Our approach contains three main modules named *Configuration Knowledge Graph (KGConfig) Construction*, *Security Document Relevancy Estimation*, and *Security Configuration Concept Classifier*. The details of each module are described as follows.

A. Configuration Knowledge Graph (KGConfig) Construction

The purpose of this module is to create a Knowledge Graph for Configuration (KGConfig) of heterogeneous platforms, cloud resource providers, and tools used in CO to mitigate data *scatteredness*. We encapsulated configuration arguments, options, types, default values, and descriptions. Configuration arguments and options are indispensable knowledge for performing the configuration task (e.g., *'RBAC'* is an option in *'–authorization-mode'* argument for configuring kube apiserver). Types and default values are necessary knowledge to verify whether the used types (e.g., strings for *'–audit-policy-file'* argument, or integer for *'MinReadySeconds'* argument) and values in the configuration file (e.g., Kubernetes manifest file) are aligned with the required setting. Descriptions are essential knowledge for discovering and understanding the functionality of the configuration argument. In this module, we used different adapters to extract the configuration information, e.g., arguments, options, descriptions, and default options, from official documentation of diverse orchestrators, tools, platforms and built a configuration corpus with the extracted information. Then we applied our schema as shown in Fig. 1 to the configuration corpus to develop KGConfig. We need different adapters (e.g., data scrapers) since the documentation content is distributed in diverse formats such HTML, XML, or JSON. One example of extracting configuration information from documentation and constructing triples to populate KGConfig is illustrated in Fig. 3 and Fig. 4. The extracted arguments, options, types, and default options are considered as the entities and we defined *'has'* relationship, e.g., *'hasArgument'*, *'hasOption'*, *'hasType'*, *'hasDefault'* and

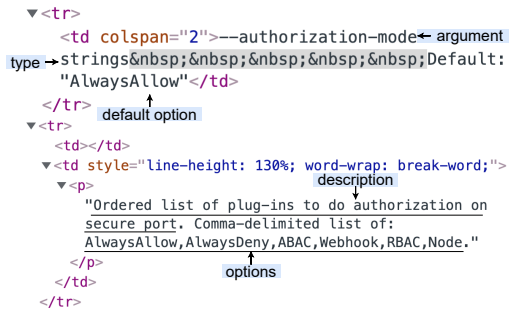


Fig. 3: Example of configuration space extraction

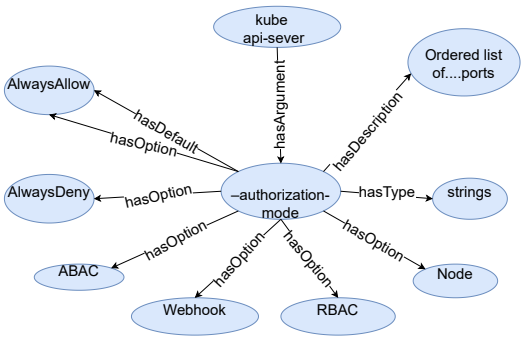


Fig. 4: Triples corresponding to Fig. 3

'hasDescription' among the entities. These entities and relations are important for identifying the configuration syntax and used to formulate keyword-based rules for estimating the relevancy of security documents with configuration (Section III-B). In summary, the input of this module is a set of official documentations, and the output is a KGConfig consisting of configuration entities and corresponding relations.

B. Security Document Relevancy Estimation (SDRE)

The purpose of this module is to filter out security documents from configuration point of view to mitigate data *dynamicity* barrier. Our KGSecConfig extracts security configuration information from publicly available security information-sharing sources. While security information-sharing sources (e.g., National Vulnerability Database (NVD), Security Tool White Paper, Security Bulletin) have grown in popularity [33], the amount of shared security information has also increased tremendously. This security information is shared in text documents and all the documents may not be relevant to configuration. Fig. 5 and Fig. 6 show the examples of two excerpts from Kubernetes Blog [34], where a document is relevant to configuration and other is not relevant to configuration. In Fig. 6, **bold** text (e.g., configuration argument) represents the reason of a sentence being relevant to configuration. In particular, we labeled a sentence that is relevant with configuration if any word in the sentence is matched with the extracted configuration argument, which is obtained from the previous step (Section III-A). In the same way, we considered a security document is relevant to configuration if there is any sentence being labelled as configuration. We used configuration argument and its associated entities as the

> Depending on the attack surface of your application, you may want to focus on specific aspects of security. For example: If you are running a service (Service A) that is critical in a chain of other resources and a separate workload (Service B) which is vulnerable to a resource exhaustion attack then the risk of compromising Service A is high if you do not limit the resources of Service B.

Fig. 5: Example of a non-configuration document

> Ensure that the **--authorization-mode** argument includes RBAC. Edit the API server pod specification file /etc/kubernetes/manifests/kubeapiserver.yaml on the master node. RBAC ensures fine-grained control over the operations that different entities can perform on different objects in the cluster. RBAC also restricts unauthorized access to the kube api-server.

Fig. 6: Example of a configuration document

keyword for estimating the relevancy of security documents to configuration. In summary, the input of this module is a set of security documents and the output is a set of security configuration documents.

C. Security Configuration Concept Classifier (SCCC)

The purpose of this module is to realize and extract configuration concepts from security configuration documents to mitigate data *overload* barrier. We proposed four main classes for security configuration concepts as (i) *'statement'* concept describes to perform a security configuration task (e.g., Ensure that the *'–authorization-mode'* argument includes *'RBAC'*); (ii) *'goal'* concept describes the reason behind the security configuration task (e.g., *'RBAC'* ensures fine-grained control over the operations...cluster, *'RBAC'* restricts unauthorized access...server); (iii) *'action'* concept describes the actionable steps to implement the task (e.g., Edit the API server pod specification file...node); and (iv) *'other'* concept describes the rest of sentences in the document (e.g., In this article, we will take a deep dive into key Kubernetes security configurations and the recommended best practices).

A secured configuration *'statement'* concept is essential to correlate statements with actionable steps in the knowledge graph. For example, Security Announcement Kubernetes [35] did not provide the actionable steps to mitigate the security issues due to CVE-2020-8557, however, it mentioned the statement (e.g., force containers to drop CAP-DAC-OVERRIDE capabilities in PodSecurityPolicy). In this regard, storing a statement in the knowledge graph can help find actionable steps for implementing secured configuration tasks. Moreover, *'goal'* concept is essential to demonstrate 'why' a particular option should be used or not for security purposes. Dietrich et al. described system administrators suffer from a lack of knowledge in configuration, which is one of the primary reasons for misconfiguration [36]. Therefore, the *'goal'* concept provides a way to interpret the cause of a particular configuration, which can be influential in broadening the configuration knowledge. The *'action'* concept provides the required low-level implementation details, which are necessary for the machine execution of configuration tasks. Besides, *'other'* concept is necessary to filter out irrelevant sentences from the security configuration document.

We leveraged learning-based techniques to classify the sentences in security configuration documents automatically.

Learning-based techniques are used since they are able to independently adapt new data for their learning ability from the previous computation, e.g., historical data for producing repeatable and data-driven decisions [37], [38], [39]. We designed the problem as a multi-class supervised text classification problem from a learning perspective. The components of building a learning model include- text **pre-processing**, **model selection**, **model building**, and **prediction**. **Pre-processing.** security configuration document may contain noise (e.g., punctuation, stop-words), which can make the learning model overfit [37], [40]. Therefore, we used the state-of-the-art approaches [41] for pre-processing the sentences, e.g., removal of noises, stop-words, lower casing, and lemmatization. Lemmatization is used to minimize the inflectional and derivational forms of a word to a standard base form [38]. We left the configuration argument and options intact, e.g., *'–anonymous-auth'*, to preserve the configuration syntax. **Model Selection.** we used the pre-processed text to perform stratified k-fold cross-validation. Stratification ensures the ratio of each input source is kept throughout the cross-validation [37], avoiding different data distribution of the folds. Our model selection component has two steps as (i) feature engineering, (ii) model training and validation. Feature engineering is the process where textual data is transformed into features to improve the performance of the learning models [42]. In the model training and evaluation steps, (k-1) folds are used for feature engineering and training a model, while the remaining one is used for validation. The validation performance of a model is the average of k runs. The model configurations with the highest performance metric would be selected as the optimal classifier for the following model building process. **Model Building.** the model building process uses the pre-processed data to generate a feature model based on the identified feature configuration. The feature model has been saved to transform the data for future prediction. **Prediction.** the prediction process is used for both testing the trained model and classifying the new sentences. In this process, the sentences are first pre-processed and then transformed to a feature set using the saved feature model. Finally, the feature set is used by the saved trained model to determine the class of the sentences.

Our SCCC module will reduce the manual analysis required for realizing concepts from security configuration documents. In summary, the input of this module is a security configuration document and the output is the sentences classified according to the four configuration concepts.

Once we identified the classified sentences, we considered all the concepts except *'other'* concept for updating the initial KGConfig. We used graph-traversal algorithm [43] to locate the configuration argument and then added concepts as entities with *'has'* relationship as described in Fig. 2 to construct a secure configuration knowledge graph. Moreover, we identified the required options for security configuration mentioned in the concepts by a rule-based approach. The rules are formulated from the available options associated with the particular arguments that had been initially encapsulated in the KGConfig. Fig. 7 shows an example to construct a secure configuration knowledge graph from the identified concepts.

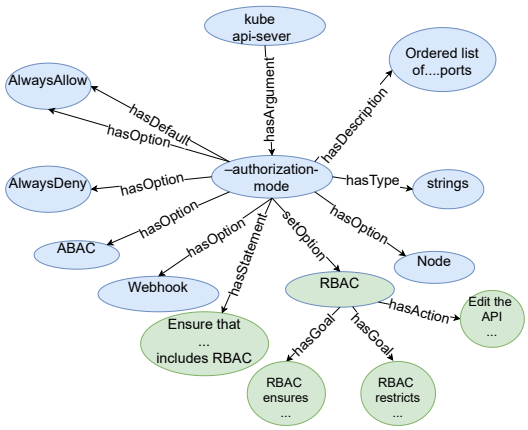


Fig. 7: Configuration concepts added to KGConfig (Fig. 4) from the text of Fig. 6

To mitigate data *scatteredness* barrier, we proposed a configuration schema which is used to capture the configuration arguments, descriptions, default, and available options from diverse, heterogeneous, and multi-vendor tools documentation using entities and relations. While the schema can be used for instantiating KG, it can also map data of different formats (e.g., structured, unstructured, or semi-structured) from multiple sources into a common structure (Section III-A). Secondly, we proposed a keyword-based configuration knowledge localization method (Section III-B) to overcome data *dynamicity* barrier. Thirdly, to mitigate the data *overload* barrier, we proposed a learning-based model (Section III-C). The proposed learning-based model can automatically adopt new data from diverse sources to keep the configuration updated for mitigating security incidents.

IV. Implementation

We applied our knowledge graph construction approach to Kubernetes [1] (container orchestrator), Docker [18] (DevOps platform), Azure [19] (cloud resource provider), and VMware [20] (infrastructure manager) which can be integrated with Kubernetes [44]. We selected the above four software systems due to their large-scale adoption for integrated orchestration service in a software-defined network [44].

We used Beautiful Soup [45], a Python library for parsing HTML and XML documents to crawl the web pages of official documentation for knowledge extraction. In addition, NLTK [46], a Python library for text processing, was used to implement the text pre-processing in descriptive knowledge extraction. Beautiful Soup and NLTK were widely used in software engineering studies involving information retrieval process [47], [48].

We leveraged following security information-sharing sources for secured configuration knowledge extraction.

- **Security Announcement Kubernetes:** Official site [35] for announcing security adversaries discovered or reported for Kubernetes and its integrated software systems, e.g., Docker or Azure. This site is maintained by

TABLE I: Description of data set

Source	Official	Internet	Vulners
Security-related Document	42	101	2187
Security Configuration Document	39	91	205

Kubernetes security experts and recommended for the Kubernetes users for security task information [1].

- **Internet Artifacts:** Shamim et al. proposed a taxonomy for secured installation of Kubernetes clusters and shared a dataset of 101 blog posts [7] from internet. Researchers have acknowledged the value of internet artifacts in deriving security tasks in various domains, such as DevOps [49], CI/CD [50], and testing [51].
- **Security Tool White Paper:** Security tools, such as SYSDIG [52] release white papers to secure deployment of container workloads and contain valuable information. The previous researches showed the importance of analyzing white papers for realizing security tasks [53]. We used Vulners database [54], a rich source of security task information for diverse and heterogeneous software systems from more than 50 security tools. Vulners database was also used by other researches [39], [55], [56].

Table I shows the distribution of the documents related to security and configuration. We used our 5,172 arguments as a keyword to search and identify security configuration-related documents. The search was performed in August 2021.

We required a labelled dataset for the supervised learning model to extract configuration concepts from the security configuration documents. To the best of our knowledge, there is no labelled dataset for security configuration concept classification. Manual labelling of data is time-consuming and labour-intensive [38]. Therefore, we randomly selected 3,300 sentences from the security configuration documents for manual labelling. Two authors manually and independently label the sentences based on our four configuration concepts (Section III-C). We used Cohen's Kappa [57] to measure the agreement between two labelers. Cohen's Kappa is used since the same two labelers had rated the set of selected sentences, and also used in software engineering studies to report the agreement [58], [59]. We obtained the Kappa value of 0.7, indicating substantial agreement [57]. We used 3,032 sentences which were agreed upon by both labelers for training the model to reduce the labeling bias.

Five traditional machine learning classifiers, Logistic Regression (LR), Naive Bayesian (NB), Support Vector Machines (SVM), Random Forest (RF), and Extreme Gradient Boosting (XGB), were selected for learning-based models. Those classifiers were chosen due to the common practice in the literature [60], [61], [62] and real data science competition (e.g., Kaggle [63]). The first three (e.g., LR, NB, SVM) are single models, whereas the rest two (RF, XGB) are ensemble models [37]. In addition, we considered Term Frequency-Inverse Document Frequency (TF-IDF)-based word level, character level, and combination of word and character level to obtain relevant features. We also considered NLP features, such as word count, character, noun, verb, adjective, adverb, pronoun, and word density. These features are common and also used in the previous studies of software engineering studies that involve classification process [27], [37], [60]. To select optimal traditional ML models, we applied Bayesian optimization [64] using hyperopt library [65]. We chose bayesian optimisation due to its robustness against noisy objective function evaluations [66]. We utilised the average Matthews Correction Coefficient (MCC) of 10-fold cross-validation with stratified sampling [37] and early stopping criteria to select the optimal hyper parameters. MCC was used to select the optimal model since MCC explicitly considers all classes and is proclaimed as the best metric for error consideration by the prior study [67]. Besides, we implemented our models using scikit-learn [68] and gensim [69].

After concept extraction from the corpus, we used the Breadth-First-Search (BFS) [43] algorithm to identify the configuration argument and then update the KGConfig. BFS is a traditional graph traversal algorithm and is also commonly used in prior knowledge graph research [70].

V. Results and Evaluation

Our KGSecConfig built upon Kubernetes, Docker, Azure, and VMWare consists of 5,172 arguments, 2,774 options, and 23,177 descriptions. We obtained 1,463 argument, 1,793 options, where 984 arguments are not secured by default, and 479 arguments are secured by default.

We conducted a series of experiments to evaluate the effectiveness and feasibility of our KGSecConfig by answering the following Research Questions (RQs).

- **RQ1**: *How effective is a keyword-based method for estimating relevancy of security documents to configuration?* Our keyword-based method considers configuration arguments with the exact syntax as mentioned in the software systems. The answer to RQ1 will inform whether the method returns the accurate configuration documents. Accurate configuration documents are important since they are the primary source of secure options and their implementation details.
- **RQ2:** *How effective is our concept classifier module based on machine-learning?* Our classifier module aims to classify sentences of the security configuration documents to identify concepts. The answer to RQ2 will reveal how accurately the module predicts in terms of classification. An accurate model is important since the predicted concepts will be further used to build KGSecConfig on top of KGConfig by adding concepts. Besides, it will help organizations to select suitable models in their orchestration service in a software-defined network.
- **RQ3:** *What is the intrinsic quality of the knowledge captured in KGSecConfig?* Our KGSecConfig aims to capture, link, and correlate the configuration knowledge of multi-vendor software systems from diverse data sources, which can be further utilized for securing the CO configuration automatically. The answer to RQ3 will reveal the quality, i.e., whether the captured knowledge precisely convey the completeness and correctness of the configuration knowledge. Ascertaining the completeness

TABLE II: Evaluation metrics

Metric	Description
Accuracy	Percentage of correctly classified instances.
Recall	The correctly identified proportion of positive instances.
Precision	The percentage of the detected positive instances that were correct.
f1-score	The harmonic mean of recall and precision.
Matthews correlation coefficient (MCC)	It is correlation coefficient that depicts the performance of the classifier by considering all four dimensions in the confusion metric.

and correctness is crucial for determine KGSecConfig's purpose for various downstream utilization.

A. Protocol for answering RQs

1) RQ1: We adopted a sampling method [58] similar to the prior studies [71], [72] to ensure that the ratios observed in the sample are generalized to the population within a specific confidence interval at a certain confidence level. The required sample size is 385 for a confidence interval of 5 at 95% confidence level. Therefore we randomly selected 385 security documents and identified how accurately the keyword-based method could select configuration-related documents.

2) RQ2: We used our manually curated dataset of 3,032 sentences (*'statement'* concept 790 sentences, *'goal'* concept 860 sentences, *'action'* concept 751 sentences, and *'other'* concept 631 sentences) to evaluate the ML-models. Besides, we used the evaluation metrics defined in Table II. 80% of the dataset was randomly selected for model selection and building, and 20% for model prediction on unseen sentences. To select the optimal hyper parameter for each model, we performed stratified 10-fold cross-validation. Stratified sampling ensures that the proportion of each source would be kept. We ran our approach (e.g., KGConfig construction, security configuration document estimation, and security configuration concept classification) with different ML-models five times to calculate the time required to build KGSecConfig. Moreover, one-tailed non-parametric Mann-Whitney U-test [73] and Cohen's d effect size [74] tests were calculated to compare the statistical significance of the observed samples. These tests were selected since those were not affected by the data distribution and also used in prior studies to confirm their observation on the selection of ML-models [37], [60].

3) RQ3: We used a statistical sampling method similar to RQ1 to examine the randomly sampled instances of entities in KGSecConfig. Our statistical sampling ensures the evaluation is in 0.05 error margin at 95% confidence level. We used accuracy as the evaluation metric for assessing the quality of KGSecConfig since accuracy is the most used and well-known evaluation metrics for the quality assessment of KG as reported in the prior researches [58], [59], [70], [71]. Two authors independently evaluated the accuracy and discussed to reach the consensus where independent assessment were different. We followed the prior research [58], [59], [71] criterion, that is whether an extracted instance is correct and meaningful (e.g., complete). We computed Cohen's Kappa [57] to measure the inter rater agreement. Our SeConfigKG is built upon KGConfig (Section III-A) by adding our three secured configuration concepts (Section III-C) as *'statement'*, *'goal'*, and *'action'* entities using learning models. As we already evaluated the learning models in RQ2, we did not repeat the evaluation of these entities in RQ3. We focused our quality assessment on other configuration entities, e.g., argument, option, types, default values, and description.

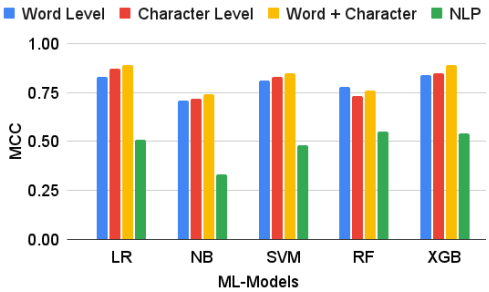


Fig. 8: MCC for each of the ML-models and features

TABLE III: Optimal hyper parameters for the studied ML-models

Model	Hyper Parameter	Model	Hyper Parameter	Model	Hyper Parameter
LR	C: 2.60 fit_intercept: True max_iter: 582 solver: liblinear tol: 9.95 e-05 warm_start: False scale: 0 normalize: 0	SVM	C: 8.68 gamma: 3.56 kernel: linear	RF	criterion: gini max_depth: 34 max_features: auto n_estimators: 100
NB	alpha: 0.52 fit_prior: False	XGB	max_depth: 92 tree_method: exact		

B. Evaluation Results

1) RQ1: We achieved an accuracy of 0.98 for our keyword-based method. Besides, we obtained 1.00, 0.95, and 0.97 precision, recall, and f1-score, respectively. Our approach could not identify the configuration documents where syntax were not preserved in the document, e.g., *'admissionControl'*, *'–azure-container-registry'*, or *'show.hidden.metrics'* violating camel case, hyphenation, or dotted syntax. However, relaxing the syntax preservation causes generation of huge false positives, e.g., providing irrelevant documents as configuration documents. For example, the sentences 'The azure container registry is Microsoft's own hosting platform for Docker images' [75], or 'How to Best Secure Azure Container Registry' [76] resulted in a configuration sentence. However, the documents are not providing any configuration information. Thus, the relaxation of syntax preservation may generate many irrelevant documents to process and large search space. In this regard, our keyword-based method with syntax preservation can keep the search space of configuration knowledge localization minimized, yet representative and provide high precision results for retrieving secured configuration documents.

2) RQ2: Fig. 8 shows a combination of word and character level feature performs better than other features in terms of MCC. For the character level feature, we chose n-gram in the range of $2 \leq n \leq 4$ since vocabulary size does not increase after character size 4. A combination of word and character level features considers both word and character to learn the semantic representation of the sentence to give better results. NLP features (e.g., count of word, character, noun, verb, adjective, adverb, and pronoun) perform similarly for all single models except ensemble models. One of the reasons could be that ensemble models learn the overlapping of NLP features among classes due to their intrinsic characteristics

TABLE IV: Accuracy achieved by the studied ML-models

Model	LR	NB	SVM	RF	XGB
Accuracy	0.94	0.82	0.88	0.76	0.93

of individual model aggregation. We verified NLP features perform statistically significant different than other features with Mann-Whitney U-test (z-score is 2.506, p-value is .006, significant at p-value $<$ .05) and large Cohen's d effect size (4.324). Table III represents the optimal hyper-parameter of each model for its best feature, e.g., a combination of word and character level features. Besides, Table IV presents the accuracy achieved by different ML-model. It is observed that LR and XGB outperform other models. We verified our observation with Mann-Whitney U-test (z-score is 3.741, p-value is $9 \times 10^{-}3$) and Cohen's d effect size. Table V shows the precision, recall, and f1-score obtained for each model. We found that LR and XGB also perform better than other models in terms of precision, recall, and f1-score.

Table VI shows the average time requirement to build KGSecConfig with different ML models. Our approach with the LR model takes the least amount of time (4.4 minutes on average) to build KGSecConfig. The difference among the time requirements is due to the different training times for ML models since the KGConfig construction, configuration documents estimation, and BFS algorithm execution required similar time to generate their respective output. XGB and RF take longer time to train than LR, NB, and SVM. We verified our observation with Mann-Whitney U-test, where we obtained statistical significant difference (z-score$<$3.185, p-value$<$9.1 $\times 10^{-}3$, significant at p-value $<$ 0.05) and larger Cohen's d effect size ($d >$174.1). XGB and RF are tree-based methods and traditionally reported to take longer time [77].

Calero and Pattini [78] mentioned current commercial designs are motivated by arguments based on sustainability (i.e., using fewer resources to achieve results). In particular, they asserted organizations used sustainability-based redesigns to motivate extensive cost-cutting opportunities. Therefore, we suggest LR should be preferred in building KGSecConfig.

3) RQ3: Table VII shows the results of our quality assessment of the extracted knowledge, where ACC-1, ACC-2, and ACC-F denotes the accuracy by two annotators independently, and final accuracy after resolving disagreement. The accuracies for all the configuration entities are above 90% and Cohen's Kappa are above 0.6, indicating substantial to almost-perfect agreement between annotators.

Our configuration entities argument and options achieved 100% accuracy, which is not surprising since such entities are extracted from structured document contents (e.g., enclosed in html, xml tags) with careful implementation and verification. Null values, empty fields, and empty strings were the common problems for default values and types extraction. For example, *'–allow-metric-labels'* argument in kube api server provided a square bracket notation to present empty fields in its default values, whereas *'–cgroup-root'* provided double quotes, *'–system-reserved'* provided backslash, and *'–kube-reserved-cgroup'* kept the space blank after mentioning types (e.g., '–kube-reserved-cgroup', 'Type:' <space>) in Kubernetes, which caused erroneous extraction. Besides, incomplete sentences due to erroneous splitting is the common problem for the description entity. For example, *'–container-runtime'* argument had the description 'The container, e.g., docker, remote runtime to use'. Our sentence tokenizer based on NLTK provided two separate sentences as 'The container, e.g.,', and 'docker, remote runtime to use' causing incomplete sentences for the description entity.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: todo-client-app-deploy
spec:
  replicas: 3
  selector:
    matchLabels:
      app: todo-client-app
  template:
    metadata:
      labels:
        app: todo-client-app
    spec:
      containers:
      - image: todo-client-kubernetes-app:latest
        name: container1
        imagePullPolicy: Always
```

Fig. 9: An example of Kubernetes manifest

Our KGSecConfig contains highly accurate configuration knowledge, which can support practical downstream applications, such as automated misconfiguration detection, verification, and mitigation. The common problems with the quality of extracted knowledge include text processing errors and meaningless description due to incomplete sentences. These problems can be minimized by developing more rules to enhance the text processing techniques, which can be further leveraged for accurate configuration entity extraction.

VI. Implication and Discussion

In this Section, we provide how KGSecConfig can be utilized for secured configuration of CO and some implications for researchers and practitioners.

A. *Secured Configuration-as-a-Service*

1) Automated mitigation of misconfiguration: Our KGSecConfig can be used to mitigate the misconfiguration of CO clusters automatically. Fig. 9 shows an example of a configuration file to deploy a containerized application [79]. We built a Kubernetes cluster with two nodes [80], a control plane [81], and a worker node [82] to explore the potential security threat actors with the configuration file as shown in Fig. 9. We leveraged our KGSecConfig to identify and mitigate the threat actors automatically, as shown in Fig. 10. A Compliance Parse Tree (CPT) as shown in Fig. 10(b) was built to tokenize configuration argument and its options, where the root of the tree is the configuration file type, e.g., *'Deployment'*, non-leaf nodes are arguments, and leaf nodes are options. We developed a compliance checker using subgraph matching [70] whose aim is to compare CPT with our KGSecConfig for possible misconfiguration detection.

Our KGSecConfig identified any user in the cluster could get access to the container as the security policy (e.g., *'SecurityContext'*) argument is missing. A user can also access the container as a root user since *'runAsNonRoot'* was not set. Moreover, a user can perform the privilege escalation

TABLE V: Precision, Recall, f1-score for each concept using the studied ML-models

	LR			NB			SVM			RF			XGB		
concept	Precision	Recall	f1-score	Precision	Recall	f1-score	Precision	Recall	f1-score	Precision	Recall	f1-score	Precision	Recall	f1-score
statement	0.98	0.93	0.96	0.94	0.77	0.85	0.93	0.93	0.93	0.97	0.82	0.89	0.97	0.9	0.93
goal	0.9	0.88	0.89	0.81	0.7	0.75	0.83	0.88	0.85	0.9	0.42	0.57	0.9	0.84	0.88
action	0.86	0.93	0.9	0.7	0.91	0.79	0.88	0.83	0.85	0.58	0.96	0.72	0.86	0.91	0.89
other	0.9	0.88	0.89	0.82	0.79	0.8	0.9	0.89	0.89	0.81	0.74	0.73	0.9	0.87	0.88

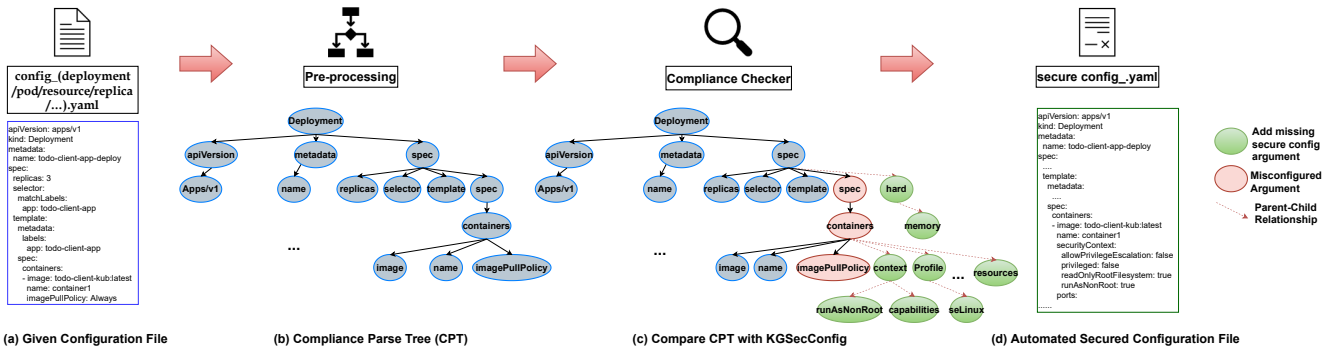


Fig. 10: Automated misconfiguration mitigation using KGSecConfig

TABLE VI: Time (s) required to build KGSecConfig by ML-models

Model	LR	NB	SVM	RF	XGB
Time (in seconds)	264	313	438	773	694

TABLE VII: Accuracy of knowledge in KGSecConfig

Entities	ACC-1	ACC-2	ACC-F	Agreement
argument	1.00	1.00	1.00	1.00
options	1.00	1.00	1.00	1.00
default values	0.97	0.98	0.98	0.62
type	0.96	0.94	0.94	0.77
description	0.90	0.91	0.91	0.86

as the *'allowPrivilegeEscalation'* argument is missing. Besides, DDoS attack can be mounted due to configuring *'imagePullPolicy'* as *'Always'* and undefined host networking policy. All of the mentioned threat actors are automatically generated from our compliance checker, as shown in the red circle in Fig. 10(c). In particular, our *'goal'* concept provided the knowledge of such threat actors where missing or undefined arguments were matched.

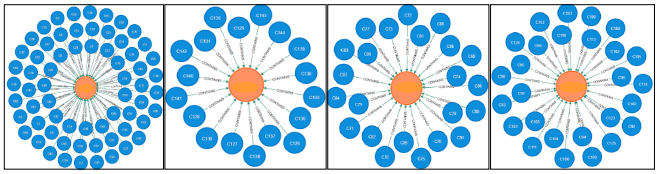
Our KGSecConfig also has the capability to automatically mitigate the misconfiguration by replacing (e.g., replacing *'Always'* option with *'IfNotPresent'* in *'imagePullPolicy'* argument) or adding (e.g., adding *'allowPrivilegeEscalation'* argument with *'false'* option) secured argument and corresponding options as a node or set of nodes as shown in green circles in Fig. 10(c) by using the knowledge in KGSecConfig. Finally, it can generate secured configuration files with YAML as shown in Fig. 10(d) or JSON format for automated execution due to the structured storage of relationships, e.g., parent-child relationship among arguments as shown in arrows in Fig. 10.

2) Automated Verification of Extracted Configuration: Our KGConfig (Section III-A) can serve as foundation support in terms of verification, e.g., whether the extracted configuration knowledge for security purposes either exists (e.g., if *'–authorization-mode'* argument is consistent with the current documentation or *'RBAC'* is a valid option for *'–authorization-mode'*) according to official documentation. For instance, our KGSecConfig has detected some inconsistencies in the NSA published report [83]. For instance, the NSA report stated Kubelet [84] and Kube-Scheduler [85] run Transmission Control Protocol (TCP) on port number 10251, however, we identified it is actually 10250 for read/write port and 10255 read-only port. Similarly, the same inconsistencies were found in other components of Kubernetes, such as Kubelet [84] and control-manager [86]. Our KGSecConfig can automatically detect inconsistencies without manual navigation of the large configuration space. In addition, we also detected some deprecated arguments (e.g., *'–repair-malformed-updates'* in kube api-server) mentioned in some white papers (e.g., RedHat). Thus our KGSecConfig ensures real-time monitoring and updated configuration knowledge aligned with official documentation by automatic verification of extracted knowledge in terms of configuration space.

B. *Automated Interpretation of Configuration Options*

Our KGSecConfig can provide the reasoning why we need to set/enable or disable a particular set of arguments that can increase the understanding of the configuration knowledge. For example, *'false'* option in *'–profiling'* argument reduces the potential attack surface since the default *'true'* option generates a significant amount of program data that could potentially be exploited to uncover system and program details. Our KGSecConfig systematically organizes such essential knowledge by *'hasGoal'* relationship with respective options which can help administrators' reasoning in configuring CO.

Besides, our KGSecConfig minimizes the manual traversal of multiple data sources required for an investigation to change the configuration options due to the disclosure of new security issues, e.g., vulnerabilities or malware. For example, a denial-of-service vulnerability of kube api-server, CVE-2019-11254 is revealed in one security information sharing-source (i.e., in a vulnerability database NVD [87]). This vulnerability can be fixed by restricting unauthorized access to kube api-server, which had been mentioned in another source, i.e., Security Announcement Kubernetes (SAK) [35]. However, neither NVD nor SAK described how to restrict unauthorized access to kube api server in terms of configuration options and arguments.



(a) Kubernetes (b) Azure (c) VMWare (d) Docker

Fig. 11: A subgraph visualizing the configuration hotspot of the studied CO software systems

Our KGSecConfig can correlate the encapsulated knowledge (e.g., *'goal'* concepts associated with arguments of kube api server and the query 'restrict unauthorized access to kube api-server') and return the mitigation approach by providing necessary arguments and options (e.g., *'–anonymous-auth'* will be *'false'* and *'–authorization-mode'* will be *'RBAC'*) for restricting the unauthorized access. Therefore, KGSecConfig can automatically provide the knowledge required for secured configuration to fix security issues. However, we acknowledge that the returned results will depend on the query formulation (e.g., query other than 'restrict unauthorized access to kube api-server'). In future research, we will perform the embedding-based textual similarity [24] on the encapsulated knowledge and the given query to reduce the semantic and lexical gap.

C. *Visualization of Configuration hot-spot*

Our KGSecConfig can support to visualize the configuration hot-spot, i.e., the arguments which need to change their default options for security purposes. The default configurations in CO and its integrated software are not always security-focused and require changes to harden the CO. Multiple cyber-security attacks were launched due to default configuration in Kubernetes. For instance, Tesla went through a cyber breach because of misconfiguration in 2018 [10]. Unauthorized users got access to secret resources exploiting an administrative CO which was being operated with default configuration in Tesla. Besides, the diversity of tools, platforms, and resource providers' configuration space makes it challenging to locate and then update the configuration manually. For example, the lead engineer of Target [88], a retail corporation, reviewed Kubernetes as "..no document versioning. Stuff is all over. It is difficult to find the right stuff.." [89]. In this regard, our KGSecConfig can be a potential approach to mitigate this challenge by visualizing a unified knowledge base. Fig. 11 represents a sub-graph (built using Neo4j [90]) providing the configuration hot-spot of the studied software systems. Fig. 11 can be a way to represent which software system needs to adjust default arguments more than the other software systems and can help the practitioner to prioritize their security configuration task. For instance, Fig. 11(a) has more more nodes compared to others (Fig. 11(b), 11(c), and 11(d)), representing it requires more adjustment of default arguments. Moreover, our KGSecConfig summarises relationships effectively (RQ1) and efficiently (RQ2 and RQ3), and it can scale to adopt new knowledge automatically using the learning-based model.

VII. Threats to Validity

1) Construct Validity: Labelled data are necessary for training a supervised learning model. Our manually labelled dataset may be biased and subjective. We followed the prior studies [30], [32], [42] to mitigate the bias. Two authors independently annotated the dataset and reported the disagreement. Moreover, we only used the dataset that both authors agreed to train the learning model for minimizing the effect of bias. Our labelled dataset is also comparable with prior studies to build the cyber security knowledge graph. For example, Islam et al. [42] labelled 1.7K, Rastogi et al. [30] labelled 3K, Piplai et al. [32] labelled 3.6K sentences for building security tools, predicting malware entities, and constructing malware KG, respectively.

2) Internal Validity: Our collected configuration options from official documentation might be incomplete. However, prior studies on configuration mining from question/answering sites [24], bug reports [25], [26] also leveraged official documentation in their studies. Besides, our security configuration arguments are derived from publicly available security information-sharing sources, which might miss some secured configuration arguments. However, we used security information sharing sources reported by prior researchers in the security of orchestrators [7] or software systems. In future work, we will investigate other sources of security information. Another threat to the internal validity is the subjective judgment in RQs, for example the evaluation of the quality of extracted configuration knowledge. To alleviate this threat, we have reported the agreement for each subjective judgment.

3) External Validity: Our evaluation results consider four case-study software systems and five ML models. We can not generalize our findings to other software systems or ML models. However, we provided an approach that can be applied to any other software system. Besides, our selected software system has a large configuration space and is popular in orchestration services [34]. Moreover, our goal is not to compare different ML models, and any learning-based model can be plugged into our approach to build the KG.

VIII. Conclusion and Future Work

85% of global organizations are forecast to relocate their legacy application to container-based development and deployment [8]. Container orchestrators, e.g., Kubernetes, are playing a vital role in managing container clusters in both the development and production environment. Container orchestrators require automated configuration for secured operation of container clusters. The diversity of software systems used in orchestrators, their vast configuration space, and information overload cause barriers for automating the secured configuration. We proposed a novel knowledge graph-based approach, KGSecConfig to aggregate and organize disparate silos of security configuration knowledge. We built a secured configuration knowledge graph with Kubernetes, Docker, Azure, and VMWare using KGSecConfig, which provide essential configuration knowledge, such as implementation details and reasoning. We demonstrated how KGSecConfig can be utilized for various downstream task, such as automated misconfiguration mitigation, inconsistency identification, and configuration

hot-spot visualization. We plan to extend our KGSecConfig to explore the potential of secure configuration migration from one orchestrator to another orchestrator for the clusters.

ACKNOWLEDGMENT

The work has been supported by the Cyber Security Research Centre Limited whose activities are partially funded by the Australian Government's Cooperative Research Centres Programme. This work has also been supported with supercomputing resources provided by the Phoenix HPC service at The University of Adelaide.

REFERENCES

[1] (2021) Kubernetes documentaiton. Access Date August, 2021. [Online]. Available: http://kubernetes.io

[2] (2021) Swarm mode overview. Access Date August, 2021. [Online]. Available: https://docs.docker.com/engine/swarm/

[3] (2021) Nomad documentation. Access Date August, 2021. [Online]. Available: https://www.nomadproject.io/docs

[4] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[5] B. Burns, J. Beda, and K. Hightower, *Kubernetes: up and running: dive into the future of infrastructure*. O'Reilly Media, 2019.

[6] Portworx and A. Security. (2019) 2019 container adoption survey. October 20, 2020. [Online]. Available: https://portworx.com/wp-content/uploads/2019/05/2019-container-adoption-survey.pdf

[7] M. S. I. Shamim, F. A. Bhuiyan, and A. Rahman, "Xi commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices," in *2020 IEEE Secure Development (SecDev)*. IEEE, 2020, pp. 58–64.

[8] StackRox. (2020) The state of container and kubernetes security. Access Date August, 2021. [Online]. Available: https://security.stackrox.com/rs/219-UEH-533/images/State_of_Container_and_Kubernetes_Report.pdf

[9] RedHat. (2021) State of kubernetes security report. Access Date August, 2021. [Online]. Available: https://www.redhat.com/en/engage/state-kubernetes-security-s-202106210910

[10] T. Taylor. (2020) 5 kubernetes security incidents and what we can learn from them. Access Date August, 2021. [Online]. Available: https://techgenix.com/5-kubernetes-security-incidents

[11] E. Zilberman. (2021) Top kubernetes configuration mistakes to avoid. Access Date August, 2021. [Online]. Available: https://www.datree.io/resources/kubernetes-configuration-mistakes

[12] R. Wilson. (2021) Why is kubernetes so hard - 4 reasons why and what to do about it. Access Date August, 2021. [Online]. Available: https://releasehub.com/blog/why-kubernetes-is-so-hard

[13] Kubernetes. (2021) Kubernetes deployments. Access Date August, 2021. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

[14] D. B. Bose, A. Rahman, and S. I. Shamim, "'under-reported'security defects in kubernetes manifests," in *2021 IEEE/ACM 2nd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS)*. IEEE, 2021, pp. 9–12.

[15] Kubernetes. (2021) kube api-server. Access Date August, 2021. [Online]. Available: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/

[16] C.-C. Chang, S.-R. Yang, E.-H. Yeh, P. Lin, and J.-Y. Jeng, "A kubernetes-based monitoring platform for dynamic cloud resource provisioning," in *GLOBECOM 2017-2017 IEEE Global Communications Conference*. IEEE, 2017, pp. 1–6.

[17] A. Modak, S. Chaudhary, P. Paygude, and S. Ldate, "Techniques to secure data on cloud: Docker swarm or kubernetes?" in *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)*. IEEE, 2018, pp. 7–12.

[18] (2021) Docker documentation. Access Date August, 2021. [Online]. Available: https://docs.docker.com

[19] (2021) Micorsoft azure documentaiton. Access Date August, 2021. [Online]. Available: https://docs.microsoft.com/en-us/azure/aks/

[20] (2021) Vmware tanzu kubernetes grid documentation. Access Date August, 2021. [Online]. Available: https://docs.vmware.com/en/VMware-Tanzu-Kubernetes-Grid/index.html

[21] (2021) Container concept kubernetes. Access Date August, 2021. [Online]. Available: https://kubernetes.io/docs/concepts/containers/

[22] (2021) Pods in kubernetes. Access Date August, 2021. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/

[23] J. Saldaña, *The coding manual for qualitative researchers*. sage, 2021.

[24] M. Sayagh and A. E. Hassan, "Configminer: Identifying the appropriate configuration options for config-related user questions by mining online forums," *IEEE Transactions on Software Engineering*, 2020.

[25] D. Jin, M. B. Cohen, X. Qu, and B. Robinson, "Preffinder: Getting the right preference in configurable software systems," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 151–162.

[26] W. Wen, T. Yu, and J. H. Hayes, "Colua: Automatically predicting configuration bug reports and extracting configuration options," in *2016 IEEE 27Th international symposium on software reliability engineering (ISSRE)*. IEEE, 2016, pp. 150–161.

[27] X. Xia, D. Lo, W. Qiu, X. Wang, and B. Zhou, "Automated configuration bug report prediction using text mining," in *2014 IEEE 38th Annual Computer Software and Applications Conference*. IEEE, 2014, pp. 107–116.

[28] D. van der Linden, E. Williams, J. Hallett, and A. Rashid, "The impact of surface features on choice of (in) secure answers by stackoverflow readers," *IEEE Transactions on Software Engineering*, no. 01, pp. 1–1, 2020.

[29] M. Verdi, A. Sami, J. Akhondali, F. Khomh, G. Uddin, and A. K. Motlagh, "An empirical study of c++ vulnerabilities in crowd-sourced code examples," *IEEE Transactions on Software Engineering*, 2020.

[30] N. Rastogi, S. Dutta, R. Christian, M. Zaki, A. Gittens, and C. Aggarwal, "Information prediction using knowledge graphs for contextual malware threat intelligence," *arXiv preprint arXiv:2102.05571*, 2021.

[31] A. Pingle, A. Piplai, S. Mittal, A. Joshi, J. Holt, and R. Zak, "Relext: Relation extraction using deep learning approaches for cybersecurity knowledge graph improvement," in *Proceedings of the 2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, 2019, pp. 879–886.

[32] A. Piplai, S. Mittal, A. Joshi, T. Finin, J. Holt, and R. Zak, "Creating cybersecurity knowledge graphs from malware after action reports," *IEEE Access*, vol. 8, pp. 211 691–211 703, 2020.

[33] O. Mendsaikhan, H. Hasegawa, Y. Yamaguchi, and H. Shimada, "Quantifying the significance and relevance of cyber-security text through textual similarity and cyber-security knowledge graph," *IEEE Access*, vol. 8, pp. 177 041–177 052, 2020.

[34] (2021) kubernetes blog. Access Date August, 2021. [Online]. Available: https://kubernetes.io/blog/

[35] (2021) kubernetes security announcement. Access Date August, 2021. [Online]. Available: https://groups.google.com/g/kubernetes-security-announce

[36] C. Dietrich, K. Krombholz, K. Borgolte, and T. Fiebig, "Investigating system operators' perspective on security misconfigurations," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1272–1289.

[37] T. H. M. Le, D. Hin, R. Croft, and M. A. Babar, "Puminer: Mining security posts from developer question and answer websites with pu learning," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 350–361.

[38] X. Zhao, Z. Xing, M. A. Kabir, N. Sawada, J. Li, and S.-W. Lin, "Hdskg: Harvesting domain specific knowledge graph from content of webpages," in *2017 ieee 24th international conference on software analysis, evolution and reengineering (saner)*. IEEE, 2017, pp. 56–67.

[39] S. Mumtaz, C. Rodriguez, B. Benatallah, M. Al-Banna, and S. Zamanirad, "Learning word representation for the cyber security vulnerability domain," in *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2020, pp. 1–8.

[40] A. Kao and S. R. Poteet, *Natural language processing and text mining*. Springer Science & Business Media, 2007.

[41] S. Vijayarani, M. J. Ilamathi, M. Nithya *et al.*, "Preprocessing techniques for text mining-an overview," *International Journal of Computer Science & Communication Networks*, vol. 5, no. 1, pp. 7–16, 2015.

[42] C. Islam, M. A. Babar, and S. Nepal, "Automated interpretation and integration of security tools using semantic knowledge," in *International Conference on Advanced Information Systems Engineering*. Springer, 2019, pp. 513–528.

[43] A. Bundy and L. Wallen, "Breadth-first search," in *Catalogue of artificial intelligence tools*. Springer, 1984, pp. 13–13.

[44] (2021) Kubernetes use case studies. Access Date August, 2021. [Online]. Available: https://kubernetes.io/case-studies/
[45] (2021) Beautiful soup documentation. Access Date August, 2021. [Online]. Available: https://www.crummy.com/software/BeautifulSoup/bs4/doc/
[46] (2021) Natural language toolkit. Access Date August, 2021. [Online]. Available: https://www.nltk.org/
[47] M. Liu, X. Peng, A. Marcus, Z. Xing, W. Xie, S. Xing, and Y. Liu, "Generating query-specific class api summaries," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 120–130.
[48] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred api knowledge," 2018.
[49] A. A. U. Rahman and L. Williams, "Software security in devops: Synthesizing practitioners' perceptions and practices," in *2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED)*. IEEE, 2016, pp. 70–76.
[50] A. A. U. Rahman, E. Helms, L. Williams, and C. Parnin, "Synthesizing continuous deployment practices used in software development," in *2015 Agile Conference*. IEEE, 2015, pp. 1–10.
[51] V. Garousi and B. Küçük, "Smells in software test code: A survey of knowledge in industry and academia," *Journal of systems and software*, vol. 138, pp. 52–81, 2018.
[52] (2021) sysdig. Access Date August, 2021. [Online]. Available: https://sysdig.com/
[53] C. Islam, M. A. Babar, and S. Nepal, "A multi-vocal review of security orchestration," *ACM Computing Surveys (CSUR)*, vol. 52, no. 2, pp. 1–45, 2019.
[54] (2021) Vulnerability assessment platform. Access Date August, 2021. [Online]. Available: https://vulners.com/
[55] V. V. Varenitca, A. S. Markov, and V. V. Savchenko, "Recommended practices for the analysis of web application vulnerabilities," in *10th Anniversary International Scientific and Technical Conference on Secure Information Technologies, BIT 2019 CEUR Workshop Proceedings*, vol. 2603, 2019, pp. 75–78.
[56] O. Valea and C. Oprişa, "Towards pentesting automation using the metasploit framework," in *2020 IEEE 16th International Conference on Intelligent Computer Communication and Processing (ICCP)*. IEEE, 2020, pp. 171–178.
[57] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.
[58] Y. Liu, M. Liu, X. Peng, C. Treude, Z. Xing, and X. Zhang, "Generating concept based api element comparison using a knowledge graph," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 834–845.
[59] J. Sun, Z. Xing, R. Chu, H. Bai, J. Wang, and X. Peng, "Know-how in programming tasks: From textual tutorials to task-oriented knowledge graph," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 257–268.
[60] T. H. M. Le, B. Sabir, and M. A. Babar, "Automated software vulnerability assessment with concept drift," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 371–382.
[61] T. Menzies, S. Majumder, N. Balaji, K. Brey, and W. Fu, "500+ times faster than deep learning:(a case study exploring faster methods for text mining stackoverflow)," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 2018, pp. 554–563.
[62] Y. Ma, S. Fakhoury, M. Christensen, V. Arnaoudova, W. Zogaan, and M. Mirakhorli, "Automatic classification of software artifacts in open-source applications," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 2018, pp. 414–425.
[63] (2021) Kaggle competitions. Access Date August, 2021. [Online]. Available: https://www.kaggle.com/
[64] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," *Advances in neural information processing systems*, vol. 25, 2012.
[65] J. Bergstra, D. Yamins, and D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *International conference on machine learning*. PMLR, 2013, pp. 115–123.
[66] Z. Wang, M. Zoghi, F. Hutter, D. Matheson, and N. De Freitas, "Bayesian optimization in high dimensions via random embeddings," in *Twenty-Third international joint conference on artificial intelligence*, 2013.
[67] A. Luque, A. Carrasco, A. Martín, and A. de las Heras, "The impact of class imbalance in classification performance metrics based on the binary confusion matrix," *Pattern Recognition*, vol. 91, pp. 216–231, 2019.
[68] (2021) scikit-learn-machine learning in python. Access Date August, 2021. [Online]. Available: https://scikit-learn.org/stable/
[69] (2021) Gensim. Access Date August, 2021. [Online]. Available: https://radimrehurek.com/gensim/
[70] X. Zou, "A survey on application of knowledge graph," in *Journal of Physics: Conference Series*, vol. 1487, no. 1. IOP Publishing, 2020, p. 012016.
[71] H. Li, S. Li, J. Sun, Z. Xing, X. Peng, M. Liu, and X. Zhao, "Improving api caveats accessibility by mining api caveats knowledge graph," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 183–193.
[72] C. Wang, X. Peng, M. Liu, Z. Xing, X. Bai, B. Xie, and T. Wang, "A learning-based approach for automatic construction of domain glossary from source code and documentation," in *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 97–108.
[73] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.
[74] J. Cohen, *Statistical power analysis for the behavioral sciences*. Academic press, 2013.
[75] T. Seals. (April 14, 2021) Security bug allows attackers to brick kubernetes clusters. Access Date August, 2021. [Online]. Available: https://vulners.com/threatpost/THREATPOST:E7B7ABC22A2369A38FEB70BA45964658
[76] C. Losh. (July 20, 2020) How to best secure your azure container registry. Access Date August, 2021. [Online]. Available: https://vulners.com/trendmicroblog/TRENDMICROBLOG:F404AF8BA3B3A80C21BEB37E66D561E0
[77] M. Ganaie, M. Hu *et al.*, "Ensemble deep learning: A review," *arXiv preprint arXiv:2104.02395*, 2021.
[78] C. Calero and M. Piattini, *Green in software engineering*. Springer, 2015, vol. 3.
[79] (2021) React, node js, and mongodb microservices-based application deployment on kubernetes. Access Date August, 2021. [Online]. Available: https://github.com/aamirpinger/todo-app-client-server-kubernetes
[80] (2021) Nodes in kubernetes. Access Date August, 2021. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/nodes/
[81] (2021) Control plane in kubernetes. Access Date August, 2021. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/control-plane-node-communication/
[82] (2021) Worker nodes in kubernetes. Access Date August, 2021. [Online]. Available: https://docs.oracle.com/en/operating-systems/olcne/1.1/orchestration/worker.html
[83] (2021) Cybersecurity and infrastructure security agency. Access Date August, 2021. [Online]. Available: https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/2716980/nsa-cisa-release-kubernetes-hardening-guidance/
[84] (2021) Kubelet in kubernetes. Access Date August, 2021. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/#kubelet
[85] (2021) Kube scheduler in kubernetes. Access Date August, 2021. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/#kube-scheduler
[86] (2021) Cloud control manager in kubernetes. Access Date August, 2021. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/cloud-controller/
[87] (2021) National vulnerability database. Access Date August, 2021. [Online]. Available: https://nvd.nist.gov/
[88] (2021) Target. Access Date August, 2021. [Online]. Available: https://www.target.com/
[89] M. Rajkarnikar. (2017) Overall satisfaction with kubernetes. Access Date August, 2021. [Online]. Available: https://www.trustradius.com/reviews/kubernetes-2017-04-11-12-36-25
[90] (2021) Graph database platform. Access Date August, 2021. [Online]. Available: https://neo4j.com/

LLMSecConfig: An LLM-Based Approach for Fixing Software Container Misconfigurations

Ziyang Ye*†, Triet Huynh Minh Le*† and M. Ali Babar*†
*CREST - The Centre for Research on Engineering Software Technologies, Adelaide, Australia
†School of Computer and Mathematical Sciences, The University of Adelaide, Adelaide, Australia
ziyang.ye@student.adelaide.edu.au, triet.h.le@adelaide.edu.au, ali.babar@adelaide.edu.au

arXiv:2502.02009v1 [cs.SE] 4 Feb 2025

***Abstract*—Security misconfigurations in Container Orchestrators (COs) can pose serious threats to software systems. While Static Analysis Tools (SATs) can effectively detect these security vulnerabilities, the industry currently lacks automated solutions capable of fixing these misconfigurations. The emergence of Large Language Models (LLMs), with their proven capabilities in code understanding and generation, presents an opportunity to address this limitation. This study introduces LLMSecConfig, an innovative framework that bridges this gap by combining SATs with LLMs. Our approach leverages advanced prompting techniques and Retrieval-Augmented Generation (RAG) to automatically repair security misconfigurations while preserving operational functionality. Evaluation of 1,000 real-world Kubernetes configurations achieved a 94% success rate while maintaining a low rate of introducing new misconfigurations.**

Our work makes a promising step towards automated container security management, reducing the manual effort required for configuration maintenance.

***Index Terms*—Container, Container Orchestrator, Configuration, Security, Large Language Models, Prompt Template, Retrieval-Augmented Generation**

I. Introduction

Container Orchestrators (COs) have become integral to modern cloud infrastructure, enabling scalable and efficient deployment of applications. However, their widespread adoption has also highlighted a critical security challenge: the prevalence of security misconfigurations. These misconfigurations are the leading cause of vulnerabilities in containerised environments, posing significant risks to system integrity and data confidentiality [1], [2]. While Static Analysis Tools (SATs) are proficient at detecting these vulnerabilities through pattern matching and rule-based analysis [3], there remains a substantial gap in automated remediation solutions. This gap necessitates manual intervention by security administrators, a process that is not only time-consuming and labour-intensive but also prone to human error, especially as the scale and complexity of deployments grow [4]–[6].

The manual process of fixing misconfigurations involves meticulous examination and correction of numerous configuration files, often requiring deep domain expertise and an in-depth understanding of interdependent settings. This complexity is exacerbated in dynamic environments where configurations frequently change due to scaling operations and Continuous Integration and Continuous Deployment (CI/CD) practices [7]. There are numerous changes in adding, removing, and modifying different services, along with a large number of corresponding configuration files in CI/CD pipelines. Consequently, organizations usually face significant operational bottlenecks and increased vulnerability exposure due to the lag between the detection and remediation of security issues (misconfigurations) in COs.

Advancements in Deep Learning and, recently, Large Language Models (LLMs) in Software Engineering offer a promising avenue to bridge this gap [8], [9]. LLMs have demonstrated remarkable capabilities in understanding and generating complex technical content [10]–[12], making them suitable candidates for automating the remediation of security misconfigurations. By leveraging LLMs in conjunction with Retrieval-Augmented Generation (RAG) techniques, it is possible to not only identify but also generate accurate and context-aware fixes for misconfigurations, thereby significantly reducing the manual effort and minimising the risk of introducing new vulnerabilities.

This study introduces LLMSecConfig, a novel framework designed to automate the repair of security misconfigurations in COs, ensuring that configurations adhere to security best practices while maintaining operational functionality.

This paper makes the following key contributions:

1) **Novel Framework for Automated Security Configuration Repair**: An innovative framework that integrates SATs with LLMs for automated security configuration repair in COs. Our approach uniquely combines LLMs with RAG techniques to generate precise fixes while maintaining operational functionality.
2) **Demonstrated Effectiveness with Real-world Evaluation**: Comprehensive evaluation using 1,000 real-world Kubernetes configurations, demonstrating superior performance with a 94.3% repair success rate and minimal error introduction (0.024). Our approach significantly outperforms baseline models, including GPT-4o-mini, which achieved only a 40.2% success rate.
3) **Open-source Dataset and Implementation**: To facilitate reproducibility and future research in container security, we provide our complete implementation, including the collected dataset of Kubernetes misconfiguration, source code of the framework, and evaluation scripts at https://figshare.com/s/2a9be8ccfbec9d8ba199.

```
apiVersion: v1
kind: Pod
metadata:
  name: <Pod name>
spec:
  containers:
  - name: <container name>
    image: <image>
    securityContext:
      allowPrivilegeEscalation: true
```

```
apiVersion: v1
kind: Pod
metadata:
  name: <Pod name>
spec:
  containers:
  - name: <container name>
    image: <image>
    securityContext:
      allowPrivilegeEscalation: false
```

Fig. 1: Comparison of misconfigured (left) versus properly configured (right) Kubernetes security contexts. The misconfigured configuration allows privilege escalation, enabling container processes to gain elevated privileges beyond their parent processes. In contrast, the secure configuration explicitly disables this capability.

II. Background and Motivation

A. Container Security Landscape

The rise of Infrastructure as Code (IaC) has transformed IT infrastructure management, enabling automated and scalable deployment of applications. Container technologies like Kubernetes offer significant benefits in terms of portability and resource efficiency, but they also present unique security challenges that span multiple layers of the system. These challenges range from kernel vulnerabilities to intricate orchestration misconfigurations [13]–[15].

Fig. 1 illustrates a common security misconfiguration in container orchestration, where incorrectly enabled privilege escalation could compromise system security. However, security fixes in container orchestrators are often complex and go beyond simple value correction. For instance, setting a Pod's *allowPrivilegeEscalation* to *false* becomes challenging when multiple security configurations interact across namespaces.

Users must analyse configuration dependencies and prevent privilege overrides by considering three main factors. Primarily, *CAP_SYS_ADMIN* will override *allowPrivilegeEscalation* to *true*, invalidating the previous setting. Consequently, any predefined *CAP_SYS_ADMIN* must be removed to ensure that *allowPrivilegeEscalation* remains *false*. This behaviour is notably different from the scenario shown in Fig. 1.

Additionally, security contexts in Kubernetes operate at different levels. *PodSecurityContext* manages pod-level security and shared container settings, while container-level *SecurityContext* controls individual container security and takes precedence when both specify the same field. This complexity is further compounded by varying behaviours between Linux and Windows containers.

Moreover, security configuration complexity scales significantly with the architectural hierarchy of Kubernetes clusters. At the highest level, a cluster encompasses multiple nodes. Each node, in turn, hosts numerous Pods, while individual Pods can run multiple applications. Throughout this hierarchy, components maintain their own security settings and network policies, creating an intricate web of security configurations.

This hierarchical structure makes manually defining and fixing rules extremely challenging. More importantly, incomplete fixes can introduce new issues, such as incompatible values between interdependent configurations. Therefore, addressing these intricate security challenges requires automated and systematic approaches rather than manual intervention.

Modern COs provide an extensive array of configuration options to support diverse deployment scenarios and security requirements. While this flexibility is beneficial, it significantly increases the complexity of maintaining secure settings across all components [16]. The dynamic nature of container environments, characterised by frequent updates and scaling operations, further complicates the security landscape by potentially introducing vulnerabilities through configuration changes [1].

Multi-tenancy environments present additional security challenges, where shared resources and namespaces require careful management to prevent unauthorised access and resource contention [15]. Supply chain security is equally critical, as container images and their dependencies can introduce vulnerabilities that demand continuous monitoring [14]. The rapid evolution of security threats, including side-channel attacks and container-specific exploits, necessitates robust and adaptable security measures [15], [17].

B. The Need for Automated Fixing of Software Container Misconfigurations

The complexity of container security configurations, combined with the rapid pace of modern development practices, creates a compelling need for automated remediation solutions. In large-scale deployments, the number of configuration files can reach into thousands, making manual corrections time-consuming and prone to errors [4]–[6]. Security administrators must address each misconfiguration meticulously to ensure consistency and compliance across the entire infrastructure. As deployments grow, the operational burden and response times increase, rendering manual approaches impractical.

Currently, it is challenging to address misconfigurations in COs like Kubernetes. Firstly, understanding the intricate relationships between different settings and ensuring that fixes do not conflict with existing configurations requires significant expertise and domain knowledge [18]. This complexity makes effective manual remediation challenging, especially in environments lacking specialised security personnel.

The rapid pace of modern development practices, such as CI/CD, also demands swift and reliable configuration updates to keep up with agile development cycles and frequent deployments. Manual remediation cannot meet the speed and effi-

ciency required by these practices, leading to security risks and deployment delays [4]–[6]. Furthermore, regulatory standards often mandate stringent security configurations and require detailed audit trails of configuration changes. Automated fixing can ensure consistent adherence to compliance requirements and provide comprehensive logs for auditing purposes [19].

Despite the clear need for automated remediation solutions, current research and industry practices have not adequately addressed the automation of security misconfiguration fixes in container environments. While significant progress has been made in automated detection through SATs and validation frameworks, the critical step of actually implementing corrections remains largely manual [3]. Existing approaches primarily focus on providing guidance and recommendations, leaving the complex task of modification and validation to human operators [20]–[23]. This limitation is further exacerbated by the heterogeneous nature of COs and SATs; their diverse design philosophies, programming languages, and technical implementations make it challenging to develop a universal framework for automated detection and repair. Consequently, this gap between detection and remediation represents a significant bottleneck in container security management, particularly as organizations scale their deployments. Our work addresses this fundamental limitation by proposing one of the first automated solutions for fixing container security misconfigurations.

III. LLMSecConfig: An Automated Framework for Fixing Software Container Misconfigurations

A. Framework Overview

LLMSecConfig presents an end-to-end pipeline designed to detect and rectify misconfigurations in COs by synergizing SATs with LLMs. The framework operates through three interconnected phases: (*i*) SAT Integration and (*ii*) Context Retrieval for detecting and understanding misconfigurations, and then (*iii*) Repair Generation and Validation, as illustrated in Fig. 2. This modular architecture ensures flexibility and scalability, allowing each component to function independently while contributing to the overall system's effectiveness.

B. Static Analysis Tools Integration for Misconfiguration Detection

SATs are instrumental in the identification of security vulnerabilities and misconfigurations within software systems [21], [24]. Among the various SATs available for COs, such as Checkov, Trivy, Terrascan, and KubeSec [25]–[28], our framework selects Checkov as the primary tool due to its strategic advantages. Checkov offers a comprehensive collection of over 1,000 built-in security policies, ensuring extensive coverage of potential vulnerabilities [25]. Its implementation in pure Python facilitates seamless integration with our framework, allowing direct access to policy definitions and enhancing flexibility. Additionally, Checkov provides rich contextual information through integration with Prisma Cloud documentation and receives regular updates that incorporate the latest security best practices and emerging vulnerability patterns [29].

C. Context Retrieval for Misconfiguration Understanding

The context retrieval module is a critical component of LLMSecConfig, where RAG, with its efficient retrieval capability, has the characteristics of knowledge augmentation and the performance of context-aware generation [30], which enables us to enhance the repair generation process. This module integrates corrective feedback mechanisms to elevate the quality and reliability of the fixes produced by the LLMs [31]. For every detected misconfiguration, the system aggregates three essential types of information to provide a comprehensive context for repair.

Firstly, the SAT output serves as the foundational context, offering detailed error messages from Checkov scans, specific policy violation identifiers (e.g., CKV_K8S_20), resource type and location information, links to relevant documentation, and severity levels within the framework context, as illustrated in Fig. 3. This information provides a precise understanding of the nature and location of each misconfiguration.

Secondly, the policy implementation source code, exemplified in Fig. 4, offers in-depth insights into the validation logic and conditions underpinning each security policy, including the complete Python implementation of security checks, key configuration parameters being evaluated, and expected values. Access to the source code allows the framework to comprehend the specific criteria used to determine compliance, enabling more accurate and context-aware repairs. For each ready-to-fix configuration, we first scan it with the Checkov tool to identify security issues and obtain detailed information. Each security issue has a unique ID in Checkov, and by examining its GitHub repository [25], we can locate the corresponding Python file for that ID, which includes the implementation of the security-related checking policy. To facilitate fast indexing, we created a CSV file to map each ID to its source code file.

Thirdly, comprehensive security documentation from Prisma Cloud, depicted in Fig. 5, supplements the context with detailed analysis of security implications, specific fix recommendations, example configurations demonstrating correct implementations, and framework-specific guidance and best practices. This documentation provides additional layers of information that guide the LLMs in generating precise and effective fixes.

The context module efficiently retrieves and maintains the relationships between these information sources by extracting policy identifiers and error details from Checkov's output, locating the corresponding Python implementations, and fetching the associated Prisma Cloud documentation. The merge of these elements ensures that the LLMs can get all the necessary information to generate accurate and contextually appropriate repairs. More details about the prompt templates can be found in the Appendix at https://figshare.com/s/2a9be8ccfbec9d8ba199.

D. Misconfiguration Repair Generation and Validation

The repair generation component leverages LLMs to produce context-aware fixes for the identified misconfigurations.

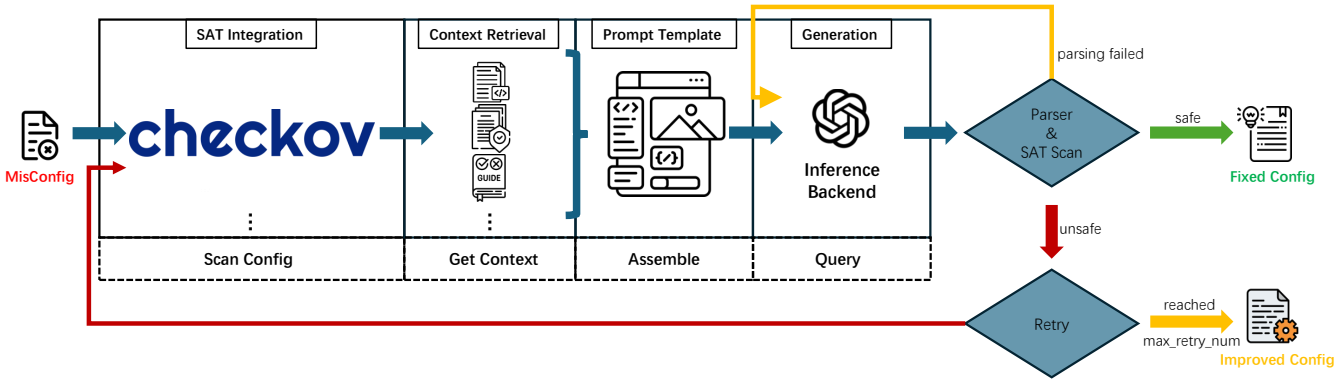


Fig. 2: Architecture overview of the LLMSecConfig framework for automated Kubernetes security configuration repair. The pipeline comprises three main stages: (1) configuration analysis using SATs, (2) context retrieval through RAG, and (3 & 4) repair generation using Large Language Models (LLMs). The system iteratively processes configurations until all security issues are resolved or the maximum number of attempts is reached.

```
$ checkov -f your_input.yaml --compact --framework
     kubernetes --quiet -c CKV_K8S_20
kubernetes scan results:
Passed checks: 0, Failed checks: 1, Skipped checks: 0
Check: CKV_K8S_20: "Containers should not run with
     allowPrivilegeEscalation"
FAILED for resource: Pod.default.{pod-name}
File: /your_input.yaml:1-10
Guide: https://docs.prismacloud.io/en/enterprise-edition/
     policy-reference/kubernetes-policies/kubernetes-policy-
     index/bc-k8s-19
```

Fig. 3: Example output from the Checkov security scanner, indicating a detected privilege escalation vulnerability in a Kubernetes configuration.

```
from typing import Dict, Any
from checkov.common.models.enums import CheckResult
from checkov.kubernetes.checks.resource.
     base_container_check \
    import BaseK8sContainerCheck
class AllowPrivilegeEscalation(BaseK8sContainerCheck):
    def init(self) -> None:
        # CIS-1.3 1.7.5
        # CIS-1.5 5.2.5
        name = "Containers should not run with
             allowPrivilegeEscalation"
        id = "CKV_K8S_20"
        super().init(name=name, id=id)
    def scan_container_conf(self, metadata: Dict[str, Any],
          conf: Dict[str, Any]) -> CheckResult:
        self.evaluated_container_keys = [
            "securityContext/allowPrivilegeEscalation"
        ]
        if isinstance(conf, dict) and conf.get("
             securityContext"):
            if conf["securityContext"].get(
                "allowPrivilegeEscalation"
            ) is False:
                return CheckResult.PASSED
        return CheckResult.FAILED
check = AllowPrivilegeEscalation()
```

Fig. 4: Python implementation of the Checkov security check for container privilege escalation.

Policy Details

Prisma Cloud Policy ID	3aa8f043-3853-4c9e-ae3a-8d3a70d69d4b
Checkov ID	CKV_K8S_20
Severity	MEDIUM
Subtype	Build
Frameworks	Kubernetes, Terraform, Helm, Kustomize

Description

The **AllowPrivilegeEscalation** Pod Security Policy controls whether or not a user is allowed to set the security context of a container to **True**. Setting it to **False** ensures that no child process of a container can gain more privileges than its parent. We recommend you to set **AllowPrivilegeEscalation** to **False**, to ensure **RunAsUser** commands cannot bypass their existing sets of permissions.

Fix - Buildtime

Kubernetes

- **Resource:** Container
- **Arguments:** allowPrivilegeEscalation (Optional) If false, the pod can not request to allow privilege escalation. Default to true.

```
apiVersion: v1
kind: Pod
metadata:
  name: <Pod name>
spec:
  containers:
  - name: <container name>
    image: <image>
    securityContext:
+     allowPrivilegeEscalation: false
```

Fig. 5: Prisma Cloud security policy documentation for CKV_K8S_20: **Containers Run with AllowPrivilegeEscalation**.

This process involves carefully designed prompt templates that incorporate error information, policy implementation details, and security documentation. By providing the LLM with this rich context, we enable it to generate precise and effective

repairs. The system supports multiple LLM backends, offering flexibility in model selection based on specific requirements.

Upon receiving the context, the LLM generates proposed configuration changes to resolve the identified security issues. To ensure that these generated fixes are valid and effective, LLMSecConfig incorporates a validation process. This process begins with using SAT to verify valid syntax, ensuring that the generated output maintains a valid YAML structure. If the syntax is valid, the system proceeds to security validation, confirming that the repairs address the identified vulnerabilities without introducing new issues.

To enhance system robustness, we implement two stages of retry mechanisms. An outer loop manages the overall repair attempts, limiting the total number of fixes attempted for each configuration to prevent endless loops. An inner loop handles potential parsing failures of LLM outputs, ensuring that only valid YAML configurations proceed to the security validation stage.

During each repair iteration, the system performs SAT analysis to identify any remaining security issues. If issues persist, the context is updated, and the LLM generates a new fix. To conform to specific SAT's security best practices, this iterative process continues until all issues are resolved, the maximum number of repair attempts is reached, or an unrecoverable error is encountered.

Additionally, LLMSecConfig maintains comprehensive control mechanisms and detailed logging. Each repair attempt is logged with information about the original configuration state, identified security issues, generated fixes, and validation results.

By combining automated repair generation with stringent validation protocols, LLMSecConfig provides a reliable approach to addressing security misconfigurations in COs. The integration of SATs and LLMs, coupled with effective context management and validation, ensures that the framework can effectively repair misconfigurations while preserving the operational functionality of the containerised applications.

IV. Experimental Setup and Design

A. Kubernetes as a Case Study Container Orchestrator

Kubernetes was selected as our experimental platform due to its position as the de facto standard for container orchestration, with over 92% of organizations either using or evaluating it according to the CNCF survey [32], [33]. Its extensive security-related settings and well-documented configuration patterns provide an ideal testing ground for our automated repair approach, making our research findings immediately applicable to a large user base [34].

B. Dataset

The need for a custom dataset stems from several key factors. First, there is an absence of public datasets specifically designed for container misconfiguration remediation [35], [36]. Moreover, even if such datasets existed, they would require constant updates due to the evolving nature of COs and their security configurations [37]. As API specifications change, security configuration requirements must adapt accordingly.

We selected ArtifactHub as our primary data source due to its repository of high-quality, vetted Kubernetes configurations that accurately represent real-world usage patterns. As the official CNCF artifact hub [38], [39], ArtifactHub provides a standardised and open format, facilitating straightforward data collection and validation processes. ArtifactHub ensures that the configurations are well-maintained and regularly updated, capturing current configuration trends and security challenges.

The data collection process was executed through a systematic multi-step approach. Initially, we accessed ArtifactHub's API to identify the top 1,000 most popular projects based on their internal ranking system, ensuring that our dataset encompassed configurations widely used and representative of common industry practices. Each selected project, managed using Helm, required conversion of Helm charts into raw Kubernetes configuration files. This step was crucial as Helm charts typically generate multiple YAML files, each representing distinct Kubernetes resources or components.

Subsequently, the exported YAML files were parsed to isolate individual sub-configurations. Each sub-configuration was treated as a separate entity corresponding to a specific Kubernetes resource, such as Pods, Services, or Deployments. These isolated configurations were then subjected to analysis using SATs to identify security misconfigurations. Only those files that triggered security issues were retained to increase the vulnerability label quality [40], [41], culminating in a robust test dataset of 1,000 configuration files with detected misconfigurations. The distribution of different types of security misconfigurations within this dataset is illustrated in Fig. 6, highlighting the prevalence of various vulnerability patterns in real-world container configurations.

C. LLMs Selection for Repair Generation

Our evaluation utilised two distinct LLMs: Mistral Large 2 and GPT-4o-mini [42], [43], strategically selected to compare both open-source and closed-source approaches while considering practical deployment factors.

Mistral Large 2 was chosen as our primary open-source model not only for its strong performance characteristics but also for its exceptional cost-effectiveness. Its open-source nature enables complete transparency in deployment and modification, allowing organizations to implement our framework without dependency on external API services. The model's efficient architecture makes it particularly suitable for production deployment, offering high performance at significantly lower computational costs compared to other open-source alternatives [42].

GPT-4o-mini was selected as our closed-source baseline for several practical reasons. As OpenAI's most cost-effective model [43], it offers a good balance between performance and operational costs. Its widespread adoption and extensive usage history in the developer community provide valuable benchmarking opportunities. The model's optimisation for speed makes it particularly suitable for rapid configuration analysis

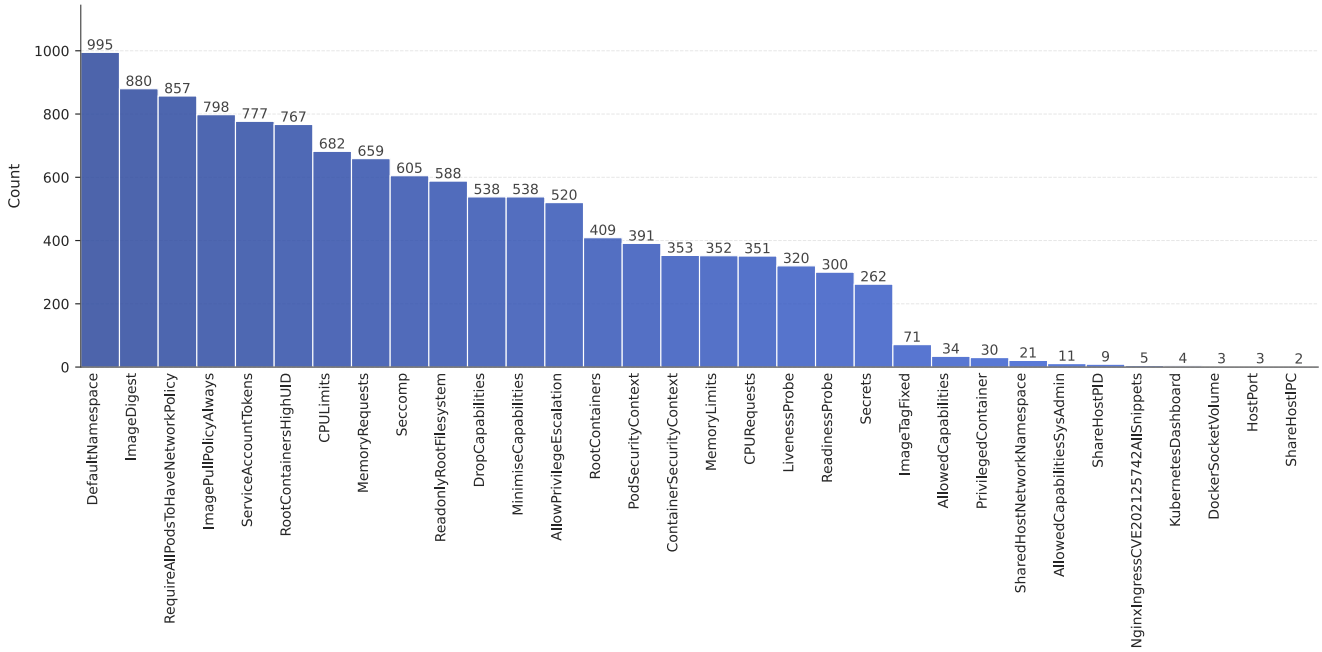


Fig. 6: Distribution of security misconfiguration types in our dataset collected from ArtifactHub. The chart illustrates the relative frequency of different security issues detected by SATs across 1,000 configuration files, highlighting the prevalence of specific vulnerability patterns in real-world container configurations.

and repair tasks, while still maintaining competitive accuracy levels. Although newer models are available, GPT-4o-mini's established track record and substantial community feedback make it reliable for evaluating our framework's performance with commercial solutions.

D. Evaluation Metrics

We adopted two primary evaluation metrics, i.e., the **Parse Success Rate (PSR)** and the **Pass Rate (PR)**. The PSR measures the ability of LLMs to generate syntactically valid YAML configurations, while the PR indicates the proportion of repaired YAML configurations that successfully pass all security checks without errors.

Let $\mathcal{D} = \{d_1, \ldots, d_N\}$ denotes our test dataset, consisting of N configurations.

$$\text{Parse Success Rate} = \frac{1}{N}\sum_{i=1}^{N} \mathbb{I}_{\text{parse}}(d_i) \times 100\% \tag{1}$$

$$\text{Pass Rate} = \frac{1}{N}\sum_{i=1}^{N} \mathbb{I}_{\text{pass}}(d_i) \times 100\% \tag{2}$$

where $\mathbb{I}_{\text{parse}}(d_i)$ and $\mathbb{I}_{\text{pass}}(d_i)$ are indicator functions that equal 1 if the YAML parsing of configuration d_i succeeds and if configuration d_i passes all security checks, respectively, and 0 otherwise. Higher values are preferred for both metrics.

To evaluate repair efficiency, we measured the **Average Pass Steps (APS)**:

$$\text{Average Pass Steps} = \frac{\sum_{i=1}^{N} s(d_i) \cdot \mathbb{I}_{\text{pass}}(d_i)}{\sum_{i=1}^{N} \mathbb{I}_{\text{pass}}(d_i)} \tag{3}$$

where $s(d_i)$ is the number of steps required for configuration d_i to pass all security checks. Lower values are preferred.

For step-wise analysis, we computed the **Area Under Curve (AUC)** metrics for both **Pass Rate over Steps** (AUC_{PRS}) and **Average Pass Steps over Steps** (AUC_{APSS}):

$$\text{AUC}_{\text{PRS}} = \frac{1}{T}\sum_{t=1}^{T} \text{PR}_t \tag{4}$$

$$\text{AUC}_{\text{APSS}} = \frac{1}{T}\sum_{t=1}^{T} \text{APS}_t \tag{5}$$

where T is the maximum number of allowed repair steps, PR_t is the pass rate achieved within t steps, and APS_t is the average number of steps needed for passing configurations when limited to t steps. Higher values are preferred for AUC_{PRS}, while lower values are preferred for AUC_{APSS}.

Finally, we assessed repair quality through the **Security Improvement** and **Average Introduced Errors** metrics:

$$\text{Security Improvement} = \frac{\sum_{i=1}^{N} (e_{\text{init}}(d_i) - e_{\text{final}}(d_i))}{\sum_{i=1}^{N} e_{\text{init}}(d_i)} \tag{6}$$

where $e_{\text{init}}(d_i)$ represents the number of security issues in configuration d_i before repair, and $e_{\text{final}}(d_i)$ represents the

number of security issues in configuration d_i after repair. Higher values are preferred.

$$\text{Average Introduced Errors} = \frac{1}{N}\sum_{i=1}^{N} e_{\text{new}}(d_i) \qquad (7)$$

where $e_{\text{new}}(d_i)$ represents the number of new security issues introduced during the repair of configuration d_i. Lower values are preferred.

V. Research Questions and Experimental Results

A. *RQ1: To what extent are LLMs effective to correct misconfigurations?*

Motivation: Our primary research question aims to evaluate the fundamental capability of LLMs in automating configuration error correction. While SATs excel at detecting misconfigurations, their inability to automatically correct them creates a significant operational gap. This question quantitatively assesses whether LLMs can effectively bridge this gap by examining several key aspects. We aim to evaluate correction accuracy rates across different LLMs and analyse the syntax validity of the generated corrections.

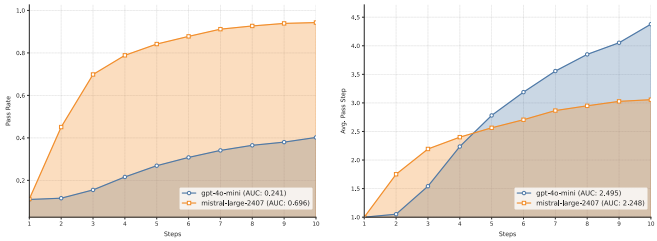
Method: We conducted the experiments using the GPT-4o-mini [43] and mistral-large-2407 [42] models on a dataset of 1,000 Kubernetes configurations as described in Section IV-B. For both models, we set the temperature to 0.5 and used the default values for all other hyperparameters. Additionally, we configured the maximum parser retry time and the maximum retry value to 10 and 5, respectively. The key parameters in our framework, namely *temperature* and *max_retry*, were selected as follows:

- ***temperature***: This parameter controls the randomness of the model's output. A value of 0 results in deterministic outputs, while higher values increase variability. We set the temperature to 0.5, as it provides a balance between diversity and stability, based on findings from prior research [44].
- ***max_retry***: To account for the non-deterministic nature of LLMs, we introduced a retry mechanism. A non-zero *max_retry* value increases the likelihood of generating correct outputs, but excessive values can lead to higher computational costs. Through experimentation, we determined that a maximum parser retry time of 10 and a maximum retry value of 5 offer a practical balance between performance and efficiency.

For each sample, we logged the input file's Checkov output and the output file's scanning result, so we could obtain all the metrics we designed and the stopping step.

Results: The results, presented in Table I and visualised in Fig. 7, demonstrate both the potential and current limitations of LLM-based configuration repair.

GPT-4o-mini exhibited considerable limitations in its ability to correct misconfigurations effectively, achieving only a 40.2% PR and 99.8% PSR. This relatively low success rate was accompanied by decreasing efficiency over time,



(a) Pass Rate over Steps (b) Average Pass Steps over Steps

Fig. 7: Step-wise performance metrics comparing GPT-4o-mini and Mistral Large 2 models. (a) Pass Rate over Steps illustrates the cumulative success rate in fixing misconfigurations across repair attempts. (b) Average Pass Steps over Steps indicates the efficiency of repair processes, where lower values represent more efficient repairs.

as indicated by an increasing APS that reached 4.38 steps. The model showed a concerning tendency to introduce new errors, with an error introduction rate of 0.029. In contrast, Mistral Large 2 demonstrated remarkably strong capabilities, achieving a 94.3% PR and 100% PSR. This high success rate was complemented by consistent efficiency, maintaining 3.06 APS, whilst demonstrating strong security improvement (0.986) with a low rate of introduced errors (0.024), suggesting its potential suitability for production environments.

The step-wise analysis can be better understood through the curves in Fig. 7. The Pass Rate over Steps curve in Fig. 7(a) shows the cumulative success rate as repair attempts progress, with the Area Under Curve (AUC_{PRS}) quantifying the overall repair effectiveness, a higher value indicates better overall success (Mistral: 0.696 vs GPT-4o-mini: 0.241). Conversely, Fig. 7(b) displays the Average Pass Steps required at each step limit, where the Area Under Curve (AUC_{APSS}) measures repair efficiency, lower values indicate more efficient repairs (Mistral: 2.249 vs GPT-4o-mini: 2.495), demonstrating Mistral Large 2's superior performance in both dimensions.

RQ1 Summary. LLMs are promising for automated configuration repair. Mistral Large 2 achieved superior performance (94.3% PR, 100% PSR, 3.06 APS) compared to GPT-4o-mini (40.2% PR, 99.8% PSR, 4.38 APS). Both models maintained low error rates (<0.03), demonstrating viability for production use.

B. *RQ2: What misconfiguration types are hard to correct?*

Motivation: Understanding the limitations of LLM-based correction is essential for both practical implementation and future research directions. We analyse patterns in failed correction attempts and identify categories of misconfigurations that consistently challenge LLM-based correction. By examining the relationships between configuration complexity and correction success, we can identify edge cases that require

LLM	PR(%)	PSR(%)	APS	AUC_{PRS}	AUC_{APSS}	Sec Imp	Avg. Introduced Err
GPT-4o-mini	40.2	99.8	4.38	0.241	2.495	0.906	0.029
Mistral Large 2	94.3	100	3.06	0.696	2.249	0.986	0.024

TABLE I: Comprehensive performance comparison of different Large Language Models (LLMs) in security configuration repair tasks. Metrics include Pass Rate (PR), Parse Success Rate (PSR), Average Pass Steps (APS), Area Under the Curve metrics (AUC_{PRS} and AUC_{APSS}), Security Improvement (Sec Imp), and Average Introduced Errors (Avg. Introduced Err).

additional mechanisms or human intervention. This analysis of challenging cases can provide valuable insights for future improvements in automated configuration correction systems and help establish realistic expectations for their capabilities.

Method: According to the intermediate and final results we logged in RQ1, after counting all errors' categories of input files and output files, we obtained and analysed each category's change (fixed or not for each category).

Results: Our analysis across different security policy types revealed distinct patterns in the difficulty of correcting various misconfigurations. As shown in Fig. 8, the success rates varied significantly depending on both the type of security policy and the model's capabilities.

Basic security configurations, particularly those involving resource limits and simple networking policies, consistently showed high success rates across all models. These configurations typically involve straightforward parameter adjustments and have clear, well-defined correction paths. The high success rates in these cases suggest that LLMs can effectively handle routine security configuration tasks that form the bulk of day-to-day security maintenance.

However, our analysis identified several categories of misconfigurations that proved consistently challenging to correct. Complex security contexts, particularly those involving privilege-related configurations, posed significant challenges for GPT-4o-mini, with Pass Rates falling below 50%. These cases often involve intricate interdependencies between different security parameters and require a deeper understanding of security implications.

While Mistral Large 2 showed better performance across all categories, it still encountered occasional difficulties with advanced security policies, particularly those involving complex network configurations. These configuration types usually have limited data in the wild (see Fig. 6), and such data imbalance has been shown to impede the performance of vulnerability prediction models [45]. We have also shown that even advanced models may require additional support mechanisms when handling such highly specialised security configurations.

RQ2 Summary. Basic security configurations showed consistently high PR across models. Complex security contexts, especially privilege-related configurations, proved challenging for GPT-4o-mini with <50% PR. While Mistral Large 2 demonstrated a better performance across different policy types, network configurations remained challenging for this model.

C. *RQ3: What context types are most useful for correction?*

Motivation: Given our framework's integration of RAG and prompting techniques, understanding the impact of different contextual information is crucial. We plan to explore the relative importance of various context sources in the correction process, including error messages from SATs, security documentation and best practices, and configuration file structure and metadata. This investigation helps establish optimal approaches for combining domain-specific knowledge with LLMs' capabilities in security-critical applications.

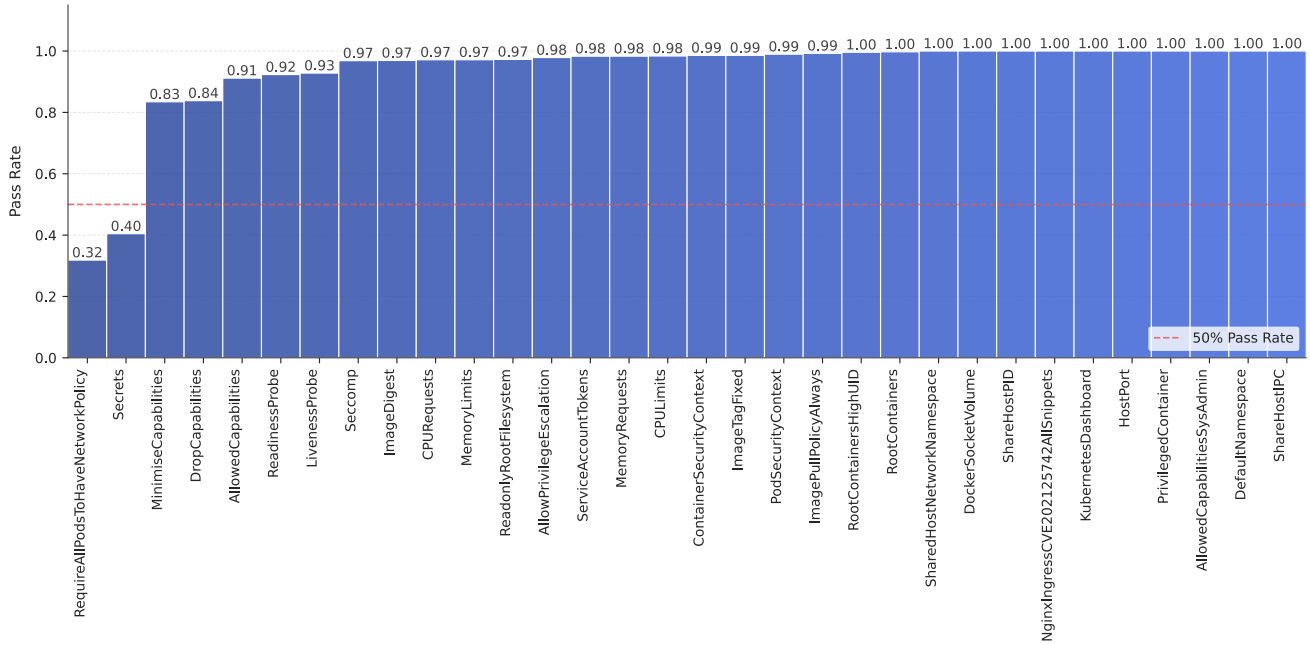
Method: Our ablation study was conducted using Mistral Large 2 due to its strong performance demonstrated in RQ1. While all parameters and settings remained identical with RQ1, we only varied the context retrieval method. There are three types of contexts in Checkov: scanning results, source code and Prisma documents from URL. The raw scanning results are mandatory for each run, so we tested four variants: scanning results only, scanning results with source code, scanning results with Prisma document content, and full context combining all three sources as used in RQ1.

Results: The results, detailed in Table II and visualised in Fig. 9, provide important insights into the relative value of different context types in the repair process.

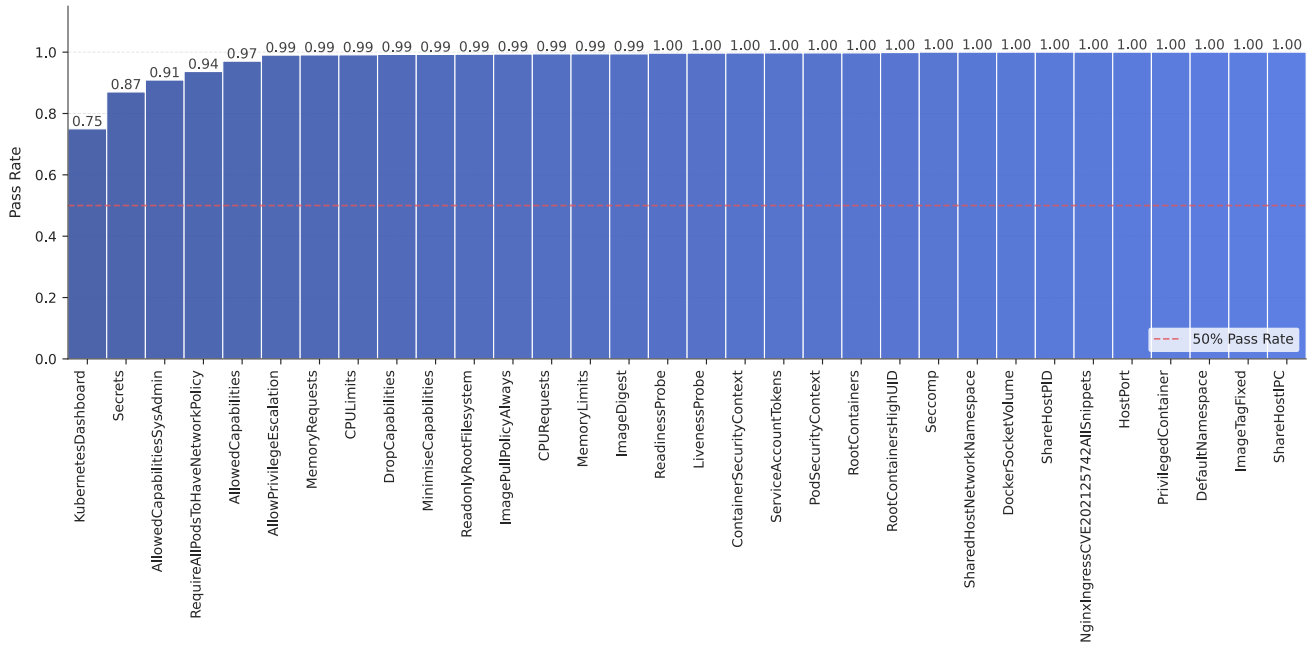
The base configuration, utilising only Checkov output (ckv_{out}), demonstrated balanced performance with a PSR of 95.9% and a PR of 88.0%. This relatively strong performance suggests that basic error messages provide essential information for many repairs without introducing excessive complexity. The AUC_{PRS} value of 0.598 indicates consistent performance across multiple repair attempts.

When enhanced with source code context (ckv_{out} + code), the system showed improvement in several metrics. The PR increased to 90.3%, while the AUC_{PRS} improved to 0.701, indicating both higher success rates and more consistent performance. Most notably, the APS decreased to 2.68, suggesting that source code context enables more efficient repairs. This improvement likely stems from the precise validation logic and expected values provided by the source code.

Interestingly, the addition of Prisma documentation alone (ckv_{out} + prisma) produced lower performance compared to other configurations. While achieving a high PSR at 98.4%, the overall PR dropped significantly to 65.2%, with reduced efficiency (APS: 4.29). However, when combined with both Checkov output, source code and Prisma documentation (full context as used in RQ1), the framework achieved its best overall performance refer to RQ1. This suggests that while documentation alone may introduce noise, its combination with precise technical context creates a complementary effect,



(a) GPT-4o-mini



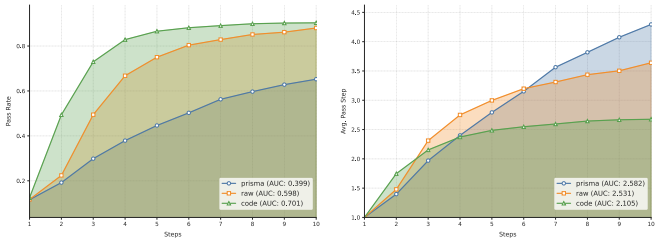
(b) Mistral Large 2

Fig. 8: Performance comparison of different Large Language Models (LLMs) (GPT-4o-mini and Mistral Large 2) in fixing Kubernetes security misconfigurations, categorised by policy type. The results demonstrate varying success rates across different security policy categories, with Mistral Large 2 consistently showing higher performance across most categories.

providing both the detailed technical requirements from source code and the broader security context from the documentation. The full context configuration achieved superior performance across all metrics, demonstrating that the combination of all three context types provides the most robust foundation for repair generation when properly integrated.

Context Type	PR(%)	PSR(%)	APS	AUC_{PRS}	AUC_{APSS}	Sec Imp	Avg. Introduced Err
ckv_out	88.0	95.9	3.64	0.598	2.531	0.9926	0
ckv_out + code	90.3	92.2	2.68	0.701	2.105	0.9946	0.0134
ckv_out + prisma	65.2	98.4	4.29	0.399	2.582	0.9675	0.0031
ckv_out + code + prisma (RQ1)	94.3	100	3.06	0.696	2.249	0.986	0.024

TABLE II: Comprehensive performance comparison of different context types in security configuration repair tasks. Metrics include Pass Rate (PR), Parse Success Rate (PSR), Average Pass Steps (APS), Area Under the Curve metrics (AUC_{PRS} and AUC_{APSS}), Security Improvement (Sec Imp), and Average Introduced Errors (Avg. Introduced Err).



(a) Pass Rate over Steps (b) Average Pass Steps over Steps

Fig. 9: Ablation study results comparing different context types in the repair process. The graphs display (a) Pass Rate progression and (b) Average Pass Steps across repair iterations for three context configurations: base configuration, source code-enhanced, and Prisma documentation-enhanced. Results demonstrate the superior performance of the source code-enhanced context in both repair success rate and efficiency.

RQ3 Summary. Source code context provided the most effective guidance (90.3% PR), while basic Checkov output alone achieved 88.0% PR. Adding Prisma documentation decreased performance (65.2% PR), indicating that precise, technical context outperforms comprehensive documentation for repairs.

VI. Discussion

A. LLMSecConfig and Beyond

Our experimental results reveal important insights about automated security configuration repair in container environments, while also highlighting areas for further development.

The significant performance gap between Mistral Large 2 and GPT-4o-mini demonstrates that advanced model architectures are crucial for processing complex security contexts with interdependent settings. This difference is particularly evident in the models' ability to maintain consistency across diverse policy types while preserving existing configuration intent.

Our ablation study reveals insights about context utilisation in repair generation for container misconfigurations. Source code context consistently provides the most effective guidance, achieving a 90.3% PR. While comprehensive, documentation can add noise that reduces repair efficiency. This suggests that future implementations should prioritise precise, structured context over verbose documentation, particularly for models with limited processing capacity.

Organizations adopting automated configuration repair should carefully consider their model selection strategy based on our experimental results. The demonstrated success of open-source models like Mistral Large 2, achieving a 94.3% pass rate while maintaining a low error introduction rate of 0.024, suggests their viability for production environments. However, the complexity of certain configuration types, particularly in advanced network policies, indicates that human oversight remains valuable for high-risk scenarios.

The strong performance of source code-based context retrieval, evidenced by its 90.3% PR compared to documentation-based approaches at 65.2%, emphasises the importance of maintaining well-structured policy implementations. Organizations should focus on establishing clear mappings between security checks and their underlying validation logic to maximise repair effectiveness. Furthermore, our successful implementation of multi-stage validation, achieving a 100% PSR while maintaining security compliance, demonstrates the necessity of comprehensive validation pipelines in production deployments. These findings collectively provide a practical foundation for implementing automated security configuration repair while highlighting specific areas that require careful consideration during deployment.

B. Threats to Validity

A primary internal validity threat lies in the potential suboptimal configuration of our framework and baseline models. While it is impractical to explore all, we mitigated this by evaluating multiple parameter settings and adapting best practices from relevant literature for LLM-based repair tasks. We also conducted extensive sensitivity analysis to understand the impact of key parameters on repair performance. We choose 0.5 as LLMs Temperature after experiments to find a balance between stabilisation and usability of retry mechanism.

For external validity, our findings may not generalise to all CO platforms and security scenarios as it was impractical to identify all security issues in the wild [46]. We addressed this threat by evaluating our approach on 1,000 real-world Kubernetes configurations from ArtifactHub, covering diverse project scales and security requirements. While focusing on Kubernetes, our dataset spans various deployment patterns and security policies commonly found in production environments.

VII. Related Work

A. Configuration Analysis and Management

Recent research has made significant progress in understanding and addressing container security challenges through

various methodological approaches. Static analysis techniques have been a commonly used approach, with studies by Smith et al. [21] and Johnson et al. [20] developing tools for analysing container configurations without execution. These approaches primarily focus on identifying potential security issues through pattern matching and semantic analysis, providing early detection of vulnerabilities in configuration specifications.

Complementing static analysis, dynamic analysis methods have been developed to provide runtime security monitoring and enforcement. Sultan et al. [15] proposed a comprehensive framework for detecting and preventing security violations during container execution, enabling real-time protection against emerging threats. Their work demonstrates the importance of runtime monitoring in maintaining container security, particularly in dynamic deployment environments.

The application of Machine Learning (ML) techniques to configuration security represents a more recent development in the field. Studies by Bandari [47] and Farkouh et al. [48] have demonstrated the potential of ML approaches in identifying patterns within secure and vulnerable configurations. These data-driven approaches offer promising capabilities for automating security analysis, though they often require substantial training data and careful feature engineering.

KGSecConfig [18] introduced a structured approach to configuration security, which leverages knowledge graphs and ontologies to represent and reason about security configurations. This semantic approach enables a more sophisticated analysis of configuration relationships and dependencies, though it requires careful knowledge of engineering and maintenance.

While these approaches have advanced our understanding of container security analysis, they primarily focus on detection rather than automated repair. Our work extends beyond detection by introducing an end-to-end framework that not only identifies misconfigurations but also automatically generates and validates fixes, addressing a critical gap in existing container security management solutions.

B. Source Code Vulnerability Detection and Fixing

The domain of automated vulnerability detection and repair in source code provides valuable insights for configuration security management. Traditional approaches have relied heavily on SATs and pattern matching. For example, Fabian et al. [49] introduced a code property graph for vulnerability detection, combining abstract syntax trees, control flow graphs, and program dependence graphs to identify complex vulnerability patterns. Building on this foundation, there have been major advances in AI (Machine Learning, Deep Learning, and recently LLMs) for detecting and addressing vulnerabilities in source code [50]–[59]. These prior studies have established key principles about context utilisation and fix validation that parallel our approach to configuration repair.

While code and configuration vulnerabilities present distinct challenges, the underlying principles of combining traditional analysis tools with AI capabilities remain valuable. Our work adapts these lessons to the specific requirements of container configurations, addressing unique challenges such as security parameter interdependencies and the need to maintain operational stability while enhancing security posture.

VIII. Conclusion

This paper presents a novel framework that combines SATs with LLMs for automated security configuration repair in COs. The framework's success (≈94% repair accuracy), particularly with context-aware repair generation, shows the potential of integrating traditional static analysis with modern AI techniques. This work represents a significant step toward automated, secure container deployment management.

IX. Data Availability

The data and code of this study are available at https://figshare.com/s/2a9be8ccfbec9d8ba199.

References

[1] E. Casalicchio, "Container orchestration: A survey," *Systems Modeling: Methodologies and Tools*, pp. 221–235, 2019.

[2] M. S. I. Shamim, F. A. Bhuiyan, and A. Rahman, "Xi commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices," *2020 IEEE Secure Development (SecDev)*, pp. 58–64, 2020.

[3] M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav, "A survey of static analysis methods for identifying security vulnerabilities in software systems," *IBM systems journal*, vol. 46, no. 2, pp. 265–288, 2007.

[4] A. Zerouali, R. Opdebeeck, and C. De Roover, "Helm charts for kubernetes applications: Evolution, outdatedness and security risks," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023, pp. 523–533.

[5] D. B. Bose, A. Rahman, and S. I. Shamim, "'under-reported' security defects in kubernetes manifests," in *2021 IEEE/ACM 2nd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS)*, 2021, pp. 9–12.

[6] E. Russell and K. Dev, "Centralized defense: Logging and mitigation of kubernetes misconfigurations with open source tools," 2024. [Online]. Available: https://arxiv.org/abs/2408.03714

[7] A. K. Arani, T. H. M. Le, M. Zahedi, and M. A. Babar, "Systematic literature review on application of learning-based approaches in continuous integration," *IEEE Access*, 2024.

[8] T. H. M. Le, H. Chen, and M. A. Babar, "Deep learning for source code modeling and generation: Models, applications, and challenges," *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–38, 2020.

[9] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–79, 2024.

[10] M. Bhatt, S. Chennabasappa, C. Nikolaidis, S. Wan, I. Evtimov, D. Gabi, D. Song, F. Ahmad, C. Aschermann, L. Fontana, S. Frolov, R. P. Giri, D. Kapil, Y. Kozyrakis, D. LeBlanc, J. Milazzo, A. Straumann, G. Synnaeve, V. Vontimitta, S. Whitman, and J. Saxe, "Purple llama cyberseceval: A secure coding benchmark for language models," 2023. [Online]. Available: https://arxiv.org/abs/2312.04724

[11] M. Bhatt, S. Chennabasappa, Y. Li, C. Nikolaidis, D. Song, S. Wan, F. Ahmad, C. Aschermann, Y. Chen, D. Kapil, D. Molnar, S. Whitman, and J. Saxe, "Cyberseceval 2: A wide-ranging cybersecurity evaluation suite for large language models," 2024. [Online]. Available: https://arxiv.org/abs/2404.13161

[12] S. Wan, C. Nikolaidis, D. Song, D. Molnar, J. Crnkovich, J. Grace, M. Bhatt, S. Chennabasappa, S. Whitman, S. Ding, V. Ionescu, Y. Li, and J. Saxe, "Cyberseceval 3: Advancing the evaluation of cybersecurity risks and capabilities in large language models," 2024. [Online]. Available: https://arxiv.org/abs/2408.01605

[13] E. Casalicchio and S. Iannucci, "The state-of-the-art in container technologies: Application, orchestration and security," *Concurr. Comput. Pract. Exp.*, vol. 32, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:211264375

[14] Y. Yang, W. Shen, B. Ruan, W. Liu, and K. Ren, "Security challenges in the container cloud," in *2021 Third IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. IEEE, 2021, pp. 137–145.

[15] S. Sultan, I. Ahmad, and T. Dimitriou, "Container security: Issues, challenges, and the road ahead," *IEEE access*, vol. 7, pp. 52 976–52 996, 2019.

[16] E. Truyen, D. Van Landuyt, D. Preuveneers, B. Lagaisse, and W. Joosen, "A comprehensive feature comparison study of open-source container orchestration frameworks," *Applied Sciences*, vol. 9, no. 5, p. 931, 2019.

[17] D. B. Bose, A. Rahman, and S. I. Shamim, "'under-reported'security defects in kubernetes manifests," in *2021 IEEE/ACM 2nd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS)*. IEEE, 2021, pp. 9–12.

[18] M. U. Haque, M. M. Kholoosi, and M. A. Babar, "Kgsecconfig: a knowledge graph based approach for secured container orchestrator configuration," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 420–431.

[19] R. Shevchuk, M. P. Karpinski, M. Kasianchuk, I. Yakymenko, A. Melnyk, and R. Tykhyi, "Software for improve the security of kubernetes-based ci/cd pipeline," *2023 13th International Conference on Advanced Computer Information Technologies (ACIT)*, pp. 420–425, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:264294763

[20] D. N. Kleidermacher, "Integrating static analysis into a secure software development process," in *2008 IEEE Conference on Technologies for Homeland Security*, 2008, pp. 367–371.

[21] H. H. AlBreiki and Q. H. Mahmoud, "Evaluation of static analysis tools for software security," in *2014 10th International Conference on Innovations in Information Technology (IIT)*, 2014, pp. 93–98.

[22] Z. Moric, V. Dakic, and M. Kulic, "Implementing a security framework for container orchestration," in *2024 IEEE 11th International Conference on Cyber Security and Cloud Computing (CSCloud)*, 2024, pp. 200–206.

[23] L. Al Mashta, "Containers: Security challenges and mitigation strategies: A systematic literature review," 2024.

[24] T. H. M. Le, R. Croft, D. Hin, and M. A. Babar, "A large-scale study of security vulnerability support on developer q&a websites," in *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*, 2021, pp. 109–118.

[25] Bridgecrew. (2024) Checkov. Prisma Cloud. Accessed: 2024-05. [Online]. Available: https://www.checkov.io/

[26] Aqua Security. (2024) Trivy. Aqua Security. Accessed: 2024-05. [Online]. Available: https://github.com/aquasecurity/trivy

[27] Tenable. (2024) Terrascan. Tenable. Accessed: 2024-05. [Online]. Available: https://runterrascan.io/

[28] controlplane. (2024) Kubesec. controlplane. Accessed: 2024-05. [Online]. Available: https://kubesec.io/

[29] Palo Alto Networks. (2024) Prisma cloud. Palo Alto Networks. Accessed: 2024-11-09. [Online]. Available: https://www.paloaltonetworks.com/prisma/cloud

[30] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, and H. Wang, "Retrieval-augmented generation for large language models: A survey," *arXiv preprint arXiv:2312.10997*, 2023.

[31] S.-Q. Yan, J.-C. Gu, Y. Zhu, and Z.-H. Ling, "Corrective retrieval augmented generation," *arXiv preprint arXiv:2401.15884*, 2024.

[32] Cloud Native Computing Foundation, "Cncf survey report 2020," The Linux Foundation, Tech. Rep., 11 2020, accessed: 2024-11-09. [Online]. Available: https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf

[33] EdgeDelta. (2023) Kubernetes adoption statistics. EdgeDelta. Accessed: 2024-11-09. [Online]. Available: https://edgedelta.com/company/blog/kubernetes-adoption-statistics

[34] The Kubernetes Authors. (2024) Kubernetes. The Linux Foundation. Accessed: 2024-05. [Online]. Available: https://kubernetes.io/

[35] A. Khraisat and A. Alazab, "A critical review of intrusion detection systems in the internet of things: techniques, deployment strategy, validation strategy, attacks, public datasets and challenges," *Cybersecurity*, vol. 4, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:232145452

[36] V. Mahajan and S. B. Mane, "Detection, analysis and countermeasures for container based misconfiguration using docker and kubernetes," *2022 International Conference on Computing, Communication, Security and Intelligent Systems (IC3SIS)*, pp. 1–6, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:252311879

[37] T. H. M. Le, B. Sabir, and M. A. Babar, "Automated software vulnerability assessment with concept drift," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 371–382.

[38] Cloud Native Computing Foundation. (2023) Artifact hub. The Linux Foundation. Accessed: 2024-11-09. [Online]. Available: https://artifacthub.io/

[39] ——. (2023) Cncf projects: Artifact hub. The Linux Foundation. Accessed: 2024-11-09. [Online]. Available: https://www.cncf.io/projects/artifact-hub/

[40] R. Croft, M. A. Babar, and M. M. Kholoosi, "Data quality for software vulnerability datasets," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 121–133.

[41] T. H. M. Le and M. A. Babar, "Automatic data labeling for software vulnerability prediction models: How far are we?" in *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2024, pp. 131–142.

[42] Mistral AI Team. (2023) Large enough. Mistral AI. Accessed: 2024-11-09. [Online]. Available: https://mistral.ai/news/mistral-large-2407/

[43] OpenAI. (2023) Gpt-4o mini: Advancing cost-efficient intelligence. OpenAI. Accessed: 2024-11-09. [Online]. Available: https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/

[44] Y. Wang, Z. Zhang, H. Chen, and H. Shen, "Reasoning with large language models on graph tasks: The influence of temperature," in *2024 5th International Conference on Computer Engineering and Application (ICCEA)*, 2024, pp. 630–634.

[45] T. H. M. Le and M. Ali Babar, "Mitigating data imbalance for software vulnerability assessment: Does data augmentation help?" in *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2024, pp. 119–130.

[46] T. H. M. Le, X. Du, and M. A. Babar, "Are latent vulnerabilities hidden gems for software vulnerability prediction? an empirical study," in *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 2024, pp. 716–727.

[47] V. Bandari, "A comprehensive review of ai applications in automated container orchestration, predictive maintenance, security and compliance, resource optimization, and continuous deployment and testing," *International Journal of Intelligent Automation and Computing*, vol. 4, no. 1, pp. 1–19, 2021.

[48] J. Farkouh, A. A. Donwoung, and A. Nazim, "Intelligent orchestration of containerized applications in cloud infrastructures," in *10th International Workshop on ADVANCEs in ICT Infrastructures and Services (ADVANCE 2023)*, 2023.

[49] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE symposium on security and privacy*. IEEE, 2014, pp. 590–604.

[50] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood, M. W. McConley, J. M. Opper, P. Chin, and T. Lazovich, "Automated software vulnerability detection with machine learning," *ArXiv*, vol. abs/1803.04497, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:3880688

[51] T. H. M. Le, D. Hin, R. Croft, and M. A. Babar, "Deepcva: Automated commit-level vulnerability assessment with deep multi-task learning," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 717–729.

[52] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3280–3296, 2021.

[53] T. H. M. Le, H. Chen, and M. A. Babar, "A survey on data-driven software vulnerability assessment and prioritization," *ACM Computing Surveys*, vol. 55, no. 5, pp. 1–39, 2022.

[54] T. H. Le, "Towards an improved understanding of software vulnerability assessment using data-driven approaches," *arXiv preprint arXiv:2207.11708*, 2022.

[55] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 608–620.

[56] T. H. M. Le and M. A. Babar, "On the use of fine-grained vulnerable code statements for software vulnerability assessment models," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 621–633.

[57] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "Vulrepair: a t5-based automated software vulnerability repair," in *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*, 2022, pp. 935–947.

[58] A. T. Nguyen, T. H. M. Le, and M. A. Babar, "Automated code-centric software vulnerability assessment: How far are we? an empirical study in c/c++," in *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2024, pp. 72–83.

[59] T. H. M. Le, M. A. Babar, and T. H. Thai, "Software vulnerability prediction in low-resource languages: An empirical study of codebert and chatgpt," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, 2024, pp. 679–685.

arXiv:2103.16423v2 [astro-ph.HE] 30 Apr 2021

Article

Gamma rays as probes of cosmic-ray propagation and interactions in galaxies

Luigi Tibaldo [1,*], Daniele Gaggero [2], and Pierrick Martin [1]

[1] IRAP, Université de Toulouse, CNRS, UPS, CNES, F-31028 Toulouse, France
[2] Instituto de Física Téorica UAM-CSIC, E-28049 Madrid, Spain
* Correspondence: luigi.tibaldo@irap.omp.eu (L.T.)

Abstract: Continuum gamma-ray emission produced by interactions of cosmic rays with interstellar matter and radiation fields is a probe of non-thermal particle populations in galaxies. After decades of continuous improvements in experimental techniques and an ever-increasing sky and energy coverage, gamma-ray observations reveal in unprecedented detail the properties of galactic cosmic rays. A variety of scales and environments are now accessible to us, from the local interstellar medium near the Sun and the vicinity of cosmic-ray accelerators, out to the Milky Way at large and beyond, with a growing number of gamma-ray emitting star-forming galaxies. Gamma-ray observations have been pushing forward our understanding of the life cycle of cosmic rays in galaxies and, combined with advances in related domains, they have been challenging standard assumptions in the field and have spurred new developments in modelling approaches and data analysis methods. We provide a review of the status of the subject and discuss perspectives on future progress.

Keywords: Gamma rays; cosmic rays; interstellar medium; Milky Way; galaxies; radiation mechanisms: non-thermal.

check for updates

Citation: Tibaldo, L.; Gaggero, D.; Martin, P. Gamma rays as probes of cosmic-ray propagation and interactions in galaxies. *Preprints* **2021**, *1*, 0. https://doi.org/

Received:
Accepted:
Published:

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Contents

1 Context and scope of the review **2**

2 The toolbox to study interstellar gamma-ray emission **4**
2.1 The progress of observational techniques in gamma rays . . . 4
2.2 Data complementary to gamma rays: recent step forwards . . . 7
2.3 A glimpse at the basics of cosmic-ray transport . . . 10
2.4 Evolutions of modelling and data analysis techniques . . . 13

3 Gamma-ray emission from the local interstellar medium: the Rosetta Stone of cosmic-ray astrophysics **17**
3.1 The emissivity of atomic hydrogen . . . 17
3.2 Molecular clouds and their spectra . . . 20

4 Large-scale interstellar gamma-ray emission: tracing cosmic rays throughout galaxies **21**
4.1 Cosmic-ray distributions through the disks of galaxies . . . 22
4.1.1 The Milky Way . . . 22
4.1.2 Implications of the gradient problem and inner-Galaxy hardening . . . 23
4.1.3 Local-group galaxies . . . 26
4.2 Cosmic-ray distributions through the halos of galaxies . . . 26
4.3 New frontiers: residual gamma-ray emission . . . 28

5 Gamma-ray emission in the vicinity of sources: the early steps of a long journey for cosmic rays **30**
5.1 Physical problem . . . 30
5.2 Emission beyond the shock in supernova remnants . . . 32
5.3 Emission around pulsars and their nebulae . . . 34
5.4 Emission coincident with star-forming regions . . . 35

6 The population of gamma-ray emitting galaxies: different realizations of the cosmic-ray phenomenon **38**
6.1 A growing source class: the gamma-infrared luminosity correlation and its implications . . . 38
6.2 How do cosmic-ray transport properties vary with galactic environment? . . . 40

7 Summary and perspectives **43**

References **47**

1. Context and scope of the review

Cosmic rays (CRs) are energetic particles first observed around the Earth with energies ranging from MeV to above 10^{20} eV and with approximately isotropic arrival directions. They are composed mainly of completely ionised nuclei, with protons accounting for a total fraction $> 90\%$ at GeV energies. They also include electrons, positrons, and antiprotons. The overall CR spectrum follows an approximate power-law distribution, which attests to the non-thermal origin of the particles. A most remarkable change of the power-law spectral slope occurs around 10^{15} eV, the so-called knee of the CR spectrum. Below the knee, the standard paradigm holding since the sixties [1] states that the particles originate in the Milky Way, very likely from shock acceleration in supernova remnants (SNR), and diffuse on turbulent magnetic fields in a kpc-sized halo encompassing the disk of the Galaxy for durations exceeding one Myr (see Gabici *et al.* [2] for a recent critical review).

CRs interact with interstellar matter and fields, producing secondary particles and radiation that are indirect means to study CRs in distant locations of the Milky Way, as well as in other galaxies. These observables usefully complement direct measurements of CRs in the heliosphere and allow us to develop our understanding of CR propagation and interactions. Among these indirect probes we find continuum gamma-ray emission produced by inelastic nucleon-nucleon collisions, Bremsstrahlung of CR electrons and positrons interacting with matter, and inverse-Compton (IC) radiation by CR electrons and positrons scattering off low-energy photons.

Overall, such probes show a fairly good agreement with the standard CR paradigm. However, many aspects are still debated or even largely uncertain, including the range of relevant transport and interaction mechanisms, their uniformity within galaxies, if and how they change based on galactic conditions, the microphysics foundation of all of these aspects, and the role played by CRs in galactic ecosystems. For the latter aspect, let us just briefly mention that CR astrophysics is getting increasing attention for its importance to other astronomical disciplines: astrochemistry and star formation [for a review see 3], galaxy formation and evolution [e.g., 4–6], and astrobiology [e.g., 7].

As early as in 1952 Hayakawa [8] had predicted that the decay of π^0 produced in inelastic nucleon-nucleon collisions in the Galactic disk would produce a measurable gamma-ray flux. This was confirmed in the '60s and the '70s thanks to the *OSO-3* [9] and *SAS-II* [10] satellites, which detected a gamma-ray signal associated with Galactic interstellar matter. The breakthrough in the field came thanks to the *COS-B* satellite (1975-1982), whose observations in the 50 MeV-5 GeV energy range enabled a detailed study of the correlation of gamma-ray

emission with interstellar medium (ISM) tracers and provided the first measurements of the large-scale distribution of CRs in the Milky Way [11].

The *Compton Gamma-Ray Observatory* (*CGRO*, 1992-1999) fully covered the energy range 1 MeV-30 GeV thanks to its two instruments COMPTEL and EGRET. *CGRO* data led to many in-depth studies of CRs in the Milky Way. EGRET also first probed CRs in external galaxies in gamma rays by detecting emission from the Large Magellanic Cloud (LMC) [12] and by setting an upper limit on emission from the Small Magellanic Cloud (SMC) [13]. The latter provided an upper limit on GeV CR densities at one third of the value observed locally in the Milky Way, and therefore established observationally the galactic origin of the particles as suggested twenty years earlier by [14].

The twenty-first century brought numerous and rapid advances in the domain of continuum interstellar gamma-ray emission studies:

- a dramatic extension of energy interval of the observations, spanning from sub-MeV to sub-PeV energies, with coverage of large portions of the Galactic plane and of the sky;
- full-sky observations of highest quality in the sub-GeV to sub-TeV domain, most notably thanks to the Large Area Telescope (LAT) onboard the *Fermi Gamma-ray Space Telescope*;
- impressive developments in theoretical and numerical tools for calculating interstellar gamma-ray emission properties, informed by spectacular improvements in the accuracy of direct CR measurements.

The next decade holds the promise for further observational advances thanks to new facilities already in the making or still in the planning/proposal phase, with guaranteed steps forward to be made in the TeV domain.

The rapid advances of the past few years and upcoming facilities make it timely to have a new review focussing on gamma rays as probes of CR propagation and interactions in galaxies, as an update to the previous ones touching these subjects [2,11,15,16]. In Section 2 we will briefly summarize the status of observational techniques, CR transport theory, modelling and data analysis tools, as well as complementary multiwavelength and multimessenger observations necessary to interpret the gamma-ray data. The following sections will review recent observations, their implications for CR physics, and future perspectives. They are organized around four broad questions.

Section 3 What do we learn from observations of the local interstellar medium near the Sun and how can we use them to connect direct and indirect CR measurements?

Section 4 What does interstellar emission tell us about the large-scale distributions of CRs in galaxies and what does it teach us about CR transport?

Section 5 What do we know about particle propagation and interactions in the vicinities of sources and what role does this phase play in the CR life cycle?

Section 6 What are the properties of gamma-ray emitting galaxies as a population and what do we learn about the variety of CR transport under different environmental conditions?

We will conclude with some final remarks and an outlook on perspectives on the coming years in Section 7.

This review will not cover the important and closely related aspects of CR origin and particle acceleration, treated in a companion paper in this volume [17], nor the propagation of CRs in the heliosphere [for a review see 18].

Before entering the main matter, we define here some terminology that will be used in the paper.

Interstellar/diffuse emission: we will refer to gamma-ray emission produced by CR interactions with interstellar matter and fields as interstellar emission. Conversely, we will use the term diffuse emission to refer to all emission that cannot be associated with

a localized object (e.g., a pulsar, a binary system etc.) that is individually detected. Based on this definition, diffuse emission will comprise of interstellar emission plus collective emission from populations of sources not detected individually (due. e.g., to instrumental sensitivity limitations).

Large-scale galactic CR population: we will use this term to refer to the CR population in a galaxy on spatial scales much larger than those where an individual CR source or sink (or localized groups thereof) can influence significantly the CR properties. The fact that this definition is useful in practice is based on observations of the Milky Way and local-group galaxies that will be discussed later in the paper. Conversely, we will avoid the term "CR sea", sometimes used in the literature with an ambiguous meaning that can refer to the CR population around the Earth, the large-scale galactic CR population (according to our definition), or the large-scale galactic CR population with an implicit assumption that it is uniform within (or even beyond) the galaxy.

2. The toolbox to study interstellar gamma-ray emission

2.1. The progress of observational techniques in gamma rays

Historically the most important facilities for the observations of interstellar gamma-ray emission have been space-based pair-tracking telescopes that cover the energy range from a few tens of MeV to tens of GeV and beyond. This is due to a combination of instrumental and intrinsic characteristics. Pair-tracking telescopes have a large field of view and a lower background than other gamma-ray detectors. Therefore, they are ideally suited to study diffuse emission, and for a long time they have been unrivalled in terms of sensitivity in the gamma-ray domain. Furthermore, in the GeV energy range we find the peak of energy output from CR interactions in the ISM, interstellar emission prevails over discrete sources, and it is dominated by hadronic emission correlated with interstellar gas (characterized by a well-defined morphology known from observations at other wavelengths, and, therefore, easier to separate from other emission components). The energy range covered by these instruments is often referred to as high-energy (HE) gamma rays.

In the past decade advances in the HE range have been driven by the *Fermi* LAT [19]. Thanks to the use of Silicon tracking devices the LAT has reached in the GeV domain an unprecedented sensitivity, field of view (2.4 sr), and angular resolution (better than $\sim 0.8^\circ$ at energies > 1 GeV and better than $\sim 0.15^\circ$ at energies > 10 GeV). The LAT has also extended the energy reach of this observing technique up to TeV owing to a combination of instrumental improvements, notably the use of a segmented anticoincidence detector for CR background rejection.

Gamma-ray observations at lower energies require the use of space-borne telescopes exploiting different detection techniques: coded masks in the energy range from hundreds of keV to MeV and Compton detectors at MeV energies. In this domain the state-of-the-art instruments date back to twenty or even thirty years ago with *INTEGRAL* SPI[1] [21] and COMPTEL [22]. Their performance cannot compete with the level reached by the LAT in the GeV domain [e.g., 23]. New missions have been proposed to improve observational capabilities in the MeV-GeV energy range thanks to Silicon detectors that can carry out at the same time Compton and high-angular-resolution pair-tracking measurements, most notably ASTROGAM and AMEGO [24–26]. Alternatively, GECCO is a concept of combined dual mode telescope that can improve measurements in the sub-MeV to MeV energy range thanks to an innovative imaging calorimeter as a standalone Compton detector and, at the same time, as a focal-plane detector for a coded aperture mask [27].

[1] For *INTEGRAL* legacy results see also [20].

The limited size of space-borne instruments makes measurements at energies beyond several tens of GeV more and more difficult. Therefore, at higher energies ground-based instruments are used. Their energy range is often referred to as very-high-energy (VHE) gamma rays, or ultra-high-energy (UHE) gamma rays beyond 100 TeV. Observational techniques in this energy range are covered in detail in a companion paper in this volume [28]. Below we will summarize the most important aspects with emphasis on the study of interstellar emission.

Ground-based Imaging Air Cherenkov Telescope (IACT) arrays have proven to be a very effective way to study gamma rays above a few tens of GeV [e.g., 29]. IACTs have fields of view of a few degrees and a high level of background due to CRs misclassified as gamma rays, and an angular resolution of several to a few arcminutes. This technique has reached its maturity with the current generation of arrays comprising 2 to 5 IACTs, namely H.E.S.S., MAGIC, and VERITAS. Among them, H.E.S.S., which is located in the Southern hemisphere and therefore has access to the inner part of the Milky Way, has engaged a systematic survey of the Galactic plane and has achieved the detection of diffuse emission that is likely to be, at least in part, of interstellar nature [30,31]. In this energy range, however, discrete sources prevail, thus we expect a sizeable fraction of diffuse emission to be due to unresolved sources not yet detected individually with the current sensitivity limitations (Section 2.4).

The field of IACTs is going to be revolutionised in the next few years by the advent of the Cherenkov Telescope Array (CTA) [32,33]. CTA will feature more than one hundred IACTs of different sizes located on two sites in the Northern and Southern hemispheres, thus it will be able to observe the entire sky. It will cover the energy range from a few tens of GeV to > 300 TeV with a sensitivity one order of magnitude better than current IACTs, a field of view reaching 10°, reduced CR background, and an angular resolution of a few arcmin.

A complementary observing technique for TeV gamma rays consists in ground-based shower particle detectors. Milagro has pioneered the use of water Cherenkov detectors [34], currently exploited by HAWC, which provides the best sensitivity among all existing instruments at energies > 10 TeV [35]. Alternative approaches are the use of scintillator detectors adopted by the Tibet Air Shower Array [36], or of resistive-plate counters adopted by ARGO-YBJ [37]. These instruments have a large field of view, corresponding to the entire sky not occulted by the Earth, and a high duty cycle (contrarily to IACTs that operate only at night). Conversely, their angular resolution is not as good as for IACTs. For example, for HAWC the angular resolution varies between 1° and 0.2° [35].

The LHAASO observatory, still under construction, combines shower particle detectors and IACTs. The expected steady-source sensitivity will be superior to that of CTA above a few tens of TeV [38]. All of the ground-based shower particle detectors mentioned so far were or are located in the Northern hemisphere. A new project has been proposed to install a water Cherenkov shower particle detector system in the Southern hemisphere, SWGO, which will then be able to observe the inner Milky Way and the Magellanic Clouds [39].

To illustrate the status of observations Figure 1 shows some recent maps of the Milky Way from different instruments. The all-sky observing capabilities of the *Fermi*-LAT make it an invaluable source of information to study the entire range of manifestations of CR propagation and interactions. We note that features correlated with interstellar structures in the Milky Way are clearly visible in the map in Figure 1(**a**) even though no background subtraction has been applied beyond event-wise selection of candidate photons.

Fermi-LAT observations are complemented at higher energies by observations with IACTs, notably H.E.S.S., that for the moment can cover much more limited regions of the sky albeit with greater detail thanks to their superior angular resolution. Another limitation of IACTs is related to the presence of a much larger background due to CR interactions with the Earth's atmosphere. The traditional analysis techniques employed to deal with residual

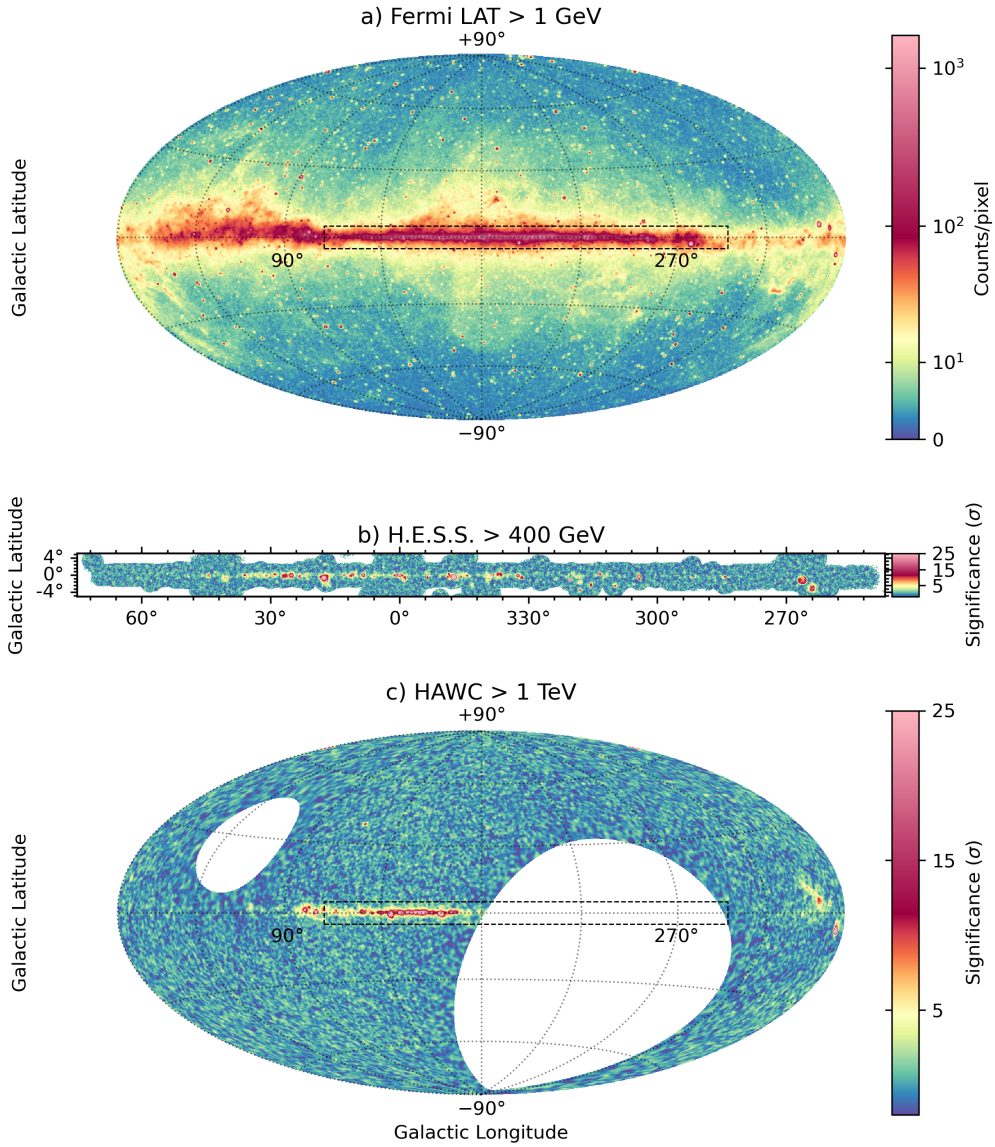


Figure 1. Images of the Milky Way from different instruments. (**a**) *Fermi*-LAT, 12 years of P8R3 data, energies > 1 GeV, *Source*/PSF3 event class/type, zenith angles < 100°, smoothed with a Gaussian kernel of $\sigma = 0.25°$. (**b**) H.E.S.S. Galactic plane survey, map described in [40] (0.2° correlation radius). (**c**) HAWC survey, map construction described in [41] (test source with 0.5° extension and a power-law spectral index of 2.5). The footprint of map (**b**) is overlaid to the all-sky maps in (**a**) and (**c**) as a dashed rectangle. Map (**a**) displays observed counts not corrected for residual CR background, while maps (**b**) and (**c**) are given in units of significance of gamma-ray emission above the residual CR background. For maps (**b**) and (**c**) the actual energy threshold varies across the map and the figure provides a representative value.

CR background are based on "Off" regions, most often chosen within the same field of view of the position of interest. The map in Figure 1(**b**) is constructed using "Off" regions in the shape of rings centered at the same position as the region of interest, or "On" region, that is a circle of fixed radius, sometimes referred to as correlation radius [see 42, for an illustration, in Figure 4, as well as for a review of the traditional background estimation techniques]. Due to the limited field of view of IACTs this results in a lesser sensitivity to extended emission [see, e.g., the discussion in 43]. Alternative background estimation methods, either data-driven or simulation-based, are sought [44–47]. The large times necessary to map large portions of the sky with IACTs and lesser sensitivity to extended emission makes their contributions to the study of large-scale interstellar emission less rich than those from wide-field of view instruments.

Shower particle detectors kick in at even higher energies offering a wide-field view of the sky as illustrated in the HAWC skymap shown in Figure 1(**c**). In spite of a large CR background, the large field of view of these detectors makes it possible to routinely employ a data-driven background estimation method known as the "direct integration" technique [48], which exploits the facts that the CR background is stable in time and varies smoothly as a function of conditions in the atmosphere and the detector (e.g., trigger rate). Background estimation is performed independently in large declination bands, with an accuracy of order 10^{-4} limited by anisotropies in the primary CR arrival directions [48]. This technique is therefore well suited for the study of large-scale emission.

2.2. Data complementary to gamma rays: recent step forwards

The study of interstellar gamma-ray emission is deeply intertwined with other multi-messenger/multi-wavelength measurements and observations, that we can group in five broad categories:

- direct CR measurements;
- the study of other ISM constituents: matter, radiation fields, magnetic fields;
- other indirect CR tracers, that is, synchrotron emission, neutrinos, ion and nuclear lines;
- the census of energetic objects such as massive stars, SNRs, pulsars and their wind nebulae;
- hadronic interactions cross-sections.

In this section we will very briefly review these five domains and highlight some recent results, with emphasis on aspects of particular importance for the subjects covered in the review.

Direct CR measurements assess the spectra, composition, and arrival direction anisotropies of charged particles around the Earth. For sub-knee CRs this is prevalently achieved using satellite- or balloon-borne particle detectors, although ground-based instruments studying the byproducts of CR interactions in the atmosphere can also explore the energy range around the knee. The past few years have been marked by the high-precision measurements of CR spectra performed by AMS-02 on the International Space Station of a wide range of species, including: light nuclei [49,50], heavier nuclear species [51–53], electrons [54] and secondary species produced by CR interactions in the ISM [55]. They are complemented by new results on the abundances of heavy nuclei, e.g., [56]. Linking these measurements with gamma-ray observations is complicated by the fact that charged particles near the Earth below rigidities of $\sim$100 GV are affected by the solar wind, which modulates their spectra as a function of the solar cycle phase. This limitation has been overcome only recently for rigidities below $\sim$1 GV thanks to measurements of light CR species in interstellar space with the Voyager 1 probe. [57]. On the higher-energy portion of the spectrum, the TeV domain has also witnessed a significant advance, thanks to the measurements of several different balloon/satellite/ground experiments, both for hadronic species (e.g., from DAMPE [58], ATIC [59], and NUCLEON [60]) and for leptons (e.g., from H.E.S.S. [61,62], CALET [63], and DAMPE [64]).

Interstellar matter constitutes a target for the production of gamma-ray emission via nucleon-nucleon inelastic collisions and electron Bremsstrahlung. The gamma-ray yield is proportional to the mass which resides predominantly in gas, the most important contributor being hydrogen, either in the atomic, molecular, or ionized form. *Atomic hydrogen* (H I) is widely distributed in galaxies, and can be traced thanks to the 21-cm hyperfine transition line. The velocity-integrated brightness temperature of the 21-cm line is directly proportional to the gas column density in the optically-thin limit. Often we need to account for H I opacity, which is typically done under the approximation of a uniform spin temperature. Recent years brought remarkable advances in the observations both for the Milky Way at large [65,66], and for specific Galactic or extragalactic regions. *Molecular hydrogen*, H_2, mostly concentrated in cold clouds, is difficult to observe directly and is most often traced indirectly using molecular lines of other species. The mm rotational lines of the second most abundant interstellar molecule, CO, with its different isotopes, have been a major tool in gamma-ray astronomy. While for Milky Way on large scales we still rely on the survey by Dame *et al.* [67], high-resolution surveys of specific regions or external galaxies become more increasingly available, e.g., [68,69]. It is empirically established that molecular hydrogen column densities, $N(H_2)$, are approximately proportional to the velocity-integrated brightness temperature of the ^{12}CO $J = 1 \rightarrow 0$ line, W_{CO}, via the $X_{CO} = N(H_2)/W_{CO}$ factor, for which, however, variations are both observed and expected [70] (see also the discussion in Section 3). *Ionized hydrogen*, H II, is present in regions around star-forming regions and in a kpc-wide layer around the Galactic disk. The Hα recombination line in the visible band is heavily absorbed in the ISM. Alternative tracers are provided, under different kinds of hypotheses and approximations, by microwave free-free emission [71], pulsar dispersion measurements [72], and radio recombination lines [73]. Owing to Doppler shift from the Galactic rotation, lines of all kinds can be used to separate different structures along the line of sight, and approximately locate them.

An alternative approach to trace interstellar matter relies on *dust*. Dust grains make a tiny fraction of the mass in the ISM, but they produce bright thermal emission in infrared and are responsible for stellar extinction in the near-infrared to visible domain. They are thought to be well mixed with gas, therefore their emission/extinction can be used as a tracer of total ISM masses. Recent observational developments include the improved mapping of thermal dust emission thanks to the *Planck* satellite [74] and strong advances in 3D dust mapping based on stellar extinction measurements combined with stellar population synthesis models, e.g., [75,76]. As for X_{CO}, variations in the ratios between dust observables and matter column densities are both observed and expected, e.g., [77]. The combination of gamma-ray and dust observations (both tracers of the total masses in the ISM) with the H I and CO lines has demonstrated that the aforementioned lines fail to trace the totality of the neutral interstellar medium. The excess with respect to the H I- and CO-bright gas is known as dark gas or *dark neutral medium* (DNM). It is predominantly located at the interface between the molecular-dominated and atomic-dominated parts of interstellar clouds, and it is likely made of a combination of optically-thick H I and CO-poor H_2 in the outer layers of the molecular regions less shielded from UV photo-dissociation [for a recent review see 16]. The existence of the DNM is confirmed by alternate molecular tracers, e.g., [78,79] and emission from ionized carbon, e.g., [80], and also supported by numerical simulations, e.g., [81].

Interstellar radiation fields (ISRFs) constitute a target for the production of gamma-ray emission via inverse-Compton scattering by CR electrons and positrons. They include the cosmic microwave background, thermal emission from dust grains heated by stellar radiation from sub-mm to infrared, and radiation from stars from near infrared to UV. Radiative transfer techniques can be used to link the ISRFs to the measured spectral energy distributions and other observational constraints on the spatial distribution of stars and interstellar dust. Recent years have seen significant advances in this field both on the observational front, notably with

the improved measurements of thermal dust emission thanks to the *Planck* satellite [74], and on the modelling front for the Milky Way [82–84] as well as for nearby external galaxies, e.g., [85,86]. The importance for gamma-ray observations was discussed recently by Niederwanger *et al.* [87].

Magnetic fields are relevant to CRs both as a target for synchrotron energy losses/radiation and as the agent of diffusion (see Section 2.3 for a discussion on the latter). We can separate interstellar magnetic fields into a large-scale regular and a turbulent component. In external galaxies they are known to follow a spiral structure similar to that of interstellar matter and stars [for a recent review see 88]. The origin of the regular field is still debated. It is constrained through rotation of polarized emission from background sources, e.g., pulsars, polarized synchrotron emission, and polarized dust emission. Many recent works have used observational constraints to model the large-scale Galactic magnetic field, e.g., [89–93], and have been revisited on the light of *Planck* results [94]. Magnetic turbulence is thought to be driven by supernova explosions, stellar winds and outflows, shocks and instability induced by galactic rotation, and shear instabilities and baroclinic effects in the ISM. It is related to interstellar turbulence in velocity, matter density, and free-electron density. It is therefore constrained observationally using high-resolution spectroscopy of interstellar lines, high-resolution imaging of interstellar matter, intensity and polarization of synchrotron and dust thermal emission, dispersion of pulsar signals, interstellar scintillation, and rotation of polarized emission from background sources [for a recent review see, e.g., 95]. The combination of these observations reveals an overall power-law spectrum as a function of wavenumber with Kolmogorov slope over spatial scales from thousands of km to a few pc [96].

Indirect CR tracers other than gamma-ray continuum emission include the already mentioned *synchrotron emission*, observed from radio to microwaves. The microwave sky was recently studied in unprecedented detail by the *Planck* satellite. Studies of synchrotron emission are used to reconstruct the broadband spectrum of CR electrons and to inform the interpretation of observations of IC emission in gamma rays [97,98]. Only a few years ago the first detection of *astrophysical high-energy neutrinos* with IceCube [99] has opened a new window that may enable us to have a complementary tracer of CR nuclei in galaxies, but for the moment only upper limits to a Galactic neutrino signal exist combining data from IceCube and Antares [100]. *Molecular line emission* driven by CR ionization, which yields, e.g., H_3^+, OH^+, H_2O^+, H_3O^+, is observed in infrared and mm waves and provides information on the low energy part of the CR spectrum [for a recent review see 16]. Recent calculations show that the observed molecular ion line emission suggests an average ionization rate a factor of 10 larger than what is expected from directly measured CR spectra (including results from Voyager 1) [101,102]. This may point to the existence of an additional CR component emerging at low energies different from those observed directly or through gamma rays, although it seems more likely that ionization sources different from CRs may play a role more prominent than previously thought [103] Furthermore, we note that the methodology used to infer the ionization rate from the data is very sensitive to the composition of the ISM, e.g., to the presence of polycyclic aromatic hydrocarbons [104]. An alternative way to study the CR nuclei population in remote locations below the pion production threshold (kinetic energies of $\sim$300 MeV/nucleon) would be to observe *nuclear de-excitation lines* in the $0.1 - 10$ MeV range induced by CR collisions with interstellar matter [105,106] thanks to a future MeV telescope [107].

Energetic objects play a twofold role: they are potential CR accelerators and they inject energy into the ISM under other forms, e.g., radiation and magnetic turbulence. Knowledge of their census and its recent history is therefore essential to model interstellar gamma-ray emission and interpret the gamma-ray observations. Different challenges are to be faced to determine the distribution in space and time of energetic object at galactic scales against obser-

vational uncertainties and biases, or to establish a detailed picture of individual remarkable regions. Among the different classes of energetic objects, massive stars are interesting at the same time for themselves, and as the progenitors of all other relevant classes such as SNRs and pulsars and their wind nebulae. Our view of stellar populations in the Milky Way is in a transformative phase thanks to the data collected by the *Gaia* satellite in the visible/near-infrared band, which provide precision measurements of positions, parallaxes, and proper motions of over 1.4 billions stars within > 4 kpc around the Solar system [108]. This makes it possible to paint a portrait of the history of stellar clusters in the disc of the Milky Way over the past billion years [109], which complements information from observations in near-infrared, e.g., [110], or at lower frequencies, e.g., [111]. At the cluster or star-forming region level, this enables us to go beyond simple models of coeval and colocated star formation, and embrace more realistic descriptions of its spatial [112] and temporal distributions [113]. For SNRs, pulsars and pulsar wind nebulae (PWNe) the most important wavebands for the observations are radio, X rays, and gamma rays. For the first two bands the coming decade is expected to be marked by the results from the Square Kilometer Array [114] and its precursors and pathfinders[2], and the eROSITA space telescope [115]. The role of gamma-ray observations in understanding particle acceleration in energetic objects is treated in a companion paper in this volume [17].

Hadronic cross sections for gamma-ray production are an essential ingredient to model and interpret gamma-ray observations. While leptonic cross sections in principle can be calculated exactly, the modelling of hadronic interactions heavily relies on experimental constraints. Accelerator data provide information with a somewhat limited coverage in terms of energies, angular distribution, and interacting species (mostly p-p), that are then used to model the cross sections resulting in non-negligible uncertainties[for a recent review see 116]. Accelerator data in the crucial energy range above the pion production threshold and around the $\Delta(1238)$ isobar resonance, and up to center-of-mass energies of 10^3 TeV mostly date back to the '50s-'80s and have been compiled by Lock and Measday [117], Stecker [118], and Dermer [119]. The energy coverage was recently extended up to 10^8 TeV in the center-of-mass frame thanks to the LHCf experiment [120,121], and improved also at hundreds GeV energies thanks to the NA61/SHINE experiment [122]. Recent cross-section derivations exploiting these data include Kamae *et al.* [123], Kelner *et al.* [124], Kachelrieß and Ostapchenko [125], Kafexhiu *et al.* [126], Mazziotta *et al.* [127], Kachelrieß *et al.* [128], and, with focus on the contributions of species heavier than protons, Mori [129], and Kachelrieß *et al.* [130].

2.3. A glimpse at the basics of cosmic-ray transport

Gamma-ray emission from the galactic ISM is intimately associated with the physical problem of CR acceleration and transport. The problem of CR acceleration is not covered in this review. We just recall that the SNR paradigm is widely considered as the reference guideline, although other classes of sources powered by a variety of mechanisms have been proposed as well (e.g. OB associations, X-ray binaries, and pulsar wind nebulae for leptonic CRs). Within the SNR scenario, the theory of diffusive shock acceleration [131–134] and its nonlinear extension [135] describe the process o acceleration of cosmic particles that are diffusing around the forward shock in an SNR, and are able to reproduce the correct CR energetics and overall many of the CR observables. We refer to Blasi [136] for an extensive review on the origin of CRs and the SNR paradigm, and also to Cristofari [17] in this volume for the role of gamma-ray observations in this context.

In this Section we focus instead on the basics of the problem of galactic CR transport. Let us start by mentioning that a large body of evidence demonstrates that high-energy charged

[2] https://www.skatelescope.org/precursors-pathfinders-design-studies/

CRs are confined in the Milky Way for a timescale that is much longer than the ballistic crossing time. In fact, the analysis of the properties of the CR fluxes that reach Earth outlines the following key features:

- isotropy at the level of $\sim 10^{-3}$ in the arrival directions for the entire energy range covered by the experiments that are sensitive to this observable, in particular in the TeV - PeV range [137–142]; this suggests that particles have suffered multiple deflections in their journey from the acceleration sites to Earth;
- a significantly larger abundance of some light species, namely Lithium, Beryllium, and Boron, with respect to the solar system abundances; this piece of evidence in particular is naturally interpreted as the smoking gun of the cumulative interactions between the primary species injected by the accelerators (protons and heavy nuclei) and interstellar gas; the total column density that primary CRs have to cross to produce the observed amount of light species is as large as a few g/cm^2: such a large value of the *grammage* strongly suggests once again that the CRs that produce the secondary species have crossed the Galactic disk multiple times over time scales exceeding one Myr.

The picture is corroborated by the ubiquitous observation of magnetic turbulence in the interstellar environment that we briefly mentioned in the previous section. The multiple, random interactions of charged CRs with these perturbations naturally provide a mechanism to explain why the motion of these particles should be described as a diffusive process. These considerations, together with an increasing amount of data in different channels discussed in the previous section, corroborate the standard paradigm that seems to capture the most relevant aspects of Galactic cosmic-ray physics, as recently extensively reviewed in [2], in which CRs are diffusively confined within a kpc-sized, magnetized and turbulent Galactic halo.

The simplest way to describe this phenomenon, widely used in the past literature, is provided by so-called *leaky-box models* [143]. In this framework, the galaxy is modelled as a cavity with almost perfectly reflecting "walls". The cosmic particles are allowed to move freely within this environment. The physics of their propagation and escape is all embedded in the energy-dependent parameter τ_{esc}, i.e., the mean residence time. Thus, the probability of particle escape per unit time is equal to τ_{esc}^{-1}. The model is described by the following equation for the particle density N:

$$\frac{\partial N}{\partial t}(E) = \frac{N}{\tau_{esc}(E)} + Q(E), \tag{1}$$

where $Q(E)$ is the source function.

Recently, a description in terms of diffusion has become prevalent. We want to emphasize that it is very challenging to obtain a general expression of the diffusion tensor from first principles. A widely used guideline in this context is the *quasi-linear theory of pitch-angle scattering* onto magnetic fluctuations presented in the pioneering papers of [144,145]. The rationale of the theory is to consider the interaction of a charged particle of momentum $\vec{p} = \gamma m \vec{v}$ with magnetic inhomogeneities $\delta\vec{B}$. The key assumptions behind this theoretical framework are the following:

- the magnetic inhomogenities are Alfvénic; they are isotropic and their energy density is characterized by a power-law energy spectrum $P(k)$ as a function of wavenumber k;
- the inhomogenities are small, at the scale of interest, with respect to the coherent large-scale regular magnetic field B_0.

The key result of this approach is that the particles mainly diffuse along the regular magnetic field. The process is resonant, i.e., the Alfvén wavepackets that contribute to the process have a wavelength comparable to the gyroradius of the particle. It is useful to notice that the length scales associated to the energy range usually covered by current CR

observations are typically very small compared to the size of a galaxy, and to the scale of injection of turbulence (10-100 pc). For instance, GeV particles resonate with fluctuations with wavelength of the order of few AU.

The resulting scattering rate can be written as [1,136,146]:

$$\nu = \frac{\pi}{4} \frac{k_{\rm res} P(k_{\rm res})}{B_0^2/(8\pi)} \Omega_g$$

where $\Omega_g = qB_0/(\gamma mc)$ is the gyration frequency and the resonant wavenumber is $k_{\rm res} = \Omega_g/v_\parallel$ ($v_\parallel$ is the velocity component along the coherent magnetic field B_0).

Starting from this expression, it is possible to obtain a (parallel) spatial diffusion coefficient of this form:

$$D(p) = \frac{v^2}{3\Omega_g} \frac{B_0^2/(8\pi)}{k_{\rm res} P(k_{\rm res})}.$$

It is useful to recast this expression into:

$$D(p) = \frac{1}{3} \frac{r_L v}{\mathcal{F}(k_{\rm res})}$$

where $r_L = p_\perp/qB_0$ is the Larmor radius of the particle and we have defined

$$\mathcal{F}(k) \equiv \frac{kP(k)}{B_0^2/(8\pi)}.$$

This expression shows that a larger power in magnetic fluctuations at a certain scale is associated to a lower diffusion coefficient for the resonating particles, hence a more effective confinement. The dependence on the Larmor radius, both direct and indirect via the resonant wavenumber $k_{\rm res}$, and the empirical power-law dependence on wavenumber of the magnetic turbulence spectrum observed at large scales drive a dependence of the diffusion coefficient on particle rigidity R. Standard implementations for the Milky Way feature a diffusion coefficient $D(R) = \mathcal{O}(10^{27})\,\beta\,(R/1\,{\rm GV})^{1/3}$ cm^2 s^{-1} in reasonable agreement with a reference estimate of the random field at the injection scale and extrapolation down to the resonant scale.

We emphasize that the theory is typically built on an isotropic picture of turbulence. However, the resulting process is highly anisotropic. We will elaborate more on these key concepts in the next Section.

Diffusive confinement is certainly a key feature characterizing CR propagation. However, all CR species interact in many different ways with the different components of the ISM, and a variety of other processes occur during their random walk across the parent galaxy. Let us briefly recap the most relevant ones.

- **Reacceleration:** This process is intimately connected to spatial diffusion. In fact, the random walk in space is expected to be accompanied by a diffusion in momentum space, since the scattering centers (namely, magnetic fluctuations) are not static. They are instead in random motion themselves, with characteristic velocities of the order of the Alfvén speed. The importance of this process hence depends on the large-scale average of this quantity over the galaxy, and has been the subject of a long debate. We refer to [147] for a critical look at this issue in the case of the Milky Way, in connection with the total energy budget available in the Galaxy.
- **Advection:** This is a rigidity-independent process that can significantly contribute to the vertical escape of CRs and is associated to the existence of so-called galactic winds. This phenomenon consists in a powerful outflow that may extend for hundred of parsecs, possibly more relevant in the inner part of galaxies, and induce a relevant mass

loss, possibly comparable to the mass formed in stars. Possible mechanisms to create galactic winds are currently under debate. Winds may be powered by stellar winds and supernova explosions, and a non-linear interplay with CRs may also be at work: cosmic rays (CRs) escaping from the galaxy can effectively push on the ISM and eventually trigger the wind itself.

- **Energy losses:** Different types of interactions transfer energy from the CR population to the ISM. In particular, hadrons loose energy due to ionization, Coulomb interactions with interstellar matter, and production of secondary particles in nucleon-nucleon inelastic collisions, most notably pions (these processes being overall more effective at low energy, in particular in the sub-GeV domain). On the other hand, leptons suffer strong losses mostly due to IC scattering onto low-energy photons, synchrotron emission (with a rate that increases with increasing energy, following a $\propto E^2$ scaling), and Bremsstrahlung. We refer for instance to [148] for a compilation of the relevant formulae associated to these processes.
- **Spallation:** A complex network of nuclear reactions and decays transform heavier CR nuclei into lighter species as a consequences of inelastic interactions with matter in the ISM. A combination of semi-empirical parametrizations and rescaling procedures to nuclear data (mostly available in the GeV domain) is typically adopted to model these phenomena. See for instance [149–154] and references therein for the modelling of the hadronic nuclear network.

Remarkably, a joint description of the most relevant phenomena mentioned above is possible in the form of a relatively compact *transport equation* that can be solved for each CR species of interest. This general reacceleration-diffusion-loss equation is usually written as follows: [1,146].

$$\begin{aligned} -\boldsymbol{\nabla}\cdot(D\boldsymbol{\nabla}N_i + \boldsymbol{v}_w N_i) + \frac{\partial}{\partial p}\left[p^2 D_{pp}\frac{\partial}{\partial p}\left(\frac{N_i}{p^2}\right)\right] - \frac{\partial}{\partial p}\left[\dot{p}N_i - \frac{p}{3}(\boldsymbol{\nabla}\cdot\boldsymbol{v}_w)N_i\right] = \\ Q + \textstyle\sum_{i<j}\left(c\beta n_{\rm gas}\,\sigma_{j\to i} + \frac{1}{\gamma\tau_{j\to i}}\right)N_j - \left(c\beta n_{\rm gas}\,\sigma_i + \frac{1}{\gamma\tau_i}\right)N_i \end{aligned} \tag{2}$$

In this equation: p is the particle momentum; N_i is the CR density for species i; D is the spatial diffusion tensor; D_{pp} the diffusion coefficient in momentum space, associated to reacceleration; $\boldsymbol{v}_w$ the velocity associated to advection; Q is the source term; $(\sigma_{j\to i}, \sigma_i)$ are the spallation cross sections associated, respectively, to the creation of the species i from parent nucleus j, and to the destruction of the species i; $(\tau_{j\to i}, \tau_i)$ are the decay times for, respectively, the unstable species j, creating i, and for i, creating lighter nuclei. For a detailed discussion on each term, we refer to the technical papers cited above.

2.4. Evolutions of modelling and data analysis techniques

Extracting properties of CRs from the gamma-ray data always involves some type of modelling. The modelling of CR propagation and interactions and the associated non-thermal emission can follow different avenues. We will review in this Section different methods and discuss the most remarkable achievements and open questions.

The **template fitting** method is a widely-used technique aimed at modelling observations of interstellar gamma-ray emission. In its simpler form the key assumption is that CR densities vary mildly on the spatial scales characteristic of interstellar gas complexes. Therefore, gamma-ray emission associated with interstellar gas can be modelled as a linear combination of maps (templates) of gas column density, or a proxy thereof, split for different regions along the line of sight by using the Doppler shift information of interstellar lines. The original implementation of this method was using templates derived from the H I 21-cm line to account for atomic gas and from the 2.6-mm CO line as a surrogate tracer of molecular gas [155]. The linear combination coefficients, known as gamma-ray emissivities, encode information

on CRs. Notably, the H I emissivity is nothing else that the gamma-ray emission rate per hydrogen atom, i.e., the convolution of the CR densities with the gamma-ray production cross sections. The gamma-ray analysis can be performed over several independent energy bins to reconstruct the underlying CR spectrum. In recent years the template fitting method has been extended to account for other forms of interstellar gas (dark neutral medium, ionized gas), and other components of interstellar emission, notably IC emission. For the latter templates need to be obtained using predictive models. More recently, the `SkyFACT` tool [156] introduced the possibility to allow pixel-by-pixel variation within each template guided by a penalized likelihood maximization by combining methods of image reconstruction and adaptive regression. Crucial uncertainties affecting the template fitting technique come from approximations in the construction of the templates, and cascade effects that stem from those in the component separation procedure.

A specular technique consists in making assumptions about the spectra of different gamma-ray emission components, either based on theory or observations, and use the gamma-ray data to infer the morphology of the components. This technique is known as spectral component analysis or spectral template fitting [157,158]. This alternative incarnation of template fitting has been used less widely owing to the absence of sharp spectral features in typical gamma-ray spectra and because the $\gtrsim$5-10% energy resolution of gamma-ray telescopes makes it less effective. We warn the reader that caution is needed in dealing with the energy-dependent point sprwad function (PSF) of the instruments when applying this technique.

Another completely **data-driven** technique aimed at studying gamma-ray emission is represented by the D^3PO (Denoised, Deconvolved, and Decomposed) inference algorithm presented in [159]. This method performs a Bayesian inference without the use of templates. Instead, it is designed to remove the shot noise, deconvolve the instrumental response, and finally provide estimates for the different flux components separately. This method is particularly suited in identifying and subtracting point sources from the data to study the remaining emission.

Let us now turn our attention to another widely adopted approach to gamma-ray modelling, which is the use of **predictive models** connected to the physics of CR propagation in galaxies. The rationale of these methods is to compute the equilibrium CR distribution in the galaxy by solving the transport equation (Equation 2) presented in Section 2.3. We have discussed how such equation captures the variety of physical processes shaping the transport of the cosmic particles from their production at the accelerator sites to their eventual escape from the large-scale diffusive halo. Today we have at our disposal several public numerical codes, equipped with different numerical methods and astrophysical ingredients, aimed at solving that equation for all CR species in the Milky Way. The most important are (in chronological order): `GALPROP`[160–162], `DRAGON` [148,154,163,164], `PICARD` [165,166]. A semi-analytical approach is instead followed by the `USINE` project [167].

The models based on this concept were remarkably successful in reproducing a variety of local CR data, and, for some of them, in modelling the non-thermal emission from the radio band all the way up to the GeV-TeV gamma-ray domain. In standard implementations the CR transport is typically described as isotropic, homogeneous diffusion characterized by a scalar diffusion coefficient with a power-law rigidity dependence. The amplitude and slope of the diffusion coefficient, together with a set of parameters associated to the astrophysical ingredients of the model (for instance, the $X_{\rm CO}$ conversion factor between the CO emission intensity and the molecular gas column density) are typically fitted to a variety of data, including the accurate dataset of secondary/primary nuclei provided by AMS-02, and the gamma-ray maps provided by the *Fermi*-LAT.

However, a number of anomalies have been highlighted over the recent years both in CR and gamma-ray data: a break in the local proton, Helium, light and heavy nuclei spectra at ∼200 GV [49,50,168], an excess of high-energy positrons first highlighted by PAMELA [169] and then confirmed by *Fermi*-LAT and AMS-02 (with better statistics) [170], a hint of an antiproton excess near 80 GeV [171,172], and several observations in gamma rays that will be discussed in detail in Section 4.

Some of these anomalies posed a challenge to standard implementations of Galactic CR models, and have spurred a variety of new developments.

- **Three-dimensional modelling.** The transport equation is usually solved under the assumption of cylindrical symmetry. In this widely used setup, that is a distinctive feature of standard Galactic CR model implementations, most astrophysical ingredients that enter the problem and influence the different types of CR interactions (for instance, the distribution of CR accelerators, the magnetic field strength, the interstellar gas and low-energy photon distributions) are implemented in the form of (smoothly-varying) functions of the Galactocentric radius R and the vertical coordinate z (perpendicular to the Galactic plane). A more realistic description of the interstellar medium, featuring a three-dimensional model for the spiral arm pattern of the Milky Way in the CR source term was first introduced in the context of numerical modelling of leptonic CR species in [164], showing a relevant impact on the local electron spectrum. The consequences of such three-dimensional pattern on CR hadronic species and gamma-ray modelling was later discussed in [173,174]. The authors of [175] further investigated the phenomenological consequences of a spiral arm pattern in both the CR source distribution and the interstellar radiation field (see also [98]).
- **Inhomogeneous diffusion.** The transport equation is usually solved under the assumption that the diffusion coefficient is constant in both slope and normalization across the Galaxy. Several steps towards a inhomogeneous description of the problem have been proposed over the years. For instance, in [176] the radial variation of the diffusion coefficient normalization was explored as a possible solution to the long-standing anomaly usually identified as *gradient problem* (Section 4). Moreover, in [177], a complete scan of the parameter space for CR injection and propagation showed that each set of species is probing a very different interstellar medium, providing further evidence in favour of non-homogeneous transport.
- **Anisotropy of CR transport.** The anisotropic nature of CR transport has been highlighted by theorists since the first pioneering studies about the quasi-linear theory of pitch-angle scattering onto magnetic fluctuations. We have stressed in the previous section that the main feature of such theory is precisely a diffusive motion that is *strongly anisotropic* locally and goes predominantly along the magnetic field lines. However, since relevant fluctuations of magnetic field on scales close to the ones associated to the injection of turbulence may in principle lead to some isotropization of global CR diffusion in the Galaxy, the typical approach in the context of large-scale modelling of CR transport based on the transport equation has been to neglect any anisotropy in the diffusion part (see for instance the discussion in [15]). Very few attempts to consider anisotropic transport aimed at reproducing local CR data by solving the large-scale diffusion equation exist in the literature. A relevant example is [178] where a fully anisotropic diffusion equation is solved numerically in cylindrical symmetry.
- **CR self-confinement and non-linear effects.** We have mentioned that the magnetic irregularities (typically Alfvén waves) play a central role in the current description of CR confinement in the Galaxy. Those waves may be either part of a pre-existing turbulent cascade, or generated by the CRs themselves, if they stream faster than the Alfvén speed, via the process of *streaming instability* [179]. The latter option may give rise to non-linear

effects in the CR transport, typically refereed to as *CR self-confinement*. This phenomenon can dominate the transport at low energies, and can be responsible for a spectral feature at the energy where the scattering onto pre-existing turbulence start to dominate (see for instance [180]). An interesting interpretation of the hardening in the CR spectra observed by the AMS collaboration was presented in [181]. An even more refined description of CR transport in the Galaxy where the (coupled) CR diffusion equation and the equation for self-generated waves are solved numerically is presented in [182]. In this work, the size of the diffusive halo is derived from the model, as the result of a combination of wave self-generation and advection from the Galactic disc. These effects take an even more prominent role in the proximity of CR accelerators (Section 5).

- **Anisotropy of the turbulent cascade. CR scattering onto magnetohydrodynamic turbulence modes.** The nature of the pre-existing magnetic fluctuations responsible for CR confinement in the high-energy range is an important matter of debate. According to the widely accepted model of turbulence by Goldreich and Shridar [183,184], the key feature of the Alfvénic cascade is its *anisotropic* nature[3]. Since most of the power is transferred to scales perpendicular to a mean-magnetic-field direction, the Alfvénic waves may actually be highly inefficient in confining CRs, as discussed extensively in [185,186], and pitch-angle scattering onto magnetosonic modes may play the dominant role. A non-linear theory of CR scattering onto magnetosonic modes was presented in [187,188]. Very recently, after the seminal attempt presented in [189], the authors of [190] provided the first comprehensive phenomenological study of such theory, and showed how local CR data above the AMS break can be reproduced by solving the aforementioned diffusion equation with the diffusion coefficients computed *ab initio* from the theory, under reasonable assumptions on the free parameters involved. Recent radio observations support the notion that magnetosonic modes, under some circumstance, may drive CR transport [191].
- The **Monte Carlo approach.** The approach of solving the transport equation in a continuous setup where all the relevant terms are provided as smoothly varying function of the position provides a well-defined prediction of the expected *average* flux of cosmic rays. However, the stochastic nature of sources may play a relevant role in several cases, depending on the type of particle and on the energy range. For instance, high-energy leptons may be highly sensitive to this aspect, especially at energies at which the characteristic time and length scales associated to their momentum losses and spatial diffusion become comparable with the mean spatial and time distance between two different CR injection episodes. An important question is therefore to assess the expected *variance* of the CR flux, and a useful technique to attack this question is a Monte Carlo simulation. In this approach, many stochastic realizations are considered. In each of them, a random set of acceleration events in considered, and the CR flux from each event is typically evaluated by means of an analytic formula and added up. Some relevant examples of works based on this technique are [192–195]. The observables that are investigated are the fluctuations of the CR spectrum and normalization, anisotropy, chemical composition (especially around the knee).

To conclude, let us discuss briefly another broad class of modelling/data analysis methods that concerns the treatment of **populations of unresolved sources** detected by instruments in the form of diffuse emission that needs to be disentangled from interstellar emission. A first approach to this challenge is to develop source population synthesis models constrained

[3] For the sake of clarity, we remark here that, in this case, the anisotropy of the turbulent cascade is the key point. This is a different concept with respect to the anistropy of the CR transport phenomenon. In fact, as seen in our recap on the quasi-linear theory, an isotropic cascade of turbulent fluctuations can yield a strongly anisotropic CR transport.

by bright sources already detected, and then use them to predict the unresolved component, e.g., [196–198]. More recently some authors have proposed to use the non-Poissonian spatial fluctuations in photon counts to infer the properties of unresolved sources in data [199]. However, the susceptibility of this technique to systematic uncertainties in the modelling of interstellar emission seems to be sizable, therefore the results should be taken with caution [200].

3. Gamma-ray emission from the local interstellar medium: the Rosetta Stone of cosmic-ray astrophysics

Gamma-ray emission from local ($\lesssim$ 1 kpc) interstellar matter is a valuable source of information to link direct CR measurements with measurements of interstellar gamma-ray emission. If the CR population was fully uniform on these spatial scales in the ISM around the Sun, then CR measurements and gamma-ray measurements would be the expression of the same local interstellar spectrum (LIS) of CRs, and could be derived from one another based on the theories of the gamma-ray production processes and of solar modulation. In practice, given the uncertainties existing on all fronts, we can combine data and theories to get the best observational constraints and answer questions such as: how representative are direct CR measurements of the average LIS, and how much do CR densities and spectra vary within in the surrounding of the Sun and on which spatial scales?

The latter question is key to assessing the validity of some hypotheses often made in standard implementations of CR propagation models, namely that the sources are smoothly distributed in space and time, and that transport properties are homogeneous. If those hypotheses were true we would expect CR nuclei densities to vary mildly on spatial scales corresponding to the $\mathcal{O}$(kpc) diffusion lengths for the GeV-TeV CR population built up over durations $>$ 1 Myr. Non-uniformities in the local interstellar space therefore can inform us on the relevance of local effects on transport, and on the clustering in time and space of sources.

3.1. The emissivity of atomic hydrogen

The gamma-ray emissivity per H I atom $q_{\rm H\,I}$ is the most important observable quantity in this context. On one hand, H I is diffuse and dense enough that it can probe CRs on different spatial scales with a lesser influence from localized CR sources. On the other hand, the H I brightness temperature provides a direct estimation of gas column densities albeit the uncertainties on the optical depth correction (spin temperature).

Figure 2(**a**) and (**c**) show some recent measurements of the local H I emissivity $q_{\rm LIS}$ based on *Fermi*-LAT data. Casandjian [201] derived an average emissivity within a few hundred pc from the solar system by analyzing LAT data at intermediate Galactic latitudes ($10^\circ < |b| < 70^\circ$). We compare it to the H I emissivity for several local clouds in three complexes that sample the same spatial region from recent studies: Chameleon [202], the Galactic anticenter region [203], and the Orion-Eridanus superbubble [204]. These three studies have been selected among the numerous results on the emissivity of local H I [including also, e.g., 205–212] for a twofold reason. 1) They employ the same sophisticated methodology to separate emission from clouds along the line of sight and from different phases of interstellar gas, which should minimize biases in the template fitting procedure due to pile-up effects and variations of $X_{\rm CO}$ and dust specific opacity from cloud to cloud and within clouds themselves [203]. 2) They are distributed over different distances from the solar system between $\sim$150 pc and $\sim$400 pc, they lie in different directions opposite with respect to the Sun, and they span a large range of column densities and rates of star formation.

In spite of this wide range of locations and properties, the dispersion between the different emissivity estimates is only of 25%. The emissivity differences do not relate to the cloud altitude above the Galactic disc, nor to the cloud location with respect to the local spiral arm. Conversely, a sizeable fraction of the difference can be attributed to uncertainties in

the H I optical depth correction. The average emissivity $q_{\rm ref}$ was obtained for a uniform H I spin temperature of 140 K, while measurements from the individual regions are based on different uniform values ranging from 100 K to the optically-thin approximation (according to the best likelihood in the gamma-ray fit). When the comparison is made using the same or similar spin temperature values the differences are largely reduced [202,204]. Another noteworthy aspect is that the emissivity of H I is separated from other components using the template fitting technique with limitations due to modelling of other emission components and the angular resolution of the data. Therefore, at present we cannot claim any significant variations of $q_{\rm LIS}$ in intensity or spectrum in atomic gas within a few hundred pc around the solar system, and we conclude that the data constrain any fluctuations to $\lesssim 25\%$. In order to use gamma-ray data to probe for smaller fluctuations in the local emissivity spectrum and intensity it is crucial to reduce uncertainties in the derivation of interstellar masses, in particular in the H I optical depth correction. Constraints could also be strengthened in the future by reducing the uncertainties in the measurement of the emissivity thanks to data with improved angular resolution and by extending it to lower energies by means of one of the proposed MeV gamma-ray missions [107].

Figure 2(**b**) and (**d**) compare the average emissivity measurement $q_{\rm ref}$ to calculations based on estimates of the CR LIS inferred from CR direct measurements combined with models of solar modulation and other multiwavelength constraints. We calculated the Bremsstrahlung emissivity based on the CR electron and positron LIS derived by Orlando [97] using the GALPROP code tuned by using the most recent experimental results from *Voyager 1* [57] outside the heliosphere, from AMS-02 [54] above a few tens of GeV, as well as complementary constraints from radio, microwave, X-ray and gamma-ray emission, so that the estimate is independent of solar modulation models. As discussed by the author, the local emissivities measured by the LAT bring important constraints to the derivation of the all-electron LIS within their framework: we use the spectrum from the plain-diffusion (PDDE) model, which best fits the gamma-ray and multiwavelength/messenger data. We adopt the formulas for the Bremsstrahlung cross section given in [215] for electron kinetic energies > 2 MeV. We took into account target H and He in the ISM by assuming that the He/H fraction in the ISM is 9.6% [129]. Figure 2(**b**) shows that, for the all-electron LIS considered, Bremsstrahlung is relevant only below $\sim$1 GeV and is the dominant component below $\sim$100 MeV.

In the literature there is a controversy about the compatibility of direct measurements of CR nuclei and the emissivity $\gtrsim 1$ GeV measured by the LAT, and it is not always clear wether this depends on assumptions made on CR LIS and hadronic interaction cross-sections. Therefore, for hadronic emission produced in inelastic nucleon-nucleon collisions we have considered a few different models taken from the recent literature on the subject. Model A features the LIS of CR H, He, C, Al, and Fe derived in Boschini *et al.* [213] based on data from *Voyager 1* [57], PAMELA [216], AMS-02 [49,50], HEAO-3-C2 [217] and other instruments using the GALPROP and HELMOD codes to model Galactic propagation and solar modulation, respectively. The CR LIS are used to calculate the gamma-ray emissivity per H I atom using the H-H, H-He, He-H, He-He, C-H, Al-H, and Fe-H cross sections by Kachelrieß *et al.* [128], derived using the QGSJET-II-04m Monte Carlo generator, an update of QGSJET-II [218] taking into account the most recent accelerator data. The cross sections are only available above minimum energies ≥ 5 GeV depending on the interaction considered. For H-H we combined the results from QGSJET-II-04m at proton energies > 20 GeV with the nondiffractive part of the cross section model by Kamae *et al.* [123] at lower energies owing to the result that this combination best reproduces experimental low-energy accelerator data [125] . For other interactions, below the minimum energy available in Kachelrieß *et al.* [128] we rescaled the H-H cross sections by the ratio of the cross section at the minimum energy available over the

H-H cross section at the same energy per nucleon. We assumed again that the He/H fraction in the ISM is 9.6%.

Model B features the same gamma-ray production models but different H and He LIS derived by Corti *et al.* [214] based on a broadly overlapping set of experimental data, but a much simpler approach which consists in seeking a parametrized formula to encode the LIS and treating solar modulation by a modified rigidity-dependent force-field approximation. The resulting CR LIS differ by at most 10% w.r.t. those by [213] in the rigidity range 1-10 GV where solar modulation still plays a significant role but there are no direct constraints on the LIS from *Voyager*. This translates to a $\lesssim$ 5% difference in the resulting gamma-ray emissivity spectrum below a few GeV after convolution with the gamma-ray production cross-sections. An additional source of difference between models A and B is that in model B we neglect the interactions from CR species heavier than He, which contribute ∼1%-3% of the emissivity in model A.

Model C employs the same CR H and He LIS spectra as in model A, but different gamma-ray production models, namely the H-H cross sections by Kamae *et al.* [123], including the diffractive part, and the scaling factors by Mori [129] based on the DPMJET-3 Monte Carlo generator to scale the H-H emissivity in order to account for heavier elements, namely CR He nuclei and He, CNO, Mg-Si, and Fe in the ISM. The scaling factors are available only for CR energies of 10 GeV/nucleon, therefore we neglect their variations with energy, that should be modest at least in the energy range $>$ 10 GeV [130] but are expected to become more relevant in the less understood energy range $<$ 10 GeV (see Figure 1 of [129], which only shows the results down to energies of 5 GeV/nucleon, below which results become inaccurate). We note that the contribution from target nuclei heavier than He, not accounted for in models A and B, represents ∼2% of the total emissivities in model C.

To summarize, in the hadronic-dominated energy range $>$ 100 MeV we find variations between different predictions of the emissivity based on direct CR measurements of $\lesssim$ 10%, more important at lower energies. The differences are comparable to the uncertainties in the most precise emissivity measurements and a factor of ∼2 smaller than the dispersion between measurements for different individual clouds.

For gamma-ray emission in the energy range $>$ 10 GeV, where the differences between alternative estimations of the CR LIS and between alternative hadronic production models are small, the average emissivity $q_{\rm ref}$ exceeds the predictions by 20-30%. This broadly agrees with several independent results in the literature [97,116,219]. The recent precise measurements by AMS-02, in particular of He [50], rule out earlier claims [201] that the average emissivity measured by the LAT is consistent within uncertainties with direct CR measurements as pointed out by Orlando [97]. We also stress that the evaluation of the contributions from species heavier than H must take into account these measurements, which, for instance, is not necessarily the case when using *enhancement factors* taken from the literature such as the one provided by [129]. The latter aspect is discussed in greater detail by [130].

However, the difference between the predictions and the average emissivity is comparable to the dispersion between different clouds, that, as discussed above, is largely imputable to uncertainties in the H I optical depth correction performed assuming a uniform spin temperature and possibly other analysis features. Once the dispersion of ∼25% among different regions and clouds is taken into account, regardless of its interpretation as a systematic uncertainty related to the extraction of the emissivity or as an effect of real fluctuations in CR densities on spatial scales smaller than a few hundred pc, for the moment it should still be considered as viable that direct CR measurements are in agreement with the local H I gamma-ray emissivities.

Therefore, also to connect direct CR measurements and gamma-ray observations of the local ISM it is crucial to reduce uncertainties in the extraction of the emissivities and

derivation of interstellar masses. At the same time, to reduce uncertainties in the derivation of the gamma-ray emissivities from direct CR measurements it is paramount to improve our knowledge of the gamma-ray production cross sections, especially at low energies reaching the kinematic threshold for pion production where currently only approximate scaling of H-H is readily accessible to account for interactions of heavier species. To this end it would be extremely useful to pursue experimental accelerator campaigns going beyond those already undertaken, to further improve Monte Carlo generators, and to make more results [e.g., 127] publicly available in the form of parametric/tabulated cross sections for gamma-ray production.

3.2. Molecular clouds and their spectra

An alternative approach consists in studying gamma-ray emission from molecular clouds. Molecular clouds provide localized (few tens of pc) targets for CR interactions with large masses. However, with observations of molecular clouds two additional classes of complications arise.

1. The column densities of molecular gas are traced only indirectly, most often via the CO lines or dust thermal emission. On one hand, the $X_{\rm CO}$ ratio is empirically observed to vary by a factor of $\sim$2 between different local clouds depending on their diffuseness [203]. Furthermore, based on numerical simulations $X_{\rm CO}$ is expected to vary significantly within individual clouds between the inner part that is more self-shielded and the outskirts where CO is more easily photodissociated by UV photons [220]. On the other hand, based on gamma-ray and multiwavelength data the conversion factor from dust thermal emission to gas column density is now well-established to vary by a factor of $\sim$3 from the low-column densities in atomic gas to cold and dense gas in molecular cloud cores probably owing to evolution of the dust grain properties (chemical composition due to irradiation and structure due to irradiation itself, but also mantle accretion, coagulation, and ice coating) [77,203]. Therefore, uncertainties of a factor of a few affect the estimates of the absolute level of CR densities from gamma-ray emission of molecular gas, which appears better suited to probe the spectral shape of CRs.
2. Two different kinds of physical phenomena can affect the spectrum of the CR populations around molecular clouds so that it is no more representative of the large-scale population. On one hand, molecular gas is associated with star formation, which in turns triggers all the phenomena expected to accelerate particles efficiently. Therefore, freshly-accelerated particles can be observed in this environment (see Section 5). In the literature molecular clouds free from the influence of nearby accelerators are often referred to as *passive clouds*, but establishing which clouds belong to this category observationally is challenging. On the other hand, several effects may alter the propagation of CRs in molecular clouds in an energy-dependent fashion. Increased ionisation losses were the first mechanism considered historically that could suppress CR densities below 300 MeV kinetic energies [221], but nowadays they are not considered by themselves the most important effect. Streaming instabilities driven by CR pressure gradients can alter the CR transport regime from diffusive to advective and so suppress the CR pressure by $\lesssim$ 10% at GeV energies [222]. The peculiar magnetic field configuration in molecular clouds via magnetic mirroring coupled with energy losses can lower the CR densities by a factor 2-3 in molecular cores in the MeV energy range, while CR enhancements due to magnetic focussing appear a less important effect [223]. Damping of magnetic turbulence by ion-neutral friction coupled again with energy losses can reduce the CR fluxes below 100 MeV kinetic energies, especially for electrons [102]. We note that all propagation effects mainly affect CRs at energies $<$ 1 GeV, for which constraints from gamma-ray observations are looser.

To this date there is a controversy in the literature about the uniformity of the spectral shape of gamma-ray emission from molecular clouds in the local interstellar space as observed with the *Fermi* LAT. Although most studies agree on the uniformity of the spectra and their compatibility with the spectrum of H I, Yang *et al.* [224] claimed low-energy CR enhancements in the Orion A, Orion B, and Chameleon clouds, not confirmed by subsequent studies [202,225]. More recently, Baghmanyan *et al.* [226] claimed spectral deviations with respect to the LIS for the molecular clouds in the Aquila rift, rho Ophiuchi, and Cepheus. For the latter two the results are at odds with previous studies based on smaller datasets [206,210,225]. The devil seems to lie in the detail, that is, how the different studies account (or not) at the analysis level for variations of CR densities in different structures along the line of sight and in the Galactic background, and variations/uncertainties in $X_{\rm CO}$ and/or dust-to-gas conversion factors. Analysis assumptions about all of these aspects combined with the energy-dependent PSF of the LAT may easily produce distortions in the derived spectra via cascade effects in the template fitting procedure. The claims of spectral variations therefore require further investigation to asses their robustness.

By stacking the spectra of several nearby molecular clouds measured by the *Fermi*-LAT Neronov *et al.* [225] could highlight the existence of a break in the CR nuclei spectrum at a rigidity of 18^{+7}_{-4} GV, which is consistent with the direct measurements from *Voyager 1* [57] and AMS-02 [49,50] in a rigidity range where the impact of solar modulation is most relevant. Neronov *et al.* [225] speculated that this coincides with the transition from the Galactic-scale steady-state population observed throughout the disk of the Milky Way (see Section 4) to a local population of CRs driven by stochastic injection around the Sun localized in space and time. We note, however, that uncertainties in the hadronic cross-sections have not been taken into account in this work.

First results at TeV energies were recently published by the HAWC collaboration [227]. No detections are reported for a set of seven nearby molecular clouds. Upper limits on their gamma-ray fluxes are less than an order of magnitude larger than expectations based on the extrapolation of direct measurements by AMS-02. For a stacked analysis assuming a power-law CR spectrum with index 2.7 the upper limit on the CR density is approximately at the level predicted by direct measurements. This implies that, if the CR spectrum follows a simple extrapolation of what is directly measured at lower energies, HAWC should reach a detection of nearby clouds by doubling the exposure, or even faster taking into account the ongoing detector upgrades. Complementary results can be expected from LHAASO and, eventually, from SWGO (although most of the nearby molecular clouds in the Gould Belt are visible from the Northern hemisphere).

4. Large-scale interstellar gamma-ray emission: tracing cosmic rays throughout galaxies

The large-scale distribution of CRs in galaxies encodes information on their injection sites and spectra, and on the transport mechanisms and their interactions with other components of the ISM. For a long time it has been known that gamma-ray data point to the existence of a large-scale population of CRs in the Milky Way disk with properties similar to that observed near the Earth, see, e.g., [228], and of CR populations with somewhat different properties in the Magellanic Clouds [12,13]. Recent years have seen impressive developments in GeV observations for the Milky Way and local-group galaxies. Their CR populations show diverse and sometimes unexpected properties. Variations in the densities and spectra of CRs across the galactic disks and halos with departures from expectations based on standard implementations of CR models, combined with ubiquitous unexplained residual features, are questioning our understanding of particle transport and its microphysical foundation. At the same time, the first sub-MeV and TeV observations have been opening new and complementary windows to constrain the large-scale CR distribution for the Milky Way.

4.1. Cosmic-ray distributions through the disks of galaxies

4.1.1. The Milky Way

A lot of recent progress is based on data from the *Fermi* LAT. A first avenue to infer the large-scale CR distribution from the observations is through the emissivity of interstellar gas, that can be extracted using the template-fitting technique for multiple regions along the line of sight thanks to the distance proxy provided by the Doppler shift of atomic or molecular lines (Section 2.4). This approach has been applied to two regions towards the outer Galaxy in the second and third Galactic quadrants for which the separation of different structures and spiral arms along the line of sight is remarkably good and free from ambiguity [206,207]. These studies strengthened the observational evidence for the long-known *gradient problem* [228], i.e., the fact that the gamma-ray emissivity only mildly decreases from the position of the Sun up to Galactocentric radii of $\sim$15 kpc, while the number density of putative CR sources shows a steep decline. Furthermore, in the third quadrant the data allow only a decrease $< 20\%$ in the low-density region between the local and Perseus arms, ruling out a strong coupling between CR and ISM densities invoked by some early modelling efforts [229].

The distribution of CRs throughout the entire Galactic disk was derived by several authors by analyzing LAT data for the entire sky with gas templates built by separating H I and CO in Galactocentric rings [230–232] and using the H I emissivity to infer the underlying CR spectra. These studies show that the CR proton density above 10 GeV varies by a factor of a few across the disk of the Milky Way, with a peak at $\sim$4 kpc from the Galactic center and a mild decrease as a function of radius beyond 5 kpc, which confirms and extends the trend inferred from the dedicated studies of the outer Galaxy. More surprisingly, LAT data have revealed a progressive hardening of the CR proton spectrum toward the inner Galaxy, with the power-law spectral index at 10 GeV going from the local value of $\sim$2.7 near the Sun to $\sim$2.5 at a few kpc from the Galactic center. For these studies, as well as for the outer Galaxy, a major source of uncertainty is due to the H I opacity correction (spin temperature), which is associated with a $\sim$30% systematic uncertainty in the H I emissivities [207,230]. Another important limitation is the possible contamination by diffuse emission from populations of unresolved sources. The fraction of unresolved sources in the total diffuse emission is estimated to be 3% at 1 GeV [233]. We expect this fraction to increase as a function of energy, but based on reasonable assumptions about the source populations the conclusions on the general trends in CR densities and spectra inferred from LAT data remain unchanged [232]. Other sources of uncertainty include the modelling of IC emission and detected gamma-ray sources.

Furthermore, the impact of the assumption that CR densities are axisymmetric, and, in particular, that the near and far region within each ring for radii smaller than the solar circle share the same CR densities, needs to be checked against observations not affected by the kinematic ambiguity. A complementary approach that may overcome this limitation is the use of well-localized targets. Aharonian *et al.* [234] have derived the emissivity of nineteen giant molecular clouds located at Galactocentric distances up to 12 kpc. The trend in inferred CR proton densities as a function of Galactocentric radius is generally consistent with earlier works based on ring emissivities [230,231], but the large uncertainties due to the separation of the clouds from foreground/background gas and the conversion of CO intensity to H_2 column density makes the results too uncertain to draw robust conclusions. A few clouds show spectral deviations from the general trend that may point to localized CR excesses, e.g., due to a nearby accelerator. More recently the same authors analysed LAT observations of nine clouds located at Galactocentic distances of 1.5-4.5 kpc employing dust opacity (inferred from thermal emission) as total gas tracer (no separation along the line of sight), and they obtained results at odds with the ring analyses [235]. However, we warn the readers that the latter results are based on the assumption of a constant dust specific opacity across the

Milky Way with an uncertainty of 20%, while local clouds show that variations of a factor ∼3 related to evolution of the dust grain properties are possible (Section 3.2), and, moreover, an increase of the dust specific opacity of a factor of a few in the inner Galaxy is expected from the correlation of dust-to-gas ratio and metallicity gradient as a function of galactocentric radius observed for external galaxies [236–238]. Different measurements of the CR proton density and spectrum across the Galactic disk are summarized in Figure 3.

A second avenue to constrain the large-scale CR distribution consists in comparing the data directly to the outcome of predictive models. Ackermann *et al.* [239] compared predictions by GALPROP to the LAT data for the entire sky by varying hypothesis on the astrophysical input such as CR source distribution or $X_{\rm CO}$. The results are consistent with the trends seen in the template analyses, notably the flat CR profile in outer Galaxy, and higher/harder emission toward the inner Galaxy. Although the data/model agreement is reasonable and demonstrate that standard implementations of Galactic CR propagation models describe the gamma ray data within ∼30%, regions of extended residuals appear for any of the models considered. Residual features will be discussed later in Section 4.3. The study [239] also demonstrates the high level of degeneracy between different inputs to predictive models, and therefore the importance to use different complementary approaches to analyse and interpret the data.

Predictions from GALPROP were also compared to data from *INTEGRAL* SPI, which reveals the existence of diffuse emission from the inner 60° of the Galactic disk in the soft gamma-ray band from 20 keV to 2 MeV. The diffuse gamma-ray emission above 60 keV is consistent with a dominant origin from inverse-Compton scattering of CR electrons , and connects spectrally with emission measured at higher energies by COMPTEL and pair-conversion telescopes [240–242]. These data tend to favor models with an important contribution from secondary electrons and positrons, which is not necessarily the case for the electron spectrum measured near the Earth [97]. A new space mission for MeV astronomy holds the potential to deepen our understanding of the large-scale properties of IC emission and bridge *INTEGRAL* and *Fermi* observations at a sensitivity largely improved compared to COMPTEL [107].

Recently we gathered the first measurements of diffuse emission in the energy range from hundreds of GeV to 1 PeV in different regions of the Galactic disk thanks to Milagro [243], H.E.S.S. [31], ARGO-YBJ [244], HAWC [245], and Tibet ASγ [246]. In this energy range the key challenge, beside separating diffuse gamma-ray emission from the charged-particle background, is to disentangle the interstellar component from the contributions from unresolved source populations. For example, Steppa and Egberts [198] estimate that ∼30% of the diffuse emission measured by H.E.S.S. can be attributed to unresolved source populations, and Amenomori *et al.* [246] estimate the same fraction to be 13% for the measurement at energies above 100 TeV with Tibet ASγ. While the measurements generally exceed expectations based on the local CR spectrum and there is now firm evidence of emission from CRs up to the knee, there are tensions between observations with different instruments [247] and better constraints on the unresolved source contribution become key to use TeV data to investigate CR properties such as the spectral hardening in the inner Galaxy [248]. At the same time, upcoming measurements with CTA and LHAASO [249,250], and, possibly, SWGO will provide much improved sensitivity, and enable an even higher complementarity with neutrino measurements.

4.1.2. Implications of the gradient problem and inner-Galaxy hardening

The gradient problem and inner-Galaxy hardening observed by *Fermi* have been stimulating a lively debate on our understanding of CR transport in the Milky Way. As far as the gradient problem is concerned, the assumption of a very extended ∼10 kpc diffusive halo seems to alleviate the discrepancy. However, this solution is in tension with the most recent observations of CR isotopic abundances and, possibly, radio/gamma-ray emission from the

Milky Way, as discussed in Section 4.2, and also covered for instance in Evoli *et al.* [176] and references therein. Also, ad-hoc assumptions on the distribution of CR sources may contribute to mitigate the problem, and suffer from similar problems when confronted with catalogs of SNR and pulsar, that are expected to trace the radial distribution of the CR injected power.

A promising attempt to solve the gradient problem based on non-homogeneous diffusion, also in connection with the anisotropy problem (i.e. the larger dipole anisotropy predicted by standard implementations of the Galactic CR transport model with respect to the observed one) was presented in Evoli *et al.* [176]. In that paper, a correlation between the CR source density and the normalization of the diffusion coefficient is invoked. The proposed solution (visualized in Fig. 3, upper panel, "non-homogeneous diffusion") stems from the following physical argument: a larger turbulence level is expected in the regions of the Galaxy characterized by a larger density of CR accelerators, in particular, along the Galactic plane, in the range of Galactocentric radii close to the so-called molecular ring. Given the topology of the large-scale regular magnetic field, and the overall geometry of the problem, the CRs accelerated in the Galactic plane mainly escape in the vertical direction, perpendicular with respect to the regular field. Given the increase of the perpendicular diffusion coefficient with increasing turbulence level that is highlighted in several numerical simulations (see for instance [254–256]), the aforementioned correlation naturally follows from these considerations.

As far as the hardening problem is concerned, a phenomenological model where the trend is obtained as a result of a smoothly varying scaling of the diffusion coefficient with respect to rigidity was presented in Gaggero *et al.* [251] (see Fig. 3, lower panel, "non-homogeneous diffusion"). A physical interpretation of this trend in terms of specific aspects of transport physics was recently presented in Cerri *et al.* [178]. In this analysis, the argument is once again based on the nature of perpendicular escape and the geometry of the magnetic field. The starting points are the following considerations: *(i)* the numerical simulations that aim at characterizing CR transport in pre-exisisting (Alfvénic) turbulence, already mentioned above, suggest a different scaling with rigidity as far as parallel and perpendicular diffusion coefficients are concerned, with the parallel transport featuring a harder rigidity dependence; *(ii)* the state-of-the-art models of the large-scale Galactic magnetic field (see for instance [91]) seem to suggest the presence of an X-shaped poloidal component in the inner Galaxy; hence, the vertical escape of CRs may be parallel in the inner Galaxy, and perpendicular in the outer Galaxy, where the field is expected to follow the spiral pattern on the Galactic plane, with a less prominent vertical component. These facts imply a progressively harder scaling of the propagated CR spectral index, as simulated by [178] with an axisymmetric, fully-anisotropic version of the DRAGON code.

Following a very different line of thought, another recent work [252] attempts to explain both the gradient and the spectral hardening problems at the same time. In this work the non-linear effects mentioned in Section 2.4 are exploited to provide an explanation to both anomalies. The idea is that the regions with a larger density of accelerators feature a larger CR gradient. Hence, the turbulence growth rate associated to streaming instability is larger: as a consequence, the non-linear phenomenon of CR self-confinement is greatly enhanced. In the GeV domain CR escape is actually the result of the competition between rigidity-dependent (possibly self-generated) diffusion and rigidity-independent advection. The more efficient self confinement implies a lower diffusion coefficient associated to self-generated turbulence. Hence, advection takes over up to larger rigidities, and the propagated spectrum is less affected due to the energy-independent nature of this process. Hence, the inner regions of the Galaxy are expected to feature a harder CR spectral slope closer to the one initially injected in the ISM. A careful numerical treatment of this problem actually shows that the gradient problem may also be fixed, provided that the magnetic field is assumed to drop exponentially

at large radii. The model is represented in Fig. 3, both lower and upper panel ("non-linear transport").

The interpretation of the spectral hardening in terms of the interplay between advection and non-linear CR confinement has an important consequence. This kind of solution is valid only in the low-energy range where the streaming instability plays a major role, and cannot be invoked if the spectral trend were to be clearly confirmed at rigidities $\gtrsim$ 100 GV. The current data do not allow to reach a firm conclusion on this point. However, the analysis presented in [232], already discussed above, seems to point towards the presence of a spectral hardening even in high-energy *Fermi*-LAT gamma-ray data. On the other hand, the explanations based on the different scaling relations of perpendicular and parallel transport are expected to hold at all energies.

The different explanations of the inner-Galaxy hardening are expected to result in different phenomenological implication for the multi-TeV gamma-ray interstellar emission. In particular, a harder hadronic CR spectrum towards the Galactic Center that extends up to the multi-TeV domain (not covered by *Fermi*-LAT data) would imply a significantly larger diffuse gamma-ray emission from the inner Galactic plane. Different scenarios were put forward in order to bracket the uncertainties in this context, mainly due to the extrapolation of the analyses based on Fermi-LAT data, and the most "optimistic" prediction were shown to saturate the diffuse emission from the Galactic Ridge measured by H.E.S.S., leaving less room for the contribution from a central accelerator [257]. These scenarios featuring the hardening in the multi-TeV domain are compatible with the interpretation based on the idea of anisotropic transport, and show some tension with the interpretation that stems from the non-linear propagation models, given that self-generated diffusion models predict a transition to transport dominated by background turbulence at $\sim$100 GeV, while, as mentioned before, the peculiar scaling relations associated to anisotropic transport extend to larger energies. However, more data and further studies on the modelling sides are needed. Currently operating and forthcoming air shower and Cherenkov experiments (in particular LHAASO, HAWC, SWGO, and CTA) will help to shed light on this issue. In particular, if the presence of a harder spectrum in the inner Galaxy were to be confirmed in the TeV-PeV range, the interpretation of this effect in terms of the interplay between advection and self-confinement would be disfavored. On the modelling side, as pointed out for instance in [258], more advanced simulations of the multi-TeV gamma-ray emission from the Galactic plane will be needed, also taking into account the crucial effect of absorption that plays a dominant role especially above 50-100 TeV.

We remark that, if confirmed, the presence of a progressive spectral hardening of the hadronic CR population in the inner Galaxy has also interesting phenomenological consequences in a multi-messenger context. In fact, as pointed out for instance in [259], phenomenological models that reproduce this trend by featuring a radially-dependent slope of the diffusion coefficient predict a neutrino flux in the inner Galaxy that is 2-5 times larger compared to conventional predictions, based on a constant spectrum across the Galactic plane (see also [260]). This may explain up to 25% of the neutrino events measured by IceCube, and may suggest that a detection of a positive correlation of the IceCube events with the Galactic plane might be just around the corner. This hypothesis was extensively tested by the ANTARES and IceCube Collaborations. The recent analysis [261] provided joint constraints that start to challenge this scenario. The constraints on a Galactic component are based on 10 years of ANTARES showers and tracks (218 showers in total), and 7 years of IceCube tracks (730130 events with 191 events expected from the optimistic Galactic model). The results are in mild tension with the most "optimistic" versions of these phenomenological models, and allow for the possibility to test this kind of prediction in the near future. A more recent analysis [262] based on seven years of IceCube cascade data (characterized by an interaction vertex inside the detector) outlined a 2σ hint for a Galactic component consistent with the

optimistic models. Future studies are needed to shed light on this issue. A firm detection of a Galactic neutrino component would represent a remarkable confirmation of the presence of a hard hadronic CR population extending above the TeV domain, and would greatly help in shedding light on the microphysics processes that originate this anomaly.

4.1.3. Local-group galaxies

Moving to external galaxies, the LMC is the best target when it comes to studying how gamma-ray emission connects to components of a galactic ecosystem due to its proximity (distance of $\sim$50 kpc) and a favourable geometry (disk-like structure with a low inclination angle $\sim 30^\circ$). The emission at 1 GeV is dominated by radiation seemingly correlated to the gas disk, as could have been expected based on what we see in the Milky Way. The gamma-ray distribution in the disk could be fitted assuming an emissivity profile decreasing by a factor 2-3 from the center to the outskirts, with a peak value of about 30% of the emissivity measured in the Solar system neighbourhood [263]. This lower value is thought to arise from the smaller size of the LMC, and a corresponding smaller confinement volume. Again, this is very reminiscent of the Milky Way (see above in this section). More surprising, though, is the fact that the emission at 10 GeV is contributed at about 50% by extended components of unknown origin but not evidently correlated with gas or recent star formation, therefore implying localized enhancements in the CR densities by factors of 2 to 6, or an alternative explanation not related to interstellar emission [263]. To date, the origin of such features is still unexplained, and revisiting the gamma-ray emission from the LMC from the twice larger Fermi-LAT data set available now and increased exposure with H.E.S.S. would be timely.

The SMC is another promising target due to its proximity (distance of $\sim$62 kpc) but the geometry of the galaxy is much more intricate, with an irregular shape elongated along the line of sight over 20 kpc [264] that complicates the interpretation of observations. Significant gamma-ray emission is detected but no obvious correlation with gas or star formation is observed [265,266]. If interpreted anyway as interstellar emission, the flux observed implies an average CR density of about 15% of the emissivity measured in the solar system neighbourhood. However, it was estimated that the measured flux could be accounted for to a large fraction by an unresolved population of pulsars, which would imply an even lower CR density [267].

At a much larger distance of 785 kpc, another target of choice is M31 (Andromeda) that, as a grand design spiral galaxy, resembles more closely our Milky Way and may allow a more direct analogy. Extended emission from the galaxy is detected but the signal is confined to the inner regions, within 5 kpc from the center. It does not fill the disk of the galaxy and in particular does not correlate with the regions rich in gas or star formation activity that are mainly located in a large ring at 10 kpc distances from the center. Emission from the disk at large is however not strongly excluded and may be present at a level of up to 50% of the currently detected flux [268]. The different gamma-ray emission distribution in M31, compared to the Milky Way, can be interpreted as resulting from its global properties: the star formation rate in M31 is about 10 times lower than in the Galaxy, which would decrease the contribution from the star-forming disk, while it has a 5-6 times more massive bulge, which would enhance the contribution from old stellar populations gathered in the central regions.

4.2. Cosmic-ray distributions through the halos of galaxies

As briefly described in Section 2.3, within the standard galactic CR paradigm it is assumed that particles are confined diffusively in a magnetized halo with a size of a few kpc for the Milky Way. Yet, the formation of the halo and the escape at its boundary are still poorly understood, and usually modelled by imposing free escape at a fixed height z_{max}. The value of z_{max} is then treated as free parameter in CR models adjusted to reproduce CR

elemental/isotopic abundances as well as the spectra of CR species measured locally. Recent measurements with AMS-02 and other instruments in this framework point to values of $z_{max} = 5$ kpc with sizeable uncertainties of a few kpc [195,269].

Until recently, gamma-ray and radio emission from CR interactions was used to constrain the propagation in the Milky Way halo only indirectly via aggregate properties (longitude, latitude, and radial profiles and spectra), which is tricky due to severe degeneracies with other unknowns such as the source distribution, e.g., [93,239]. Conversely, kpc-wide synchrotron halos around external edge-on galaxies have been observed in radio for almost thirty years, e.g., [270].

The last decade has seen two major advances in the understanding of the halo and related observational constraints especially from gamma rays: more direct observations in gamma rays thanks to observations with the LAT of clouds at large distances from the Galactic plane, and, possibly, of the halo of M31; the emergency of models of galactic CR halos based on the microphysics of magnetic turbulence and particle transport.

The improved sensitivity of the LAT made it possible to measure the emissivity from atomic clouds at large distances from the disk. Those include notably a set of high- and intermediate-velocity clouds which span heights from a few hundred pc to several kpc [271] with robust distance determinations based on stellar brackets [272]. Their emissivities, shown in Figure 4, testify to a significant decrease of CR densities as a function of distance from the Galactic plane. On one hand, this represents a new and robust observational test of the Galactic origin of CRs below the knee. On the other hand, the emissivity measurements can be compared to predictions from CR propagation models and directly constrain the CR gradient in the halo. A fit to these measurements yields a z_{max} value of $\sim$2 kpc with an uncertainty of a few kpc [271]. In particular, the upper intermediate-velocity Arch at a height of 0.7-1.7 kpc above the disk was found to have an emissivity $< 45\%$ relative to the local value at 95% confidence level.

Even more recently, Joubaud *et al.* [204] reported an emissivity of 0.657 ± 0.031 with respect to the local value for the Eridu cirrus at a modest altitude of 200-250 pc (also shown in Figure 4). We note that the distance to the Eridu cirrus is based on dust reddening [273], a method more indirect than the use of stellar brackets. For the moment it remains unclear whether this observation should be interpreted in terms of the large-scale vertical gradient of CRs or a peculiar magnetic field configuration in this cloud [204].

It is therefore essential to study in gamma rays a broader sample of clouds at large distance from the disk, to map the large-scale distribution of CRs and probe correlations with localized magnetic structures and outflows, especially in the key altitude range between a few hundred pc and a few kpc. Robust estimates of their distances, e.g., using brackets based on Gaia data [274], would be of great help to aid the interpretation of the gamma-ray data in term of CR gradients. Furthermore, we note that observations of emission from the halo of the Milky Way may be corroborated in the near future by observations of the nearby edge-on Andromeda galaxy: claims of detection of gamma rays from its halo already exist, but for the moment it remains unclear whether it is due to CR interactions, either in the form of an escaping CR flux interacting with the intergalactic medium or of nonthermal lobes analogous to those observed in galaxies with an active nucleus, or it is of exotic nature related to hypothetical dark-matter particles annihilations [275–277].

From the theoretical point of view the last few years have shown a renewed interest in treating CR propagation in the halo on a more physical ground rather than relying on imposing a boundary condition at a height z_{max} and a diffusion coefficient adjusted to reproduce the data. Several attempts have been made to explain CR diffusion as a result of the non-linear interaction with plasma waves self-excited owing to the CR streaming instability [5,278–280], as briefly introduced in Section 2.4. In particular, Evoli *et al.* [182] discussed a

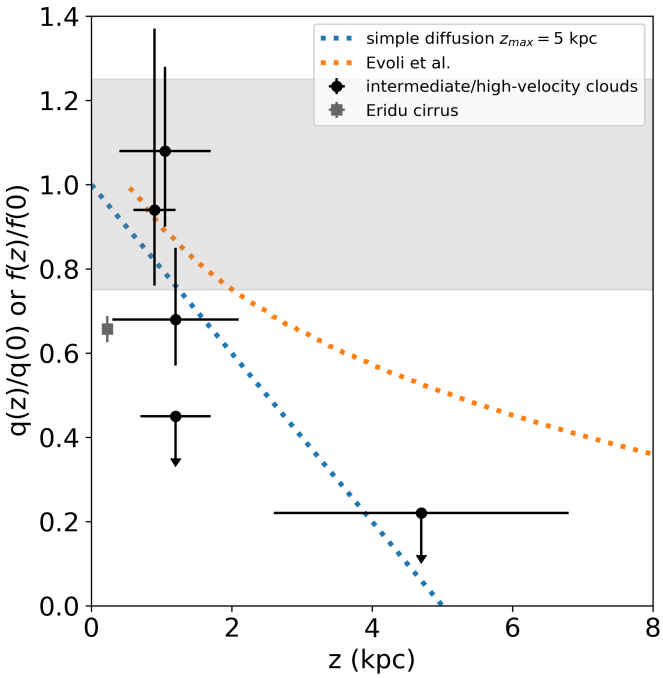


Figure 4. Vertical gradient of gamma-ray emissivities or cosmic-ray densities in the Milky Way halo. The points correspond to emissivity measurements derived from *Fermi* LAT data for intermediate- and high-velocity clouds with distance brackets based on stellar probes [271] and for the Eridu cirrus [204]. The horizontal band shows the 25% dispersion of emissivity measurements for nearby clouds (Section 3). The lines correspond to CR densities from two models: the z-dependent part of the solution of the diffusion equation in the plane-parallel geometry (infinitely thin Galactic plane) with a uniform source distribution when only ionization losses are assumed for the halo height $z_{max} = 5$ kpc inferred from recent direct CR measurements [195,269] and the model by Evoli *et al.* [182] which describes CR vertical propagation based on a mixture of advected turbulence and of CR self-generated waves (we show the CR densities at 10 GeV which are most representative for the gamma-ray energies considered).

scenario in which the diffusion properties of CRs are derived from a combination of wave self-generation and advection from the Galactic disc, with a halo of a few kpc naturally arising as a consequence. All of these models are non-linear, and explore the feedback from CRs on galaxy evolution via the formation of winds and the subsequent impact on star formation and galaxy evolution, while succeeding to reproduce to some extent the observed properties of primary CRs. Predictions by [182] are shown for illustration in Figure 4. Their model overpredicts CR densities at a few kpc from the disk compared to gamma-ray emissivities. However, the tuning of the model parameters did not take into account gamma-ray measurements, which serves as a nice illustration of the importance of halo emissivity measurements in this context. Further observables that can complement the halo gamma-ray emissivities are the isotropic gamma-ray background and the flux of high-energy neutrinos [277,281,282].

4.3. New frontiers: residual gamma-ray emission

One of the most interesting and surprising results from *Fermi*-LAT observations has been that extended residual emission as large as ∼30% on a variety of different scales appears on top of large-scale interstellar emission from the Milky Way as accounted for via either template fitting or standard implementations of CR propagation models [230,283]. There are

clear indications that large-scale CR trends described in the previous sections are not sufficient to capture the richness of the CR phenomenology in the Galaxy, which also mirrors the gamma-ray enhancements observed in the LMC (Section 4.1). In other words, the sensitivity and breadth of gamma-ray observations is now such that we may have reached the limits of standard basic modelling approaches.

Some of the features emerging in the residuals are considered amongst the most important results from *Fermi*. The *Fermi* bubbles are a large bipolar structure seemingly emanating from around the Galactic center [284–286], which has been interpreted, for instance, as the result of a hadronic wind that advects particles out of the Galactic disk, or of a leptonic jet accompanied by anisotropic diffusion along magnetic field lines that drape around the bubble surface (for a review, see, e.g., [287]). The Galactic center excess is a large feature with approximate spherical symmetry around the center of the Galaxy, which has attracted a lot of attention owing to the possible interpretation as the result of annihilation/decay of hypothetical dark-matter particles, but that could also find a more mundane explanation, for example in terms of unresolved populations of sub-threshold gamma-ray sources such as millisecond pulsars (see, e.g., [288] and references therein). Observations and interpretation of these features are already covered in full reviews by their own, included those just referenced, so they are left out from our paper.

Aside from speculations on possible exotic phenomena such as radiation from dark matter, three main families of solutions to the puzzle of residuals have been proposed, invoking either (*i*) contributions to diffuse emission from unresolved gamma-ray sources, (*ii*) undetected or poorly modelled interstellar gas and radiation fields acting as target for CR interactions, or (*iii*) the effect of CR injection in the ISM localized in space and time possibly accompanied by peculiar conditions of particle transport. While in the context of this review we will be mainly concerned with the latter, let us remark that a mix of the three is the most likely explanation for the observations. We will cover in detail observations of gamma-ray emission in the vicinity of sources and their implications for CR transport in the next Section.

However, the discretized nature of CR sources may explain also large-scale residual emission. In a recent work, Porter *et al.* [289] have assessed how discrete and steady-state CR injection differs when it comes to predict gamma-ray interstellar emission. Even using simplified prescriptions for discrete injection, i.e. no localized self-confinement around sources and no spatial and temporal clustering from OB associations, the work illustrates the wealth of effects that can be expected. Compared to the corresponding steady-state case, stochastic injection can give rise to intensity features with sizes from a few to a few tens of degrees, both excesses and deficits, with amplitudes reaching 50% and above of that predicted in the steady-state case. The effect is more pronounced at larger energies, at intermediate/high longitudes and latitudes, and for leptonic radiation processes, which constitutes interesting challenges for the new and future VHE instruments that will provide extended coverage of the Galactic plane and give access to larger angular scales (e.g., HAWC, LHAASO, or CTA). In that endeavour, one should keep in mind that simply extracting excess or residual emission in a reliable way may be challenging. Porter *et al.* [289] illustrate the biases that can result from using a reference interstellar emission model based on a mismatched steady-state smooth source distribution (e.g. including spiral arms or not). Very large scale emission structures can ensue, some of which reminiscent of the *Fermi* bubbles. In addition, isolating excess emission from residual significance maps can lead to various troubles, such as the splitting of the true component into various substructures and a biased determination of other emission components. The most promising avenue to address such issues is to incorporate multiwavelength information and try to constrain the CR injection and propagation history from a large set of observations, from radio/microwave to X-rays and gamma rays.

5. Gamma-ray emission in the vicinity of sources: the early steps of a long journey for cosmic rays

Fully understanding the life cycle of galactic CRs requires connecting the properties of particles still confined within their sources and/or undergoing acceleration to the characteristics of a galactic population built up over Myr time scales from thousands of transient accelerators of various subclasses. Recent observations highlight that the early phases in which CRs are still wandering around their original source play a special role within this life cycle, and have a remarkable impact on the associated non-thermal emissions. There is no unambiguous terminology for this particular moment in the CR life cycle: it is sometimes referred to as the escape of CRs, but that term may also refer to the specific process in which CRs are released by the accelerator. For that reason, and also because what happens close to the sources can be more complex than CRs escaping from a single and well-defined accelerator, and, furthermore, sources may have an impact on the older galactic CR population roaming around them, we will focus in a more generic fashion on the phenomenology of CRs in the source vicinity.

As will be developed below, there are indications that CRs are not swiftly and seamlessly transferred from their source to the ISM at large. Instead, they likely experience some confinement in the vicinity of the source, and the duration and extent of this confinement will influence their early interaction with the ISM, with possible consequences on several of the observables through which we probe the CR phenomenon: for instance, the isotopic and spectral properties of the local flux of CRs, or the morphology and spectrum of the large-scale interstellar emission [290–292]. Before diving into gamma-ray observations, let us mention that, although a less direct evidence, features in the local CR flux can also be interpreted as resulting from processes related to the early stages of CR propagation. This possibility is important as it competes with dark matter interpretations of anomalies in the CR signal. This is true for instance of the local positron flux, which can be influenced by the dynamics of pair release from nearby pulsars [293–295]. In the context of this review, however, we will leave such considerations aside.

In the context of gamma-ray observations, CRs in the vicinity of sources will give rise to emission structures lying at intermediate spatial scales between isolated objects, such as SNRs or PWNe, and the large-scale diffuse emission of the ISM. Here we will focus on how the phenomenon fits into the global gamma-ray emission of a star-forming galaxy, with particular emphasis on the Milky Way. CRs in the vicinity of sources can be associated to specific populations of gamma-ray sources or specific regions of the ISM, and we will review below the current knowledge on such objects for three categories: emission beyond the shock in SNRs, emission around pulsars and their nebulae, and extended emission coincident with star-forming regions (SFRs). This categorization does not necessarily corresponds to specific kinds of acceleration sites, as mixed scenarios are likely to occur: particle escape from a PWN influenced by conditions inherited from the parent SNR, or superposition of processes in rich SFRs. Before going deeper into the topic, let us mention that the topic of CRs in the vicinity of sources is still actively being explored, owing both to the complexity of the physical problem, the diversity of possible astrophysical setups, and the difficulty in giving a clear-cut interpretation to existing observations. Rapid evolutions are therefore expected in this field in the coming years.

5.1. Physical problem

The release of CRs from a source is most likely more than just a localized and temporary enhancement in CR density. First of all, in general the release of the non-thermal particle content of a source will not be instantaneous, but it will be spread over time with some energy dependence [296,297]. Second, the energy density associated to the CR enhancement around

a source will exceed the typical energy density of the ISM for long durations [194], and it will be hard to avoid some dynamical feedback of escaping CRs on the surrounding medium [298]. Both considerations point to CR escape being a complex problem of non-linear dynamics and plasma physics, with strong dependence on both the actual history of particle release from the source and the environmental conditions around it.

The non-thermal particle yield of an accelerator is expected to be released progressively, over time scales comparable to the lifetime of the source and in an energy-dependent way. In the specific case of diffusive shock acceleration in SNRs, the highest-energy particles in the 100 TeV-1 PeV regime detach from the accelerator in the very first few 10^2 yr of the SNR expansion, while 1-10 GeV particles are released after 10^5 yr, when the SNR enters the radiative stage [299]. Particle escape is intimately connected to the acceleration process [300,301], so lacking a fully consistent and effective theory for the latter necessarily impacts our understanding of the former. That difficulty, however, can be turned into an opportunity. Since escape is so deeply rooted into the acceleration process, studying the vicinity of sources provide a complementary opportunity to address several key questions of the CR phenomenology. What is the exact CR spectrum fed into the ISM by the (different categories of) sources? What was the maximum energy attained by accelerated particles and how did it evolve in time? What fraction of the source power/energy went into CRs? What is the nature of the accelerated particles? We will illustrate below how some of these questions can be addressed in practice. Before doing so, we emphasise the relevance [and challenges, see 302] of searching for signatures of CRs around sources when trying to elucidate the question of acceleration up to the so-called knee region of the local CR spectrum, essentially because the time spent by such very-high-energy particles in the accelerator is small and the probability of finding active sources accelerating particles to PeV energies (PeVatrons) in the Galaxy is accordingly limited [303].

Once decoupled from the acceleration zone, fresh CRs will influence the transport conditions in the medium surrounding the source, via the same processes that governed their confinement in the source, i.e., self-generation of magnetic turbulence from resonant and non-resonant instabilities [301,304]. This modifies the conditions of particle transport around the source for long durations and over large scales, with a strong energy dependence. In [297], it is estimated that CRs escaping in a hot fully-ionized medium will experience suppressed diffusion by a factor reaching up to 20 with respect to the ISM at large within 50-100 pc around the source and over durations of the order of several 10 kyr for 1 TeV particles and several 100 kyr for 10 GeV particles. For a medium containing neutral species, the damping of the self-generated turbulence from ion-neutral friction reduces the confinement duration by nearly a factor of 10 [296,305]. In addition to this impact on the local turbulence, if the density of energy and momentum carried by escaping particles is comparable to or well in excess of what is found in the medium, major dynamical effects can result, such as the clearing of the surrounding medium by an overpressurised bubble of trapped CRs [298].

These processes can be expected to give rise to a large variety of observable situations depending on the actual parameters of the problem: the stage of the process being witnessed, the particle energy being probed, the physical scales accessible to the observation, the objects and processes involved in particle acceleration, the interstellar conditions in which escape takes place and by which it is made visible to us. On the latter point, one should mention that anisotropic diffusion around sources, physically well-motivated, and resulting either from the orientation of the regular background magnetic field or from the actual topology of the large-scale turbulence modes, can produce non-trivial emission patterns hard to identify and interpret [306]. Indeed, as illustrated below, the observational evidence associated to escaping CRs is very diverse and its interpretation is far from unified.

5.2. Emission beyond the shock in supernova remnants

SNRs still remain the leading candidates for the acceleration of Galactic CRs, if not necessarily for the PeV-energy particles at least for the bulk of the lower-energy population, so searching around them for particles in the process of merging with the galactic population is a promising avenue, and may provide useful information complementary to that inferred from on-going acceleration. Observationally, however, the picture can be very diverse: emission from the immediate shock upstream, from particles that are detaching from the shock precursor [307]; emission from shocks crushing into nearby clouds, possibly causing a sudden release of the CRs trapped downstream [308] or the reacceleration of ambient CRs trapped in the clouds [309,310]; emission from escaped particles that are well detached from the shock and have diffused out to some distance and illuminate large gas clouds [311,312]. In the last two cases, the overall picture can be complicated by the fact that escaping particles can diffuse back into their parent SNR, even without participating anymore in the acceleration, and contribute to the gamma-ray emission of the object [313].

Let us focus first on emission relatively close to the shock. Deep H.E.S.S. observations of RX J1713.7-3946 illustrate the challenge of studying the early stages of CR release from an SNR, i.e., particles just detaching from the shock, even with unprecedented angular resolution allowing to probe sub-parsec scales [307]. Significant extension of the gamma-ray emission beyond the shock (traced by X-ray synchrotron emission) is detected but cannot clearly be attributed to the shock precursor or diffusive escape, nor can it be used to tell the exact nature of emitting particles. It is unclear whether next-generation gamma-ray instruments will provide sufficient improvement in angular resolution to revolutionize such analyses, and here it seems that multiwavelength studies will be key to advance our knowledge. On slightly larger scales, and on a slightly older object more prone to significant escape, gamma-ray emission beyond the shock of γ Cygni was detected over a broad spectral range thanks to *Fermi*-LAT and MAGIC observations [314]. Interpreted in a coherent framework linking acceleration and escape along the SNR's history, the observations illustrate the joint constraints that can be derived on both processes, e.g. the time evolution of the maximum CR energy, the acceleration efficiency, or the diffusion coefficient in the vicinity of the remnant. The results also point to a diffusion coefficient two orders of magnitude smaller than in the Galaxy at large.

Looking at larger physical scales, SNRs interacting with molecular clouds have become over the past decade a growing class of gamma-ray sources. Currently, 8 TeV sources are classified as such in the TeVCat catalog[4], and 11 GeV sources were classified as such in Acero *et al.* [315], based on coincident molecular line emission, especially OH maser emission at 1720 MHz. The two sets overlap and we summarized the full sample in Table 1 for convenience.

Although the sample is still limited and there is large scatter in the observed or inferred properties, interacting SNRs seem to be older and more luminous systems, with a softer emission spectra, compared to younger SNRs that are still in their Sedov phase and feature high-velocity shocks [315]. In this evolutionary trend, there may be a gradual shift in emission processes, with younger systems being dominated or having a more significant contribution from IC scattering, while older interacting systems would be dominated by pion decay emission. Increased population statistics provided by the future Galactic plane survey with CTA may well reveal a less clear-cut separation between the two classes. In a few cases the data allow a (model-dependent) estimation of the diffusion coefficient around the SNR, which is found to be a factor of a few to a hundred lower than in the ISM at large [312,316].

In the VHE range, more interacting SNRs are likely to be found among the $\sim$60 unidentified low-latitude TeV sources, for instance HESS J1852-000 [317] or HESS J1702-420 [318].

[4] See http://tevcat2.uchicago.edu, database version tevcat2_test.3437

Table 1. List of the main established or candidate interacting SNRs detected in gamma-rays

Name	α (deg)	δ (deg)	Distance (kpc)	GeV source	TeV source
SNR G006.4-00.1 (W 28)	270.34	-23.29	$1.9^{+0.4}_{-0.4}$	4FGL J1801.3-2326e	HESS J1801-233/J1800-240A,B,C
SNR G008.7-00.1 (W 30)	271.41	-21.61	4.5	4FGL J1805.6-2136e	
SNR G023.3-00.3 (W 41)	278.72	-08.74	$4.2^{+0.3}_{-0.3}$		HESS J1834-087
SNR G034.7-00.4 (W 44)	284.04	01.22	3.0	4FGL J1855.9+0121e	
SNR G043.3-00.2 (W 49B)	287.79	09.11	10^{+2}_{-2}	4FGL J1911.0+0905	HESS J1911+090
SNR G049.2-00.7 (W 51)	290.96	14.10	$4.3^{+1.7}_{-0.0}$	4FGL J1923.2+1408e	HESS J1923+141
SNR G089.0+04.7 (HBH 21)	311.25	+50.58	$1.7^{+1.3}_{-1.1}$	4FGL J2045.2+5026e	
SNR G189.1+03.0 (IC 443)	94.51	22.66	1.5	4FGL J0617.2+2234e	TeV J0616+225
SNR G318.2+00.1	223.70	-59.07	$4.0^{+5.4}_{-0.7}$		HESS J1457-593
SNR G348.5+00.1 (CTB 37A)	258.63	-38.48	$9.0^{+0.5}_{-2.7}$	4FGL J1714.4-3830	HESS J1714-385
SNR G349.7+00.2	259.54	-37.31	$11.5^{+0.7}_{-0.7}$	4FGL J1718.0-3726	HESS J1718-374
SNR G357.7-00.1	265.07	-30.97	12.0	3FGL J1741.1-3053	
SNR G359.1-00.5	266.37	-29.95		4FGL J1745.8-3028e	HESS J1745-303
LHA 120-N132D	81.26	-69.64	50		TeV J0525-696

Notes: Right ascensions and declinations in degrees were recovered from the CDS and rounded to two decimals. Associations with gamma-ray sources were obtained from the Centre de Données astronomiques de Strasbourg (CDS) and, for the TeV counterparts, we favoured the HESS naming when available and otherwise used the TeVCat naming. Distance estimates in kpc were reproduced from Acero *et al.* [315] or from TeVCat when not available in the former reference, except for the distance to the LMC which was set to 50 kpc. Note that LHA 120-N132D was clearly detected as a GeV source but has no specific 4FGL name.

At intermediate energies between the GeV and TeV ranges, Eagle *et al.* [318] find a dozen unassociated low-latitude and hard-index objects in the 2FHL catalog of LAT sources emitting above 50 GeV, and argue that some of them may be SNRs interacting with gas clouds, via direct shock interaction or escaping CRs illuminating distant gas; such an interpretation is proposed for two sources with emission coincident with the edge of an SNR [318,319]. In the HE range, Acero *et al.* [315] indicate that an additional ∼50 significant sources only marginally overlapping with radio SNRs are found and may be interpreted as regions of high-density gas illuminated by escaping CRs that propagated away from their source. Overall, the detected source population at GeV energies is consistent with being mostly composed of SNRs interacting with dense material with effective densities of the order of tens H cm^{-3}. The numbers quoted above show the potential of CRs in the vicinity of sources to account for some fraction of the currently unidentified gamma-ray sources, both in the GeV and TeV range.

Tang [320] investigated 10 *Fermi*-LAT SNRs[5], testing two competing scenarios for the origin of the emission: direct interaction of the SNR shock with dense gas clouds, or escaped CRs illuminating nearby molecular clouds. The author concludes that the observed properties of the sample are inconsistent with the escape scenario, because the latter would imply a variety of spectral shapes, especially low-energy cutoffs, that is not observed. Instead, direct interaction involving reacceleration of ambient, potentially harder CRs and adiabatic compression is claimed to explain the diversity of spectral shapes of the sample, in particular the variety of high-energy breaks (as observed in W49B, W51C, or G349.7+0.2). Clearly, a

[5] The sample of 11 sources from Acero *et al.* [315] minus HBH 21, which is not detected above 10 GeV [321] owing to its spectral turnover at ∼1 GeV [322,323].

consensus on the physics at play in these object is not yet reached, and it is not obvious that gamma-ray observations alone will suffice to lift the ambiguities in the interpretation.

The object W28 alone exemplifies how complex the emission scenario may be: the emission observed around the SNR is composed of 4-5 components, most of which are detected at both GeV and TeV energies, and currently proposed interpretations involve a combination of direct shock interaction, possible triggering leakage of CRs from the remnant, escaped CRs illuminating distant clouds, and the contribution of background CRs [308,312].

5.3. Emission around pulsars and their nebulae

Pulsars and their wind nebulae are highly efficient factories of non-thermal electron/positron pairs, which are produced and accelerated in the magnetosphere, the relativistic wind, and its termination shock. PWNe are a good example of the complexity of studying particles freshly detached from their source, with a strong impact from both the original source and its surroundings, and are a remarkable constituent of the non-thermal landscape of the Milky Way, especially in the VHE range [324].

Recently, the phenomenon has acquired a new dimension with the discovery of very extended emission components beyond what was held for the boundaries of PWNe in a few systems. The pulsars concerned have ages of the order of 100 kyr and have reached an evolutionary phase where a large fraction of the accelerated electron/positron pairs can rapidly escape from the shocked pulsar wind into the surrounding medium, for instance by leakage from a bow-shock PWNe [325], instead of being long trapped into a hot and magnetized nebula [see a possible evolutionary path in 326]. These so-called halos were originally discovered with HAWC in the TeV range around pulsars B0633+17 (Geminga) and B0656+14 [293]. The most natural interpretation for the observed signal was radiation from energetic electrons/positrons IC scattering off ambient photons.

A major result was that the intensity distribution and flux level indicates a very strong confinement of particles around the source, with diffusion being suppressed by a factor of a few hundreds compared to the average ISM value inferred from local CR measurements [293]. Halos are a growing source class that may account for a large fraction of currently unidentified extended VHE sources [41,327–329], and that, as a population, potentially give a non-negligible contribution to the diffuse emission from the Galaxy, or at least some regions of it [330,331]. The phenomenon can naturally be expected to give rise to emission in other bands, and indeed the Geminga halo was later found at 10-100 GeV energies using *Fermi*-LAT data [332]. A broadband picture of pulsar halos is however largely missing today and searches are currently aiming at uncovering or expanding the population of halos in the radio, X-ray, and GeV bands.

Pulsar halos constitute a good opportunity to study particle transport around sources, especially because the radiating pairs are energetically subdominant in the medium [333]. Self-confinement by the streaming pairs could be responsible for diffusion suppression in the early phases, but the challenge in the case of middle-aged pulsars like Geminga is to sustain that confinement at later times, when the pulsar spin-down power has much decreased [334]. An alternative explanation is the presence of fluid turbulence injected at small scales to guarantee sufficient power at the scales relevant for 100 TeV particle scattering [335]. Such conditions could be inherited from the expansion of the parent SNR. Another possibility is that escaping pairs are experiencing the turbulence imprinted in the vicinity of the system by CRs escaping from the parent SNR, although here again the question of maintaining such a turbulence over several 100 kyr should be further investigated. In any case, pulsar halos offer a great opportunity to study CR transport in the vicinity of some accelerators.

5.4. Emission coincident with star-forming regions

SFRs, especially the most extensive ones involving massive stars, are expected to be prominent objects in the gamma-ray sky. First, because a large fraction of the most promising sites for particle acceleration will be found clustered in SFRs (colliding-wind binaries, pulsars and their nebulae, SNRs). Second, because SFRs are rich in targets for CR interactions, massive gas remainders from the parent molecular clouds and radiation fields enhanced by the many luminous stars, which guarantees an efficient conversion of CR energy into gamma rays. Therefore, SFRs seem to offer optimal conditions when it comes to studying how CRs are transferred from accelerators to the galactic population, by providing frequent injection of accelerated particles and favourable conditions to observe them.

The reverse side of the medal, however, is that the clustering of high-energy objects in the same region of space and time makes it difficult to unambiguously interpret gamma-ray observations of limited angular resolution and clearly associate a gamma-ray source with particles released from a specific object. From stellar evolution data used in [336], supernova explosions will be nearly uniformly distributed in time between 3 Myr after the initial burst of star formation (for the most massive 120 $M_\odot$ stars) and 37 Myr (for the least massive 8 $M_\odot$ stars). As soon as the SFR hosts more than a few hundreds massive stars, the average time interval between supernova explosions is less than the gamma-ray lifetime of many high-energy sources ($\sim$50 kyr for SNRs or PWNe and $\sim$ 1 Myr for pulsars and their halos or colliding-wind binaries), and so a superposition of gamma-ray-emitting objects can be expected.

Furthermore, it has long been speculated that the clustering of high-energy objects may play a distinctive role in the origin of CRs [337–339]. This may occur through a variety of possible processes, for instance long-lived particle acceleration at the termination shocks of stellar winds [340,341], repeated shock acceleration or turbulent reacceleration in the interior of superbubbles and/or at their bounding shell [336,342,343], acceleration at the highest energies in converging shock flows [344]. Although solid observational evidence of this class of phenomena is still largely missing, a link between CR origin and SFRs is supported by the isotopic composition of CRs [56,345,346]. Therefore, the potential of SFRs to bring answers to key issues in CR astrophysics justifies a continued search to detect and characterize their gamma-ray emission. Yet, as illustrated below, isolating that specific contribution amidst a variety of concurrent particle accelerators and gamma-ray emitting sources is a real challenge.

Table 2. List of the most prominent star-forming regions with established or potential detections in gamma-rays

Name	α (deg)	δ (deg)	Distance (kpc)	GeV sources	TeV sources
Cygnus	307.17	41.17	1.3-1.8	4FGL J2028.6+4110e	TeV J2031+406
GC	266.42	-29.01	8.5		TeV J1745-290
					TeV J1745-290d
NGC 3603	168.83	-61.26	6-8	4FGL J1115.1-6118	
Westerlund 1	251.77	-45.85	4-5	4FGL J1645.8-4533	HESS J1646-458
				4FGL J1648.4-4611	
				4FGL J1649.2-4513	
				4FGL J1650.3-4600	
				4FGL J1652.2-4516	
Westerlund 2	155.99	-57.76	4-6	4FGL J1023.3-5747e	HESS J1023-575
W 43	281.88	-01.94	5-7	4FGL J1847.2-0141	HESS J1848-018
				4FGL J1848.6-0202	
				4FGL J1848.7-0129	
30 Doradus C	83.96	-69.21	50		30 Doradus C

Notes: Right ascensions and declinations in degrees were recovered from the CDS and rounded to two decimals. Associations with gamma-ray sources were obtained from the CDS, complemented by the 4FGL [310] and the H.E.S.S. Galactic Plane Survey [40] catalogs. Some SFRs are associated to multiple 4FGL pointlike sources when no dedicated extended template was included in the automated catalog analysis. For the TeV counterparts, we favoured the H.E.S.S. naming when available and appropriate, and otherwise used the TeVCat naming. Typical distance estimates come from determinations and/or literature review in [112] for Cygnus, in [347] for NGC 3603, in [113] for Westerlund 1, in [348,349] for Westerlund 2, and in [350] for W 43. Note that the ranges of distances result from both the difficulty in accurately locating the region in the Milky Way and its actual spread along the line of sight (e.g. substructures, in the Cygnus OB2 association). Typical distances of 8.5 and 50 kpc are used for the GC and LMC, respectively.

As of today, GeV emission has been detected in the direction of about half a dozen SFRs. The most prominent SFRs studied in our Galaxy are listed in Table 2: the Cygnus region [351], NGC 3603 [352], Westerlund 1 [353], Westerlund 2 [354], and W 43 [355]. Several of these targets are also associated to emission in the TeV range, for instance Cygnus [356,357], Westerlund 1 [358], or Westerlund 2 [359]. We can also include in this category the Galactic center, in which diffuse TeV emission is detected in the direction of three super-massive star clusters, although the CR source there may alternatively be connected to the central black hole [360]. Outside of the Milky Way, a handful of SFRs may be studied with existing gamma-ray instruments, for instance 30 Doradus or the N11 region in the LMC, NGC 602 and NGC 346 in the SMC. The 30 Doradus region was observed at both GeV and TeV energies [263,361]. The entire region is detected at GeV energies only, although with no apparent specific feature, while only a more peripheral emission was detected at TeV energies only in the direction of the 30 Doradus C superbubble.

Over recent years, the list has been rapidly expanding thanks to more candidates emerging in gamma-ray source catalogs and a growing number of dedicated studies. The latest 4FGL-DR2 revision of the main *Fermi*-LAT catalog contains five associations of gamma-ray sources with SFRs: beside Cygnus and Westerlund 2 that were already mentioned, we find ρ Ophiuchi and the H II region Sh 2-152 in our Galaxy, and NGC 346, which is the brightest star-forming region in the SMC [362]. The general LAT catalogs are mainly aimed at pointlike

sources, therefore important complementary information is gathered by catalogs targeting extended sources, which revealed emission in the direction of W 30 [321], and of NGC 7822, NGC 1579, and IC 1396 [363]. The latter work, however, illustrates the difficulties of disentangling the potential emission of a SFR from the foreground and background emission in a catalog-like analysis.

Among dedicated studies investigating the link between gamma-ray emission and SFRs we mention for instance those of W 30 [364], W 40 [365], or the H II region G41.1-0.2 [366]. In the direction of region G25.0+0.0, extended gamma-ray emission was detected and, based on similarities with Cygnus and a positional correlation with gas structures and energetic sources, it was proposed to be associated with a putative candidate OB association [367].

A common feature of several of these gamma-ray sources is their extended morphology and relatively hard spectrum, which is exactly what one would expect from young CRs freshly detached from their sources and diffusing away into the ISM. Where both GeV and TeV detections are available, however, it is frequent to observe mismatch and offsets in the GeV and TeV morphologies [see e.g. 353, for Westerlund 1]. This is often explained in terms of superposition of sources, as anticipated above [e.g. an SNR interacting with a molecular cloud and a pulsar/PWN system in the case of W30; see 364]. To date, the most convincing association between gamma-ray source and SFR remains Cygnus, where the morphology of the gamma-ray emission at GeV energies shows a striking resemblance with the cavities carved in the ISM by the stellar winds and ionization fronts [351].

For some of these sources, the gamma-ray data were used to infer a radial distribution of CRs around the presumed accelerator and derive constraints jointly on the CR injection history and transport properties. A $1/r$ radial profile determined for a handful of SFRs was recently invoked as evidence that SFRs are continuously releasing CRs over several Myr, presumably from the conversion of stellar wind power into CRs with 1-10% efficiencies [368]. For one of these regions, Westerlund 2, the diffusion coefficient was estimated to be 100 times smaller than in the large-scale ISM. While the radial CR distribution is indeed a powerful discriminant to characterize the acceleration site and transport process, establishing it from the gamma-ray observations is a highly non-trivial task.

First, the extended gamma-ray emission from the region of interest along the line of sight needs to be separated from the foreground and background emission in the Galaxy. Second, the conversion of gamma-ray intensity profile into CR density profile requires a robust knowledge of the gas mass distribution in/around the SFR. Unfortunately, both steps can be shaky. The first point was recently illustrated in the case of NGC 3603, where an updated Galactic interstellar emission model and a better treatment of other extended sources in the field resulted in the source associated with NGC 3603 not being significantly extended [352], contrary to earlier claims [354], even without resorting to dedicated interstellar emission models such as those developed for an analysis of the Cygnus region [351]. As to the second point, the lack of apparent correlation of the gamma-ray emission with the ambient gas distribution, e.g. for Westerlund 1 and 2 or Cygnus, even accounting for some radial CR distribution, should at the very least call for caution in the interpretation of the observations. In that respect, the $\sim$1-10 TeV emission from the Galactic center and its correlation with gas in the central molecular zone may still constitute the most convincing evidence for a $1/r$ radial CR density profile [360]. It is interesting that even in such conditions, the identification of the CR source remains elusive. The CR gradient at the center of the galaxy can be equally well explained from continuous CR injection by a central stationary source such as Sgr A* active over several Myr [360], or from stochastic CR injection by dozens of SNRs over 100 kyr [369,370].

In an effort to identify the specific role of SFRs in the CR lifecycle, it is equally interesting to consider those objects that were not detected yet. For instance, *Fermi*-LAT observations

of eight young star clusters with ages below a few Myr constrain the particle acceleration efficiency in stellar winds to be below 10% in four of them and below 1% in 2 of them [371]. For more evolved objects, it is striking that the extraordinary 30 Doradus region in the LMC does not stand out in GeV gamma-rays: after removing the emission from two very powerful pulsars lying in the field, the region seems to fit perfectly in the larger scale-emission of the galaxy and does not stand out in gamma-rays, or at least not in proportion of its ionizing luminosity [263,372]. Looking at even more evolved structures, it was recently established that the Orion-Eridanus superbubble does not harbour any significantly enhanced CR population at GeV energies with respect to the solar neighborhood [204]. It is not clear yet if this has to be attributed to time evolution, stellar content, or its less compact nature compared to, e.g., Cygnus, but this non detection needs to be taken into account when assessing the potential of superbubbles for turbulent acceleration/reacceleration [204,342].

6. The population of gamma-ray emitting galaxies: different realizations of the cosmic-ray phenomenon

In this section we will turn our attention to the population of external galaxies whose gamma-ray emission is not dominated by the activity from a central supermassive black hole. We will frequently refer to these as star-forming or starburst galaxies (hereafter SFGs and SBGs), although we recognize that such a naming is not fully appropriate because there are numerous examples of active galactic nuclei accompanied by star formation (establishing the relationship between the two phenomena being an active field of research).

Going beyond the three external galaxies spatially resolved by current gamma-ray telescopes (covered in Section 4), the interest of studying the integrated emission from gamma-ray emitting galaxies as a population is at least threefold:

1. it allows an investigation of the physics behind CR acceleration and transport in a wide variety of conditions that differ markedly from those typical of the Milky Way (ISM gas structure and densities, radiation and magnetic field intensities, role of strong galactic winds,...), thus providing a good test bed for our current understanding of CR physics;
2. it is needed to pinpoint the actual integrated contribution of star-forming galaxies to various extragalactic backgrounds, especially the gamma-ray background detected in the energy interval 0.1-820 GeV with the *Fermi*-LAT, and the diffuse astrophysical neutrino background observed in the energy interval 10 TeV-10 PeV with IceCube;
3. it offers a probe of feedback processes in galaxy evolution, because CRs can play an active role in mediating this feedback (e.g. by driving gas-loaded winds and outflows off the galactic disk, thus removing material for star formation, or by ionizing molecular gas, hence altering the conditions for it to take place), or simply because CR interactions can provide information on the ongoing processes affecting the matter cycle in galaxies.

In the following, we focus on the first two items. We first review the current status of gamma-ray observations before addressing how they can improve our understanding of CRs. Concerning the last item, an example for recent progress in incorporating gamma-ray observations in galaxy evolution works can be found in [4].

6.1. A growing source class: the gamma-infrared luminosity correlation and its implications

SFGs and SBGs were foreseen as a distinct GeV to TeV gamma-ray source class well before their actual detection [see e.g. 373,374]. As of today, gamma-ray emission has been established from a dozen SFGs and SBGs, mostly at GeV energies using the *Fermi*-LAT, and among these only two starbursts have also been observed at TeV energies. In a recent systematic search for gamma-ray emission from a sample of 588 SFGs using *Fermi*-LAT, 11 objects were firmly detected in the ∼0.1-100 GeV range [375]. For 2 out of the 11 (NGC 4945 and NGC 1068) and 2 additional candidates (NGC 2403 and NGC 3424), the detected emission cannot be

dominantly attributed to processes linked to star formation, and might be contaminated by an active galactic nucleus [283,375,376]. Among the 11 galaxies, M 82 and NGC 253 were also detected at TeV energies with the current generation of IACTs [377,378].

Although still limited, the detected sample covers a variety of galactic properties, ranging from dwarfs (SMC and LMC) to large spirals (M31), and from relatively quiescent to starbursting objects (Arp 220). Investigating how the inferred total gamma-ray luminosity for detected and non-detected objects scales with global galactic properties, evidence was found for a correlation of the former with the star-formation rate, as traced for instance by the infrared luminosity (in the 8-1000 μm band). Initially investigated in [379] and [283], the correlation was recently revisited in [375] and is now more firmly established with a significance close to 5σ. The relation between the two quantities appears mildly non-linear, with gamma-ray luminosity evolving as infrared luminosity to the power $\sim$1.3. In fitting the observed correlation to a power-law, however, a large dispersion is obtained and it is still not clear whether that is intrinsic, or due to biases and uncertainties in the galactic parameters adopted (e.g. distances or infrared luminosities), or a combination of both effects. The gamma-ray to infrared luminosity correlation is actually reminiscent of the far-infrared to radio correlation, both in their form and commonly accepted physical explanation, and several authors actually investigated them in a unified approach [380,381].

The correlation is thought to be driven by massive-star formation, which is at the origin of strong UV/optical light and CRs that, upon interaction with the ISM, power infrared and gamma-ray emission, respectively. The commonly accepted interpretation of the non-linear relation between gamma-ray emission and star-formation rate (traced by infrared) is that of an increasing calorimetric efficiency of the galaxies with respect to CRs. The calorimetric efficiency is defined as the fraction of CR power that is deposited in the ISM relative to that initially injected. High star-formation rates are reached in galaxies harbouring large amounts of dense molecular gas packed in small regions, typically $\sim 1 - 10 \times 10^8\,\mathrm{M}_\odot$ in a volume spanning a few hundred pc. The average gas volume density in such regions can reach a few $1000\,\mathrm{H\,cm^{-3}}$ and, due to the high density of young stars, the interstellar radiation field energy densities can exceed a few $1000\,\mathrm{eV\,cm^{-3}}$ (for comparison, typical values for the Milky Way are $1\,\mathrm{H\,cm^{-3}}$ and $1\,\mathrm{eV\,cm^{-3}}$). As a consequence, CRs lose their energy much more efficiently, in particular through radiative processes such as nucleon-nucleon inelastic collisions and IC scattering. The Milky Way is thought to be a poor proton calorimeter but a good electron calorimeter, with efficiencies of of the order of 1-2% and 40-80% respectively, depending on the transport scenario assumed [382] (these efficiencies were computed for radiative processes only; in terms of emission the low calorimetric efficiency of protons is compensated by the fact that they are a factor of a hundred more numerous than electrons in CRs). In contrast, a starburst galaxy like Arp 220 can reach a calorimetric efficiency above 80% for protons in some propagation models [383]. The argument for increasing calorimetry in high star-formation rate galaxies seems backed up by the fact that their gamma-ray spectrum is observed to be hard, or at least harder than in the Milky Way, with a photon index $\sim$2.2-2.3 [375,378], which is expected if the emission is mostly hadronic in origin and CR transport is loss-dominated.

Although such a scheme is very reasonable as a general trend, there are several caveats in the interpretation of the still limited sample of galaxies available today. First of all, there is not yet a clear consensus on how CR transport evolves or differs among galaxies, and this inevitably affects our ability to predict the corresponding gamma-ray emission. For example, advection-dominated transport in high-star formation rate galaxies with powerful winds could provide an alternative explanation to the hard spectra. This will be discussed in more detail in the next section. Furthermore, most galaxies in the sample are detected as point-like objects at GeV energies, and as such the gamma-ray emission is a blend of various sources: interstellar emission, populations of sources such as SNRs or young and recycled pulsars, etc.

In that respect, the only spatially resolved galaxies so far show diverse and rather unexpected pictures. The distribution and flux levels of the emission from the LMC, SMC, and M 31, as summarized in Sect. 4.1, call for caution in interpreting the existing observations of the population of SFGs and SBGs, especially at low star formation rates. Continuous efforts are needed to extend the sample of gamma-ray-emitting SFGs, both in number and spectral coverage.

In that respect, extending the spectra towards the MeV and TeV ranges would certainly be instrumental in securing our understanding of the emission from SFGs and SBGs. On one hand, observations of emission in the hard X-ray/soft gamma-ray bands, with NuSTAR of future instruments like ASTROGAM/AMEGO/GECCO, can probe the population of secondary and tertiary leptons in SFGs and SBGs, which is an indirect diagnostic of the degree of calorimetric efficiency in individual systems. On the other hand, in the VHE range, more detections of galaxies are crucially needed to enlarge the sample beyond M 82 and NGC 253 and allow for a deeper broadband population study. The higher sensitivity observations > 100 GeV with CTA and > 1 TeV with HAWC, LHAASO, and, possibly, SWGO, are expected to provide an extended spectral coverage relevant to study all those effects that are more likely to show up at $>$TeV energies in SBGs: energy-dependent escape, photon-photon absorption, emission from unresolved population of sources such as PWNe and pulsar halos. Also incorporating 100 MHz-10 GHz radio observations, with their much better angular resolution, in the interpretation of gamma-ray observations of SFGs and SBGs can help to separate emission components and add useful information, e.g., on the intensity distribution in the galactic wind, for those systems that are viewed mostly edge-on [384].

Beyond individual detections towards infrared-selected targets, SFGs and SBGs are expected to contribute to the extragalactic gamma-ray emission as an unresolved component, integrated over cosmological distances. Because their emission results from hadronic interactions (only partially for moderately star-forming galaxies like the Milky Way, but predominantly for starbursts like M 82 or Arp 220), they are also expected to contribute some background emission of neutrinos. Over recent years, advances in the understanding of the composition of the extragalactic gamma-sky above 50 GeV have seriously constrained the possible contribution from SFGs and SBGs. From improved catalogs of sources and photon counting statistics, the extragalactic source population is now constrained to be composed at 86% of blazars, thus leaving SFGs and SBGs as subdominant component [385]. A forward modelling of that contribution based on luminosity functions for star-forming galaxies and the observed gamma-ray to infrared luminosity correlation yield a contribution of the order of 10% of the extragalactic gamma-ray background [375,386,387], and a corresponding contribution at the 1% level to the 10 TeV-10 PeV astrophysical neutrino flux detected with IceCube [388]. It should be noted, however, that the latter constraint relies on the very strong assumption that the CR/gamma-ray properties inferred in the *Fermi*-LAT energy range can be readily extrapolated up to > 10 PeV. Here, more gamma-ray observations at the highest energies with LHAASO, HAWC, CTA, and SWGO are crucially needed. In addition, a more realistic modeling of SBGs, taking into account a diversity of spectra in the population, as actually observed in [375], allows a larger contribution to the neutrino background, while still being consistent with the extragalactic gamma-ray background [389].

6.2. How do cosmic-ray transport properties vary with galactic environment?

The sample of detected SFGs and SBGs spans 3-4 orders of magnitude in star formation rate, average gas density, or interstellar radiation field intensity [see 390, for some examples of extreme values]. This set of very diverse conditions can be expected to give rise to markedly different CR populations, in particular as a result of CR transport in the ISM being very unlike that prevailing in the Milky Way (so far, to our knowledge, a more limited attention was paid

to whether/how extreme galactic conditions could alter the population of accelerated particles fed by sources into the ISM). As such, the study of SFGs and SBGs in gamma rays is a very good test bed for CR physics. After about a decade since the very first detections of external SFGs and SBGs in gamma rays (beyond the LMC, already observed with EGRET), however, it is fair to say that this new class of gamma-ray sources has not dramatically modified our understanding of CRs in galaxies and there is still room for deeper studies. Below, we review our present ideas on how CRs may evolve in galaxies different from the Milky Way.

The efforts in modelling or interpreting the observed population can be broadly separated into two kinds: those aiming at reproducing the spectra of one or several detected individual objects, and those concerned with reproducing the gamma-ray to infrared luminosity correlation. In both cases, as already mentioned, CR injection into the ISM was treated with prescriptions similar to those used for the Milky Way, and it is rather on the side of CR transport that alternative scenarios were explored. The main processes regulating CR transport in the ISM were assessed in the specific context of SFGs and especially SBGs. Examples of relevant questions are:

- Energy losses: what are the properties of the various ISM components with which CRs interact (gas, radiation and magnetic fields)? What is the actual gas structure in terms of gas phase and relative volume filling factor? Do CRs experience large-scale volume-averaged ISM conditions or are they confined to specific phases?
- Diffusion: what is the nature of externally-driven turbulence in extreme environments such as the cores of SBGs (strength, injection and cutoff scales)? How will the self-generation of turbulence by CRs proceed in potentially very dense and weakly ionized media? If CR transport can be approximated by diffusion, what is its the form of the diffusion coefficient, its normalization and momentum dependence?
- Advection: what is the exact role of advection in CR transport in those galaxies where large-scale winds are observed or expected? What is the structure of the wind and how is it connected to the galactic disk? In particular, is it fast and distributed enough at its base to significantly affect CRs in the disk?

In the following, we illustrate how some of these questions were addressed in recent works and with what conclusions. We note that having at our disposal an open-source, versatile model, analogous to GALPROP, DRAGON, or PICARD (see Sect. 2.4), that can be easily configured and adapted to different galaxies would be ideal to go beyond simple leaky-box models and allow easier comparison of various transport scenarios.

The extreme conditions encountered in some SFGs, and especially in SBGs, translate into much stronger energy losses than those experienced by CRs in the average ISM of the Milky Way. This effect alone has already significant implications. In [381], CR transport is assumed to be very similar to that in the Milky Way (e.g. same diffusion coefficient), and the author assessed how the global interstellar gamma-ray emission is influenced by interstellar conditions, gas mass and distribution and the related radiation and magnetic fields, for molecular gas densities increasing from 5 to 500 $H_2\,cm^{-3}$ (which is a bit short to describe extreme SBGs like Arp 220). The impact is evaluated in three different energy ranges, 0.1-10 MeV, 0.1-100 GeV, and 0.1-100 TeV. As gas density increases (and so do star formation and radiation and magnetic fields), the MeV range sees a strong increase of IC and Bremsstrahlung emission and dominance of secondary electrons and positrons as radiating particles. In the GeV range, emission from nucleon-nucleon inelastic collisions is the dominant process, but its intensity increases slower than linearly with gas density and flattens as CR protons and nuclei propagation becomes loss dominated, which yields a harder spectrum with a 2.3-2.4 photon index. In the TeV range, while emission from IC scattering from primary electrons can be dominant or comparable to that from nucleon-nucleon inelastic collisions for the lowest average densities, the latter overwhelms the former at large densities because it increases

almost linearly with density, as CR protons and nuclei transport is largely diffusion-dominated at the highest energies.

One effect not investigated in [381] is that of photon-photon opacity in the densest radiation fields typical of SBGs. This was included in works such as [390] or [391], and the most interesting effects are the softening of the emission spectrum beyond TeV energies, and the generation of an additional population of CR electrons at TeV energies that, in the case of hard enough injection and strongly inhibited diffusion, yields a dominant IC contribution in hard X-rays. Both [381] and [390] (see also [392]) illustrate the potential of the hard X-ray/soft gamma-ray range as a diagnostic of CR interactions in dense galaxies, in particular for an indirect evaluation of the calorimetric efficiency of the system. The works of [390] and [393] show that improving current hard X-ray constraints from NuSTAR by a factor of a few may become interesting (in the case of NGC 253).

The above statements are dependent on the respective diffusion and advection properties assumed, and it actually remains to be clarified how both processes evolve among different galaxies and in which regime one or the other may dominate CR transport. Works like [390] or [394] succeeded in reproducing the gamma-ray emission spectrum from selected SBGs with a model in which wind advection is the dominant spatial transport mechanism (either by neglecting diffusion or using a small coefficient). In contrast, the model of [383] uses dedicated diffusion schemes but no advection, and is able to achieve the same goal. At the population level, the observed gamma-ray to infrared luminosity correlation can be fairly well reproduced with a Milky-Way-like diffusion scheme [381], but an advection scheme with star formation dependent wind velocity may be a viable alternative solution [395].

Figure 5 shows the most recent published version of the gamma-ray to infrared luminosity correlation, based on data from Ajello *et al.* [375]. Observations are compared to predictions from [381], based on a GALPROP-like model, and from Pfrommer *et al.* [396], based on an alternative modelling approach. In the latter work, the authors evaluated the gamma-ray emission in a series of two-fluid MHD galaxy formation simulations for a variety of dark matter halo masses and distributions. CRs are injected as a relativistic fluid following SN explosions and two CR transport scheme are considered: pure advection with gas motions, and advection plus anisotropic diffusion along magnetic field lines (with a single energy-independent value for the diffusion coefficient). Such an approach allows to track self-consistently and in a time-dependent way the interplay between CRs, gas motions and outflows, and galactic magnetic field structure, but it comes at the expense of approximations on the CR physics. Nevertheless, the authors show that, if a typical 10% of the supernovae kinetic energy goes into CRs, the observed gamma-ray to infrared luminosity correlation can be fairly well reproduced and argue that it is little dependent on uncertainties in the CR transport. At high star formation rates, most of the CR energy is lost to hadronic interactions, in agreement with the calorimetry argument put forward in other works; at low star-formation rates, however, the model overpredicts the emission and the authors suggest that more realistic multi-phase ISM descriptions would be needed in that range. Such works support a trend towards numerical models in which the complexity of CR physics is dynamically coupled to the galactic environment, models whose interest reaches far beyond high-energy astrophysics (e.g. cosmological simulations of galaxy formation).

In exploring CR spatial transport in external galaxies, the work of [383] offers an interesting alternative to the generic prescriptions mainly in use so far. The authors evaluate that starburst regions will be dominated by volume-filling, cold and weakly ionized gas, unlike the Milky Way, where the ISM is mostly filled with hot and warm ionized gas. This implies an efficient damping of MHD turbulence modes, from ion-neutral friction, and a cutoff of the externally-driven turbulence cascade at scales such that particles with energies below a few hundreds TeV are not efficiently scattered. The streaming of CRs is then controlled

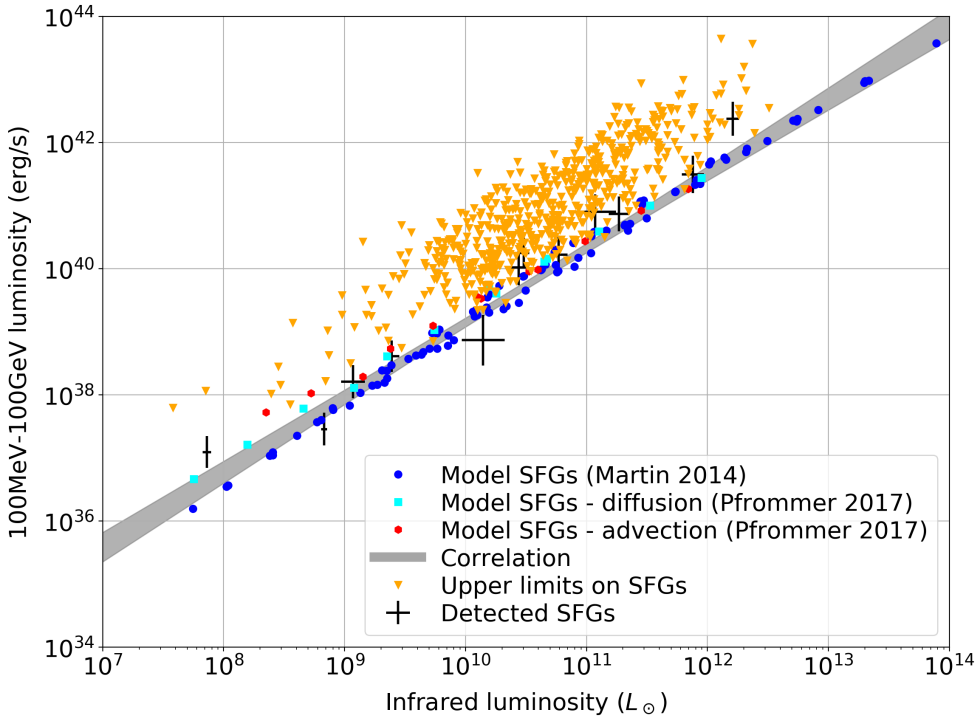


Figure 5. Gamma-ray to infrared luminosity correlation. The observed luminosities plotted as black crosses and orange triangles come from Ajello *et al.* [375] and were converted to the 0.1-100 GeV range under the assumption that the emission spectrum is a power law spectrum with photon index 2.2 (the average value measured by the authors). The correlation band plotted in gray is the one corresponding to the so-called combined fit. Overlaid are predicted luminosities: from [381], as blue dots, for a GALPROP-like model of galaxies with different size and gas distributions and under the assumption of a Milky-Way-like CR diffusion scheme; from Pfrommer *et al.* [396], as red hexagons and cyan squares, for a series of two-fluid magneto-hydrodynamical galaxy formation simulations for a variety of dark matter halos and under two different assumptions for CR transport: pure advection, and advection plus anisotropic diffusion (see text).

by self-generated turbulence only, which is demonstrated to occur at the Alfvén speed for particles ≤ 1 TeV and to increase with some power > 1 of energy above. The corresponding macroscopic diffusion scheme implies a specific flavour of energy-dependent diffusive escape such that the resulting hadronic-dominated gamma-ray spectrum follows the CR injection spectrum up to $\sim$100 GeV and gradually falls off at higher energies (also because photon-photon absorption comes into play). This is shown to provide a convincing description of SBGs M 82, NGC 253, and Arp 220. In that picture, advection in a galactic wind is not needed but not excluded either. The authors argue that, while most CRs should be injected into the cold gas phase to ensure a high calorimetric efficiency and a sufficient level of gamma-ray emission, some CRs can be advected away from the plane with the hot ionised phase but will contribute few gamma rays (and retain the hard injection spectrum anyway).

7. Summary and perspectives

In the past decade gamma-ray observations have shed a new light on the richness and diversity of the processes that govern the build-up of CR populations in galaxies. While there is an overall agreement with the foundations of the standard Galactic CR paradigm, gamma-ray

data, along with direct CR measurements and other related observables, have highlighted the limits of some standard assumptions and modelling approaches, and, therefore, have spurred many new developments in the field. We summarize below the main points highlighted in this review, before discussing forthcoming prospects and challenges.

- Gamma-ray emission from the local interstellar medium (Sec. 3) as seen by the *Fermi* LAT shows an overall consistency with direct CR measurements. However, a few hints of deviations are arising: fluctuations $\lesssim$ 25% in the emissivity of atomic hydrogen between different regions within 1 kpc from the Sun; an excess of $\sim$20-30% in the average emissivity of atomic hydrogen above 10 GeV; possible spectral deviations in a few molecular clouds. In order to definitely establish whether those deviations are related to CR injection and transport or to observational biases we need to improve our understanding of interstellar medium tracers and hadronic gamma-ray production cross sections. Complementary results are expected in the near future in the TeV energy range.
- The large-scale distribution of CR nuclei throughout the Milky Way disk (Sec. 4.1.1) as inferred from *Fermi*-LAT data shows a decline milder than expected toward the outer Galaxy and an increase by a factor of 1.5-3 in the inner Galaxy accompanied by a spectral hardening. Although alternative explanations, such as a contribution larger than currently believed of unresolved gamma-ray sources to diffuse emission, are still not completely ruled out, these trends may have deep implications for the physics of particle transport in the Galaxy (Sec. 4.1.2). Competing explanations include non-homogeneous diffusion related to turbulence generation around CR accelerators, the difference scaling with energy of parallel and perpendicular diffusion coupled with the peculiar structure of the Galactic magnetic field, and non-linear transport due to CR-driven instabilities. With the first detections of Galactic diffuse emission by current experiments, very-high-energy telescopes are one step away from helping with clarifying this puzzle.
- The large-scale distribution of CR electrons throughout the Milky Way disk (Sec. 4.1.1) is less constrained by existing observations. However, it is now established thanks to *INTEGRAL* SPI that there is a diffuse gamma-ray emission above 60 keV consistent with a dominant IC origin, and which suggests abundances of secondary CR electrons and positrons in the inner Galaxy larger than what is observed locally. Improved measurements in the MeV and TeV domains, along with multiwavelength data, are key to improve our understanding of CR electrons.
- Spatially-resolved gamma-ray observations of the closest external galaxies (Sec. 4.1.3), namely the LMC, SMC, and M31, reveal a varied picture at GeV energies. Large-scale CR density variations of a factor 2-3 similar to those found in the Milky Way appear in the LMC, but with 50% of the emission at 10 GeV contributed by extended components of a nature yet to be understood. Extended emission from the SMC lacks remarkable correlation with gas densities and star-formation sites, while in M31 gamma-ray emission is concentrated in innermost 5 kpc, possibly due to the larger concentration of an old stellar population in the bulge. These studies have so far been limited to the GeV range, and the topic has hardly been explored observationally in the TeV range.
- The *Fermi* LAT has enabled for the first time to image the CR diffusion halo in the Milky Way, and perhaps in M31 (Sec. 4.2). For the Milky Way observations of gas clouds at large distances from the disk show a marked decline of CR densities at heights above 2 kpc, while it is unclear if large fluctuations found between different clouds at lower altitudes are related to the large-scale vertical gradient of CRs or peculiar transport conditions. These observations complement and inform ongoing theoretical efforts to establish the physical origin of the CR diffusive halo in terms of plasma waves self-excited by the CR streaming instability possibly combined with turbulence advected from the Galactic disk.

- Standard implementations of CR propagation models for the Milky Way show an overall agreement with observations, but extended residual emission as large as 30% appears all over the sky in *Fermi*-LAT observations (Sec. 4.3). Possible explanations include contributions from populations of sub-threshold sources and an incomplete/imperfect census of target gas and radiation fields for gamma-ray production, but part of these residuals probably calls for refinements of the models, such as including a more realistic description of CR sources (e.g., time-dependent particle injection, detailed spatial distribution, including more classes of sources in addition to SNRs) and accounting for a variety of different aspects of transport (e.g., localized confinement in the vicinity of sources, role of galactic winds and outflows).
- GeV and TeV gamma-ray observations have been unveiling CR propagation in the vicinity of sources (Sec. 5) as a field rich in phenomenology, connecting acceleration at the sources and interstellar transport in the non-linear regime. This has the potential to give rise to emission structures and specific populations of gamma-ray sources on various scales. Gamma-ray emission around SNRs can be interpreted in a variety of scenarios: particles just detaching from the shock, escaping particles illuminating nearby clouds, and the reacceleration of old CRs trapped in shocked clouds. Gamma-ray halos surrounding old pulsars are a brand new class of emitters, governed by physics still largely to be understood, and whose long lifetimes may translate into a numerous population yet to be uncovered. Long-suspected gamma-ray emission in the direction of SFRs was finally detected, but without consensus yet on the processes that shape particle acceleration and transport in these environments, and whether they play a distinctive role if the CR life cycle. For most of these cases, theory and observations currently seem to point towards a suppression of the CR diffusion coefficient by one/two order of magnitudes around sources with respect to typical interstellar values, the implications of which deserve further investigation. A robust characterization of CRs in the vicinity of sources needs to go hand-in-hand with the characterization of large-scale interstellar emission and requires an extensive multiwavelength approach.
- A dozen star-forming galaxies are detected at GeV energies, and two at TeV energies (Sec. 6). The data reveal a mildly non-linear correlation with a significance close to 5σ between GeV and infrared luminosity. The correlation is driven by massive-star formation, which is at the origin of strong UV/optical light and CRs that, upon interaction with the ISM, power infrared and gamma-ray emission, respectively. The non-linear relation between the two is interpreted as resulting from an increasing CR calorimetric efficiency for galaxies with larger gas and radiation content. As a population, star-forming galaxies are estimated to contribute $\sim$10% of the extragalactic gamma-ray background, which extrapolates to only $\sim$1% of the TeV-PeV astrophysical neutrino flux. Several avenues are explored to model CR transport and interactions in a variety of galactic environments, but it is fair to say that our understanding of the CR life cycle still lacks extensive testing for other galaxies.

In the GeV range, where the entire sky was surveyed by the *Fermi* LAT, interstellar emission has been explored in depth. In the TeV range, however, we have just started to scratch the surface. In the next decade the deep and extensive surveys of the Galactic plane and LMC with CTA, combined with continued operation of HAWC and starting exploitation of LHAASO, and, possibly, the advent fo SWGO (Sec. 2.1) will unveil what interstellar emission looks like in the TeV range and above. This will shed light on crucial questions such as the origin of the inner-Galaxy hardening. It also has the potential to reveal how interstellar emission from >TeV CRs connects to ISM structures and the source regions on a variety of different scales. Very-high-energy observations are of particular interest to study the early phases of the CR life cycle. In particular, for the study of gamma-ray emission in the source

vicinity developments in the very-high-energy range may alleviate the problems encountered when trying to disentangle these objects from large-scale diffuse emission, since we expect that at energies above 100 GeV large-scale interstellar emission should be comparatively less intense. The good angular resolution of CTA is essential for proper identification of the emission components in complex regions or resolving of fine structures close to the shock in SNRs. At the same time TeV instruments are expected to increase the number of external galaxies detected in gamma rays and provide a broader view of different realizations of the CR phenomenon at very high energies.

Toward the lower-end of the gamma-ray spectrum, a new (sub-)MeV mission such as ASTROGAM, AMEGO or GECCO (see again Sec. 2.1) has a huge potential to better reveal spectral and spatial properties of IC emission from the Galaxy and, therefore, to infer the distribution of CR electrons which better sample CR inhomogeneities since they are affected by energy losses more strongly than nuclei and remain much closer to their sources. The improved performance in the MeV to GeV energy range of ASTROGAM or AMEGO would also allow to probe CR nuclei in nearby clouds at sub-pc scale to test how low-energy CRs get depleted at their interior, to follow the release and diffusion of particles around SNRs and pulsars, and to infer the spectral energy distribution of the bulk of CR nuclei in SFRs in order to estimate the CR pressure in these environments.

Besides developments in gamma-ray instruments, major progress in the years to come is expected to stem from a more extensive integration of gamma-ray observations in a broad multiwavelength/multimessenger context. This is necessary to unambiguously interpret gamma-ray observations, as illustrated along the review, and it is also needed to explore the cross effects with related fields (e.g., star formation, galaxy formation and evolution, astrobiology). At the same time, major theoretical and numerical developments are already warranted and under way (Sec. 2.4) and a challenge to the community is to ensure their open dissemination and use, in order to fully exploit the existing and forthcoming experimental data, and better connect macroscopic observables to the microphysics of non-thermal particle transport.

Author Contributions: The three authors contributed equally to conceptualization, literature review, visualization, and writing.

Funding: L.T. and P.M. acknowledge financial support by CNES for the exploitation of *Fermi*-LAT observation and from CNRS-INSU for their work on CTA, as well as from the French Agence Nationale de la Recherche under reference ANR-19-CE31-0014 (GAMALO project). D.G. has received financial support through the Postdoctoral Junior Leader Fellowship Programme from la Caixa Banking Foundation (grant n. LCF/BQ/LI18/11630014). D.G. was also supported by the Spanish Agencia Estatal de Investigación through the grants PGC2018-095161-B-I00, IFT Centro de Excelencia Severo Ochoa SEV-2016-0597, and Red Consolider MultiDark FPA2017-90566-REDC.

Acknowledgments: The authors acknowledge discussions during the preparation of the review with Peter von Ballmoos, Henrike Fleishack, Michael Kachelrieß, Elena Orlando, and Andy Strong. Special thanks to Sarah Recchia for providing the model curve in Figure 3, to Carmelo Evoli for providing the model curve in Figure 4, to Marco Ajello and Mattia Di Mauro for providing the data in Figure 5, and to Reshmi Mukherjee for the comments on the manuscript. This work has made use of the SIMBAD database, operated at CDS, Strasbourg, France, and of NASA's Astrophysics Data System Bibliographic Services. The preparation of the figures has made use of the following open-access software tools: APLpy [397], Astropy [398], Matplotlib [399], NumPy [400], SciPy [401].

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the review, in the literature selection, in the writing of the manuscript, or in the decision to publish.

Abbreviations

The following abbreviations are used in this manuscript:

CDS	Centre de Données astronomiques de Strasbourg
CR	cosmic ray
GC	Galactic center
HE	high energy
IC	inverse Compton
ISM	interstellar medium
LAT	Large Area Telescope
LIS	local interstellar spectrum
LMC	Large Magellanic Cloud
PSF	point spread function
PWN	pulsar wind nebula
SBG	starbust galaxy
SFG	star-forming galaxy
SFR	star-forming region
SMC	Small Magellanic Cloud
SNR	supernova remnant
VHE	very-high energy

References

1. V. L. Ginzburg and S. I. Syrovatskii. *The Origin of Cosmic Rays*; 1964.
2. Stefano Gabici et al. The origin of Galactic cosmic rays: challenges to the standard paradigm. *Int. J. Mod. Phys. D* **2019**, *28*, 1930022, [arXiv:astro-ph.HE/1903.11584]. doi:10.1142/S0218271819300222.
3. Marco Padovani et al. Impact of Low-Energy Cosmic Rays on Star Formation. Space Sci. Rev. **2020**, *216*, 29, [arXiv:astro-ph.GA/2002.10282]. doi:10.1007/s11214-020-00654-1.
4. T. K. Chan et al. Cosmic ray feedback in the FIRE simulations: constraining cosmic ray propagation with GeV γ-ray emission. MNRAS **2019**, *488*, 3716–3744, [arXiv:astro-ph.GA/1812.10496]. doi:10.1093/mnras/stz1895.
5. Tobias Buck, Christoph Pfrommer, Rüdiger Pakmor, Robert J. J. Grand, and Volker Springel. The effects of cosmic rays on the formation of Milky Way-mass galaxies in a cosmological context. MNRAS **2020**, *497*, 1712–1737, [arXiv:astro-ph.GA/1911.00019]. doi:10.1093/mnras/staa1960.
6. Philip F. Hopkins et al. Effects of different cosmic ray transport models on galaxy formation. MNRAS **2021**, *501*, 3663–3669, [arXiv:astro-ph.GA/2004.02897]. doi:10.1093/mnras/staa3692.
7. Noemie Globus and Roger D. Blandford. The Chiral Puzzle of Life. ApJ **2020**, *895*, L11, [arXiv:q-bio.OT/2002.12138]. doi:10.3847/2041-8213/ab8dc6.
8. S. Hayakawa. Propagation of the Cosmic Radiation through Interstellar Space. *Progress of Theoretical Physics* **1952**, *8*, 571–572. doi:10.1143/ptp/8.5.571.
9. G. W. Clark, G. P. Garmire, and W. L. Kraushaar. Observation of High-Energy Cosmic Gamma Rays. ApJ **1968**, *153*, L203. doi:10.1086/180252.
10. C. E. Fichtel, G. A. Simpson, and D. J. Thompson. Diffuse gamma radiation. ApJ **1978**, *222*, 833–849. doi:10.1086/156202.
11. Hans Bloemen. Diffuse galactic gamma-ray emission. ARA&A **1989**, *27*, 469–516. doi:10.1146/annurev.aa.27.090189.002345.
12. P. Sreekumar et al. Observations of the Large Magellanic Cloud in High-Energy Gamma Rays. ApJ **1992**, *400*, L67. doi:10.1086/186651.
13. P. Sreekumar et al. Constraints on the cosmic rays in the Small Magellanic Cloud. Phys. Rev. Lett. **1993**, *70*, 127–129. doi:10.1103/PhysRevLett.70.127.
14. Vitalii L. Ginzburg. Reviews of Topical Problems: Gamma Astronomy and Cosmic Rays. *Soviet Physics Uspekhi* **1973**, *15*, 626–631. doi:10.1070/PU1973v015n05ABEH005033.
15. Andrew W. Strong, Igor V. Moskalenko, and Vladimir S. Ptuskin. Cosmic-ray propagation and interactions in the Galaxy. *Ann. Rev. Nucl. Part. Sci.* **2007**, *57*, 285–327, [astro-ph/0701517]. doi:10.1146/annurev.nucl.57.090506.123011.
16. Isabelle A. Grenier, John H. Black, and Andrew W. Strong. The Nine Lives of Cosmic Rays in Galaxies. ARA&A **2015**, *53*, 199–246. doi:10.1146/annurev-astro-082214-122457.
17. Pierre Cristofari. TBD. *Universe, to be printed in this Special Issue* **2021**, *TBD*, TBD. doi:TBD.
18. M. S. Potgieter. Cosmic Rays in the Inner Heliosphere: Insights from Observations, Theory and Models. Space Sci. Rev. **2013**, *176*, 165–176. doi:10.1007/s11214-011-9750-7.

19. W. B. Atwood et al. The Large Area Telescope on the Fermi Gamma-Ray Space Telescope Mission. ApJ **2009**, *697*, 1071–1102, [arXiv:astro-ph.IM/0902.1089]. doi:10.1088/0004-637X/697/2/1071.
20. A. Malizia et al. INTEGRAL view of TeV sources: a legacy for the CTA project. *Universe, to be printed in this Special Issue* **2021**, *TBD*, TBD. doi:TBD.
21. G. Vedrenne et al. The SPI Spectrometer for the INTEGRAL Mission. *Physica Scripta Volume T* **1998**, *77*, 35.
22. V. Schoenfelder et al. Instrument Description and Performance of the Imaging Gamma-Ray Telescope COMPTEL aboard the Compton Gamma-Ray Observatory. ApJS **1993**, *86*, 657. doi:10.1086/191794.
23. Jürgen Knödlseder. The future of gamma-ray astronomy. *Comptes Rendus Physique* **2016**, *17*, 663–678, [arXiv:astro-ph.IM/1602.02728]. doi:10.1016/j.crhy.2016.04.008.
24. A. De Angelis et al. The e-ASTROGAM mission. Exploring the extreme Universe with gamma rays in the MeV - GeV range. *Experimental Astronomy* **2017**, *44*, 25–82, [arXiv:astro-ph.HE/1611.02232]. doi:10.1007/s10686-017-9533-6.
25. Julie McEnery et al. All-sky Medium Energy Gamma-ray Observatory: Exploring the Extreme Multimessenger Universe. Bulletin of the American Astronomical Society, 2019, Vol. 51, p. 245, [arXiv:astro-ph.IM/1907.07558].
26. Alessandro De Angelis et al. Gamma-ray Astrophysics in the MeV Range: the ASTROGAM Concept and Beyond. *arXiv e-prints* **2021**, p. arXiv:2102.02460, [arXiv:astro-ph.IM/2102.02460].
27. M. Sasaki and Gecco Team. Galactic Explorer with a Coded Aperture Mask Compton Telescope (gecco). American Astronomical Society Meeting Abstracts, 2021, Vol. 53, *American Astronomical Society Meeting Abstracts*, p. 132.02.
28. Razmik Mirzoyan. Technological and instrumental advances in ground-based gamma-ray astronomy. *Universe, to be printed in this Special Issue* **2021**, *TBD*, TBD. doi:TBD.
29. Mathieu de Naurois and Daniel Mazin. Ground-based detectors in very-high-energy gamma-ray astronomy. *Comptes Rendus Physique* **2015**, *16*, 610–627, [arXiv:astro-ph.IM/1511.00463]. doi:10.1016/j.crhy.2015.08.011.
30. F. Aharonian et al. Discovery of very-high-energy γ-rays from the Galactic Centre ridge. Nature **2006**, *439*, 695–698, [arXiv:astro-ph/astro-ph/0603021]. doi:10.1038/nature04467.
31. A. Abramowski et al. Diffuse Galactic gamma-ray emission with H.E.S.S. Phys. Rev. D **2014**, *90*, 122007, [arXiv:astro-ph.HE/1411.7568]. doi:10.1103/PhysRevD.90.122007.
32. M. Actis et al. Design concepts for the Cherenkov Telescope Array CTA: an advanced facility for ground-based high-energy gamma-ray astronomy. *Experimental Astronomy* **2011**, *32*, 193–316, [arXiv:astro-ph.IM/1008.3703]. doi:10.1007/s10686-011-9247-0.
33. Cherenkov Telescope Array Consortium et al. *Science with the Cherenkov Telescope Array*; 2019. doi:10.1142/10986.
34. A. A. Abdo et al. Observation and Spectral Measurements of the Crab Nebula with Milagro. ApJ **2012**, *750*, 63, [arXiv:astro-ph.HE/1110.0409]. doi:10.1088/0004-637X/750/1/63.
35. A. U. Abeysekara et al. Observation of the Crab Nebula with the HAWC Gamma-Ray Observatory. ApJ **2017**, *843*, 39, [arXiv:astro-ph.HE/1701.01778]. doi:10.3847/1538-4357/aa7555.
36. M. Amenomori et al. Observation of Multi-TEV Gamma Rays from the Crab Nebula using the Tibet Air Shower Array. ApJ **1999**, *525*, L93–L96, [arXiv:astro-ph/astro-ph/9909172]. doi:10.1086/312342.
37. G. Aielli et al. Layout and performance of RPCs used in the Argo-YBJ experiment. *Nuclear Instruments and Methods in Physics Research A* **2006**, *562*, 92–96. doi:10.1016/j.nima.2006.02.136.
38. Zhen Cao et al. Introduction to Large High Altitude Air Shower Observatory (LHAASO). Chinese Astron. Astrophys. **2019**, *43*, 457–478. doi:10.1016/j.chinastron.2019.11.001.
39. A. Albert et al. Science Case for a Wide Field-of-View Very-High-Energy Gamma-Ray Observatory in the Southern Hemisphere. *arXiv e-prints* **2019**, p. arXiv:1902.08429, [arXiv:astro-ph.HE/1902.08429].
40. H. E. S. S. Collaboration et al. The H.E.S.S. Galactic plane survey. A&A **2018**, *612*, A1, [arXiv:astro-ph.HE/1804.02432]. doi:10.1051/0004-6361/201732098.
41. A. Albert et al. 3HWC: The Third HAWC Catalog of Very-high-energy Gamma-Ray Sources. ApJ **2020**, *905*, 76, [arXiv:astro-ph.HE/2007.08582]. doi:10.3847/1538-4357/abc2d8.
42. D. Berge, S. Funk, and J. Hinton. Background modelling in very-high-energy γ-ray astronomy. A&A **2007**, *466*, 1219–1229, [arXiv:astro-ph/astro-ph/0610959]. doi:10.1051/0004-6361:20066674.
43. A. Jardin-Blicq, V. Marandon, and F. Brun. A complementary view of the galactic plane in TeV gamma rays by HAWC and H.E.S.S. 36th International Cosmic Ray Conference (ICRC2019), 2019, Vol. 36, *International Cosmic Ray Conference*, p. 706, [arXiv:astro-ph.HE/1908.06658].
44. I. Vovk, M. Strzys, and C. Fruck. Spatial likelihood analysis for MAGIC telescope data. From instrument response modelling to spectral extraction. A&A **2018**, *619*, A7, [arXiv:astro-ph.IM/1806.03167]. doi:10.1051/0004-6361/201833139.
45. J. Knödlseder et al. Analysis of the H.E.S.S. public data release with ctools. A&A **2019**, *632*, A102, [arXiv:astro-ph.IM/1910.09456]. doi:10.1051/0004-6361/201936010.
46. L. Mohrmann et al. Validation of open-source science tools and background model construction in γ-ray astronomy. A&A **2019**, *632*, A72, [arXiv:astro-ph.IM/1910.08088]. doi:10.1051/0004-6361/201936452.

47. M. Holler, J. P. Lenain, M. de Naurois, R. Rauth, and D. A. Sanchez. A run-wise simulation and analysis framework for Imaging Atmospheric Cherenkov Telescope arrays. *Astroparticle Physics* **2020**, *123*, 102491, [arXiv:astro-ph.HE/2007.01697]. doi:10.1016/j.astropartphys.2020.102491.
48. R. Atkins et al. Observation of TeV Gamma Rays from the Crab Nebula with Milagro Using a New Background Rejection Technique. ApJ **2003**, *595*, 803–811, [arXiv:astro-ph/astro-ph/0305308]. doi:10.1086/377498.
49. M. Aguilar et al. Precision Measurement of the Proton Flux in Primary Cosmic Rays from Rigidity 1 GV to 1.8 TV with the Alpha Magnetic Spectrometer on the International Space Station. Phys. Rev. Lett. **2015**, *114*, 171103. doi:10.1103/PhysRevLett.114.171103.
50. M. Aguilar et al. Precision Measurement of the Helium Flux in Primary Cosmic Rays of Rigidities 1.9 GV to 3 TV with the Alpha Magnetic Spectrometer on the International Space Station. Phys. Rev. Lett. **2015**, *115*, 211101. doi:10.1103/PhysRevLett.115.211101.
51. M. Aguilar et al. Properties of Iron Primary Cosmic Rays: Results from the Alpha Magnetic Spectrometer. *Phys. Rev. Lett.* **2021**, *126*, 041104. doi:10.1103/PhysRevLett.126.041104.
52. M. Aguilar et al. Properties of Heavy Secondary Fluorine Cosmic Rays: Results from the Alpha Magnetic Spectrometer. *Phys. Rev. Lett.* **2021**, *126*, 081102. doi:10.1103/PhysRevLett.126.081102.
53. M. Aguilar et al. Properties of Neon, Magnesium, and Silicon Primary Cosmic Rays Results from the Alpha Magnetic Spectrometer. *Phys. Rev. Lett.* **2020**, *124*, 211102. doi:10.1103/PhysRevLett.124.211102.
54. M. Aguilar et al. Electron and Positron Fluxes in Primary Cosmic Rays Measured with the Alpha Magnetic Spectrometer on the International Space Station. Phys. Rev. Lett. **2014**, *113*, 121102. doi:10.1103/PhysRevLett.113.121102.
55. M. Aguilar et al. Observation of New Properties of Secondary Cosmic Rays Lithium, Beryllium, and Boron by the Alpha Magnetic Spectrometer on the International Space Station. Phys. Rev. Lett. **2018**, *120*, 021101. doi:10.1103/PhysRevLett.120.021101.
56. R. P. Murphy et al. Galactic Cosmic Ray Origins and OB Associations: Evidence from SuperTIGER Observations of Elements $_{26}$Fe through $_{40}$Zr. ApJ **2016**, *831*, 148, [arXiv:astro-ph.HE/1608.08183]. doi:10.3847/0004-637X/831/2/148.
57. A. C. Cummings et al. Galactic Cosmic Rays in the Local Interstellar Medium: Voyager 1 Observations and Model Results. ApJ **2016**, *831*, 18. doi:10.3847/0004-637X/831/1/18.
58. Q. An et al. Measurement of the cosmic-ray proton spectrum from 40 GeV to 100 TeV with the DAMPE satellite. *Sci. Adv.* **2019**, *5*, eaax3793, [arXiv:astro-ph.HE/1909.12860]. doi:10.1126/sciadv.aax3793.
59. A.D. Panov et al. Energy Spectra of Abundant Nuclei of Primary Cosmic Rays from the Data of ATIC-2 Experiment: Final Results. *Bull. Russ. Acad. Sci. Phys.* **2009**, *73*, 564–567, [arXiv:astro-ph.HE/1101.3246]. doi:10.3103/S1062873809050098.
60. E. Atkin et al. New Universal Cosmic-Ray Knee near a Magnetic Rigidity of 10 TV with the NUCLEON Space Observatory. *JETP Lett.* **2018**, *108*, 5–12, [arXiv:astro-ph.HE/1805.07119]. doi:10.1134/S0021364018130015.
61. F. Aharonian et al. Probing the ATIC peak in the cosmic-ray electron spectrum with H.E.S.S. *Astron. Astrophys.* **2009**, *508*, 561, [arXiv:astro-ph.HE/0905.0105]. doi:10.1051/0004-6361/200913323.
62. D. Kerszberg. The cosmic-ray electron spectrum measured with H.E.S.S. *International Cosmic Ray Conference, [CRI215] (2017)* **2017**.
63. O. Adriani et al. Extended Measurement of the Cosmic-Ray Electron and Positron Spectrum from 11 GeV to 4.8 TeV with the Calorimetric Electron Telescope on the International Space Station. *Phys. Rev. Lett.* **2018**, *120*, 261102, [arXiv:astro-ph.HE/1806.09728]. doi:10.1103/PhysRevLett.120.261102.
64. G. Ambrosi et al. Direct detection of a break in the teraelectronvolt cosmic-ray spectrum of electrons and positrons. *Nature* **2017**, *552*, 63–66, [arXiv:astro-ph.HE/1711.10981]. doi:10.1038/nature24475.
65. HI4PI Collaboration et al. HI4PI: A full-sky H I survey based on EBHIS and GASS. A&A **2016**, *594*, A116, [arXiv:astro-ph.GA/1610.06175]. doi:10.1051/0004-6361/201629178.
66. J. E. G. Peek et al. The GALFA-H I Survey Data Release 2. ApJS **2018**, *234*, 2. doi:10.3847/1538-4365/aa91d3.
67. T. M. Dame, Dap Hartmann, and P. Thaddeus. The Milky Way in Molecular Clouds: A New Complete CO Survey. ApJ **2001**, *547*, 792–813, [arXiv:astro-ph/astro-ph/0009217]. doi:10.1086/318388.
68. N. Furukawa et al. The Jet and Arc Molecular Clouds toward Westerlund 2, RCW 49, and HESS J1023-575 ^{12}CO and ^{13}CO (J = 2-1 and J = 1-0) observations with NANTEN2 and Mopra Telescope. ApJ **2014**, *781*, 70, [arXiv:astro-ph.GA/1401.4845]. doi:10.1088/0004-637X/781/2/70.
69. Catherine Braiding et al. The Mopra Southern Galactic Plane CO Survey—Data Release 3. PASA **2018**, *35*, e029, [arXiv:astro-ph.GA/1902.04249]. doi:10.1017/pasa.2018.18.
70. Alberto D. Bolatto, Mark Wolfire, and Adam K. Leroy. The CO-to-H_2 Conversion Factor. ARA&A **2013**, *51*, 207–268, [arXiv:astro-ph.GA/1301.3498]. doi:10.1146/annurev-astro-082812-140944.
71. T. J. Sodroski et al. A Three-dimensional Decomposition of the Infrared Emission from Dust in the Milky Way. ApJ **1997**, *480*, 173–187. doi:10.1086/303961.
72. J. M. Cordes and T. J. W. Lazio. NE2001. II. Using Radio Propagation Data to Construct a Model for the Galactic Distribution of Free Electrons. *arXiv e-prints* **2003**, pp. astro–ph/0301598, [arXiv:astro-ph/astro-ph/0301598].
73. Marta I. R. Alves et al. The HIPASS survey of the Galactic plane in radio recombination lines. MNRAS **2015**, *450*, 2025–2042, [arXiv:astro-ph.GA/1411.4497]. doi:10.1093/mnras/stv751.

74. Planck Collaboration et al. Planck 2018 results. IV. Diffuse component separation. A&A **2020**, *641*, A4, [arXiv:astro-ph.CO/1807.06208]. doi:10.1051/0004-6361/201833881.
75. Gregory M. Green, Edward Schlafly, Catherine Zucker, Joshua S. Speagle, and Douglas Finkbeiner. A 3D Dust Map Based on Gaia, Pan-STARRS 1, and 2MASS. ApJ **2019**, *887*, 93, [arXiv:astro-ph.GA/1905.02734]. doi:10.3847/1538-4357/ab5362.
76. R. Lallement et al. Gaia-2MASS 3D maps of Galactic interstellar dust within 3 kpc. A&A **2019**, *625*, A135, [arXiv:astro-ph.GA/1902.04116]. doi:10.1051/0004-6361/201834695.
77. Q. Remy, I. A. Grenier, D. J. Marshall, and J. M. Casandjian. Cosmic-rays, gas, and dust in nearby anti-centre clouds. III. Dust extinction, emission, and grain properties. A&A **2018**, *616*, A71, [arXiv:astro-ph.HE/1803.05686]. doi:10.1051/0004-6361/201731488.
78. S. A. Balashev et al. CO-dark molecular gas at high redshift: very large H_2 content and high pressure in a low-metallicity damped Lyman alpha system. MNRAS **2017**, *470*, 2890–2910, [arXiv:astro-ph.GA/1705.10283]. doi:10.1093/mnras/stx1339.
79. Philip D. Engelke and Ronald J. Allen. OH as an Alternate Tracer for Molecular Gas: Quantity and Structure of Molecular Gas in W5. ApJ **2019**, *874*, 49, [arXiv:astro-ph.GA/1901.02515]. doi:10.3847/1538-4357/aafc29.
80. Philip D. Engelke, Ronald J. Allen, and Michael P. Busch. Star-forming versus Quiescent Regions in the Galaxy: A Case Study of ISM Properties Based on 18 cm OH and ^{12}CO (1-0) Observations. ApJ **2020**, *901*, 50. doi:10.3847/1538-4357/abaeeb.
81. Rowan J. Smith, Simon C. O. Glover, Paul C. Clark, Ralf S. Klessen, and Volker Springel. CO-dark gas and molecular filaments in Milky Way-type galaxies. MNRAS **2014**, *441*, 1628–1645, [arXiv:astro-ph.GA/1403.1589]. doi:10.1093/mnras/stu616.
82. Silvia Vernetto and Paolo Lipari. Absorption of very high energy gamma rays in the Milky Way. Phys. Rev. D **2016**, *94*, 063009, [arXiv:astro-ph.HE/1608.01587]. doi:10.1103/PhysRevD.94.063009.
83. C. C. Popescu et al. A radiation transfer model for the Milky Way: I. Radiation fields and application to high-energy astrophysics? MNRAS **2017**, *470*, 2539–2558, [arXiv:astro-ph.GA/1705.06652]. doi:10.1093/mnras/stx1282.
84. T. A. Porter, G. Jóhannesson, and I. V. Moskalenko. High-energy Gamma Rays from the Milky Way: Three-dimensional Spatial Models for the Cosmic-Ray and Radiation Field Densities in the Interstellar Medium. ApJ **2017**, *846*, 67, [arXiv:astro-ph.HE/1708.00816]. doi:10.3847/1538-4357/aa844d.
85. C. C. Popescu et al. Modelling the spectral energy distribution of galaxies. V. The dust and PAH emission SEDs of disk galaxies. A&A **2011**, *527*, A109, [arXiv:astro-ph.CO/1011.2942]. doi:10.1051/0004-6361/201015217.
86. Déborah Paradis et al. A Statistical Study of Dust Properties in Large Magellanic Cloud Molecular Clouds. AJ **2011**, *141*, 43, [arXiv:astro-ph.GA/1010.3134]. doi:10.1088/0004-6256/141/2/43.
87. F. Niederwanger et al. The consequence of a new ISRF model of the Milky Way on predictions for diffuse gamma-ray emission. *Astroparticle Physics* **2019**, *107*, 1–14, [arXiv:astro-ph.HE/1811.12683]. doi:10.1016/j.astropartphys.2018.11.001.
88. Rainer Beck. Magnetic fields in spiral galaxies. A&A Rev. **2015**, *24*, 4, [arXiv:astro-ph.GA/1509.04522]. doi:10.1007/s00159-015-0084-4.
89. Xiao-Hui Sun and Wolfgang Reich. The Galactic halo magnetic field revisited. *Research in Astronomy and Astrophysics* **2010**, *10*, 1287–1297, [arXiv:astro-ph.GA/1010.4394]. doi:10.1088/1674-4527/10/12/009.
90. L. Fauvet et al. Expected constraints on the Galactic magnetic field using Planck data. A&A **2012**, *540*, A122, [arXiv:astro-ph.GA/1201.5742]. doi:10.1051/0004-6361/201016349.
91. Ronnie Jansson and Glennys R. Farrar. A New Model of the Galactic Magnetic Field. ApJ **2012**, *757*, 14, [arXiv:astro-ph.GA/1204.3662]. doi:10.1088/0004-637X/757/1/14.
92. T. R. Jaffe et al. Comparing polarized synchrotron and thermal dust emission in the Galactic plane. MNRAS **2013**, *431*, 683–694, [arXiv:astro-ph.GA/1302.0143]. doi:10.1093/mnras/stt200.
93. Elena Orlando and Andrew Strong. Galactic synchrotron emission with cosmic ray propagation models. MNRAS **2013**, *436*, 2127–2142, [arXiv:astro-ph.GA/1309.2947]. doi:10.1093/mnras/stt1718.
94. Planck Collaboration et al. Planck intermediate results. XLII. Large-scale Galactic magnetic fields. A&A **2016**, *596*, A103, [arXiv:astro-ph.GA/1601.00546]. doi:10.1051/0004-6361/201528033.
95. K. Ferrière. Plasma turbulence in the interstellar medium. *Plasma Physics and Controlled Fusion* **2020**, *62*, 014014, [arXiv:astro-ph.GA/1912.08237]. doi:10.1088/1361-6587/ab49eb.
96. A. Chepurnov and A. Lazarian. Extending the Big Power Law in the Sky with Turbulence Spectra from Wisconsin Hα Mapper Data. ApJ **2010**, *710*, 853–858, [arXiv:astro-ph.GA/0905.4413]. doi:10.1088/0004-637X/710/1/853.
97. E. Orlando. Imprints of cosmic rays in multifrequency observations of the interstellar emission. MNRAS **2018**, *475*, 2724–2742, [arXiv:astro-ph.HE/1712.07127]. doi:10.1093/mnras/stx3280.
98. E. Orlando. Implications on spatial models of interstellar gamma-ray inverse-Compton emission from synchrotron emission studies in radio and microwaves. Phys. Rev. D **2019**, *99*, 043007, [arXiv:astro-ph.HE/1901.08604]. doi:10.1103/PhysRevD.99.043007.
99. IceCube Collaboration. Evidence for High-Energy Extraterrestrial Neutrinos at the IceCube Detector. *Science* **2013**, *342*, 1242856, [arXiv:astro-ph.HE/1311.5238]. doi:10.1126/science.1242856.
100. A. Albert et al. Joint Constraints on Galactic Diffuse Neutrino Emission from the ANTARES and IceCube Neutrino Telescopes. ApJ **2018**, *868*, L20, [arXiv:astro-ph.HE/1808.03531]. doi:10.3847/2041-8213/aaeecf.

101. David A. Neufeld and Mark G. Wolfire. The Cosmic-Ray Ionization Rate in the Galactic Disk, as Determined from Observations of Molecular Ions. ApJ **2017**, *845*, 163, [arXiv:astro-ph.GA/1704.03877]. doi:10.3847/1538-4357/aa6d68.
102. V. H. M. Phan, G. Morlino, and S. Gabici. What causes the ionization rates observed in diffuse molecular clouds? The role of cosmic ray protons and electrons. MNRAS **2018**, *480*, 5167–5174, [arXiv:astro-ph.HE/1804.10106]. doi:10.1093/mnras/sty2235.
103. S. Recchia, V. H. M. Phan, S. Biswas, and S. Gabici. Can a cosmic ray carrot explain the ionization level in diffuse molecular clouds? MNRAS **2019**, *485*, 2276–2280, [arXiv:astro-ph.HE/1901.04912]. doi:10.1093/mnras/stz555.
104. Gargi Shaw and G. J. Ferland. Roll of Polycyclic Aromatic Hydrocarbons on the Cosmic-Ray ionization rate in the Galaxy. *arXiv e-prints* **2021**, p. arXiv:2101.03732, [arXiv:astro-ph.GA/2101.03732].
105. Vincent Tatischeff and Jürgen Kiener. γ-ray lines from cosmic-ray interactions with interstellar dust grains. New A Rev. **2004**, *48*, 99–103, [arXiv:astro-ph/astro-ph/0310827]. doi:10.1016/j.newar.2003.11.013.
106. Bing Liu, Rui-zhi Yang, and Felix Aharonian. Nuclear de-excitation lines as a probe of low-energy cosmic rays. *arXiv e-prints* **2021**, p. arXiv:2101.03695, [arXiv:astro-ph.HE/2101.03695].
107. Elena Orlando et al. Cosmic Rays and interstellar medium with Gamma-Ray Observations at MeV Energies. BAAS **2019**, *51*, 151, [arXiv:astro-ph.HE/1903.05660].
108. Gaia Collaboration et al. Gaia Early Data Release 3: Summary of the contents and survey properties. *arXiv e-prints* **2020**, p. arXiv:2012.01533, [arXiv:astro-ph.GA/2012.01533].
109. T. Cantat-Gaudin et al. Painting a portrait of the Galactic disc with its stellar clusters. A&A **2020**, *640*, A1, [arXiv:astro-ph.GA/2004.07274]. doi:10.1051/0004-6361/202038192.
110. N. V. Kharchenko, A. E. Piskunov, E. Schilbach, S. Röser, and R. D. Scholz. Global survey of star clusters in the Milky Way. II. The catalogue of basic parameters. A&A **2013**, *558*, A53, [arXiv:astro-ph.GA/1308.5822]. doi:10.1051/0004-6361/201322302.
111. M. J. Reid et al. Trigonometric Parallaxes of High-mass Star-forming Regions: Our View of the Milky Way. ApJ **2019**, *885*, 131, [arXiv:astro-ph.GA/1910.03357]. doi:10.3847/1538-4357/ab4a11.
112. S. R. Berlanas, N. J. Wright, A. Herrero, J. E. Drew, and D. J. Lennon. Disentangling the spatial substructure of Cygnus OB2 from Gaia DR2. MNRAS **2019**, *484*, 1838–1842, [arXiv:astro-ph.SR/1901.02959]. doi:10.1093/mnras/stz117.
113. Emma R. Beasor, Ben Davies, Nathan Smith, Robert D. Gehrz, and Donald F. Figer. The Age of Westerlund 1 Revisited. *arXiv e-prints* **2021**, p. arXiv:2103.02609, [arXiv:astro-ph.SR/2103.02609].
114. R. Braun, T. Bourke, J. A. Green, E. Keane, and J. Wagg. Advancing Astrophysics with the Square Kilometre Array. Advancing Astrophysics with the Square Kilometre Array (AASKA14), 2015, p. 174.
115. A. Merloni et al. eROSITA Science Book: Mapping the Structure of the Energetic Universe. *arXiv e-prints* **2012**, p. arXiv:1209.3114, [arXiv:astro-ph.HE/1209.3114].
116. C. D. Dermer et al. On the Physics Connecting Cosmic Rays and Gamma Rays: Towards Determining the Interstellar Cosmic Ray Spectrum. *arXiv e-prints* **2013**, p. arXiv:1303.6482, [arXiv:astro-ph.HE/1303.6482].
117. W. O. Lock and D. F. Measday. *Intermediate-energy Nuclear Physics*; Methuen: York, 1970.
118. F. W. Stecker. Neutral-Pion Gamma Rays from the Galaxy and the Interstellar Gas Content. ApJ **1973**, *185*, 499–504. doi:10.1086/152435.
119. C. D. Dermer. Binary Collision Rates of Relativistic Thermal Plasmas. II. Spectra. ApJ **1986**, *307*, 47. doi:10.1086/164391.
120. O. Adriani et al. Early results of the LHCf experiment and their contribution to ultra-high-energy cosmic ray physics. *Nuclear Physics B Proceedings Supplements* **2011**, *212*, 270–276, [arXiv:hep-ex/1012.1490]. doi:10.1016/j.nuclphysbps.2011.03.037.
121. H. Sato, T. Shibata, and R. Yamazaki. Revised production cross-section of γ-rays in p-p collisions with LHC data for the study of TeV γ-ray astronomy. *Astroparticle Physics* **2012**, *36*, 83–92, [arXiv:astro-ph.HE/1205.2145]. doi:10.1016/j.astropartphys.2012.05.003.
122. A. Aduszkiewicz et al. Measurement of meson resonance production in π ++ C interactions at SPS energies. *European Physical Journal C* **2017**, *77*, 626, [arXiv:nucl-ex/1705.08206]. doi:10.1140/epjc/s10052-017-5184-z.
123. Tuneyoshi Kamae, Niklas Karlsson, Tsunefumi Mizuno, Toshinori Abe, and Tatsumi Koi. Parameterization of γ, $e^{+/-}$, and Neutrino Spectra Produced by p-p Interaction in Astronomical Environments. ApJ **2006**, *647*, 692–708, [arXiv:astro-ph/astro-ph/0605581]. doi:10.1086/505189.
124. S. R. Kelner, F. A. Aharonian, and V. V. Bugayov. Energy spectra of gamma rays, electrons, and neutrinos produced at proton-proton interactions in the very high energy regime. Phys. Rev. D **2006**, *74*, 034018, [arXiv:astro-ph/astro-ph/0606058]. doi:10.1103/PhysRevD.74.034018.
125. M. Kachelrieß and S. Ostapchenko. Deriving the cosmic ray spectrum from gamma-ray observations. Phys. Rev. D **2012**, *86*, 043004, [arXiv:astro-ph.HE/1206.4705]. doi:10.1103/PhysRevD.86.043004.
126. Ervin Kafexhiu, Felix Aharonian, Andrew M. Taylor, and Gabriela S. Vila. Parametrization of gamma-ray production cross sections for p p interactions in a broad proton energy range from the kinematic threshold to PeV energies. Phys. Rev. D **2014**, *90*, 123014, [arXiv:astro-ph.HE/1406.7369]. doi:10.1103/PhysRevD.90.123014.
127. M. N. Mazziotta et al. Production of secondary particles and nuclei in cosmic rays collisions with the interstellar gas using the FLUKA code. *Astroparticle Physics* **2016**, *81*, 21–38, [arXiv:astro-ph.HE/1510.04623]. doi:10.1016/j.astropartphys.2016.04.005.

128. M. Kachelrieß, I. V. Moskalenko, and S. Ostapchenko. AAfrag: Interpolation routines for Monte Carlo results on secondary production in proton-proton, proton-nucleus and nucleus-nucleus interactions. *Computer Physics Communications* **2019**, *245*, 106846, [arXiv:hep-ph/1904.05129]. doi:10.1016/j.cpc.2019.08.001.
129. Masaki Mori. Nuclear enhancement factor in calculation of Galactic diffuse gamma-rays: A new estimate with DPMJET-3. *Astroparticle Physics* **2009**, *31*, 341–343, [arXiv:astro-ph.HE/0903.3260]. doi:10.1016/j.astropartphys.2009.03.004.
130. Michael Kachelrieß, Igor V. Moskalenko, and Sergey S. Ostapchenko. Nuclear Enhancement of the Photon Yield in Cosmic Ray Interactions. ApJ **2014**, *789*, 136, [arXiv:astro-ph.HE/1406.0035]. doi:10.1088/0004-637X/789/2/136.
131. R. D. Blandford and J. P. Ostriker. Particle acceleration by astrophysical shocks. *The Astrophysical Journal Letters* **1978**, *221*, L29–L32. doi:10.1086/182658.
132. A. R. Bell. The acceleration of cosmic rays in shock fronts. I. *MNRAS* **1978**, *182*, 147–156. doi:10.1093/mnras/182.2.147.
133. W. I. Axford, E. Leer, and G. Skadron. The acceleration of cosmic rays by shock waves. *International Cosmic Ray Conference* **1977**, *11*, 132–137.
134. G. F. Krymskii. A regular mechanism for the acceleration of charged particles on the front of a shock wave. *Akademiia Nauk SSSR Doklady* **1977**, *234*, 1306–1308.
135. Pasquale Blasi. A semi-analytical approach to non-linear shock acceleration. *Astroparticle Physics* **2002**, *16*, 429–439, [arXiv:astro-ph/astro-ph/0104064]. doi:10.1016/S0927-6505(01)00127-X.
136. Pasquale Blasi. The origin of galactic cosmic rays. A&A Rev. **2013**, *21*, 70, [arXiv:astro-ph.HE/1311.7346]. doi:10.1007/s00159-013-0070-7.
137. T. Antoni et al. Large-Scale Cosmic-Ray Anisotropy with KASCADE. *The Astrophysical Journal* **2004**, *604*, 687.
138. A. A. Abdo et al. The Large-Scale Cosmic-Ray Anisotropy as Observed with Milagro. *The Astrophysical Journal* **2009**, *698*, 2121.
139. M. Aglietta et al. Evolution of the Cosmic-Ray Anisotropy Above 1014 eV. *The Astrophysical Journal Letters* **2009**, *692*, L130.
140. M. Amenomori et al. Northern Sky Galactic Cosmic Ray Anisotropy between 10 and 1000 TeV with the Tibet Air Shower Array. *The Astrophysical Journal* **2017**, *836*, 153.
141. M. G. Aartsen et al. Observation of Cosmic-Ray Anisotropy with the IceTop Air Shower Array. *The Astrophysical Journal* **2013**, *765*, 55.
142. R. Abbasi et al. Observation of Anisotropy in the Galactic Cosmic-Ray Arrival Directions at 400 TeV with IceCube. *The Astrophysical Journal* **2012**, *746*, 33.
143. R. Cowsik, Yash Pal, S. N. Tandon, and R. P. Verma. Steady State of Cosmic-Ray Nuclei-Their Spectral Shape and Path Length at Low Energies. *Physical Review* **1967**, *158*, 1238–1242. doi:10.1103/PhysRev.158.1238.
144. J. R. Jokipii. Cosmic-Ray Propagation. I. Charged Particles in a Random Magnetic Field. ApJ **1966**, *146*, 480. doi:10.1086/148912.
145. J. R. Jokipii and E. N. Parker. Random Walk of Magnetic Lines of Force in Astrophysics. *Physical Review Letters* **1968**, *21*, 44–47. doi:10.1103/PhysRevLett.21.44.
146. V.S. Berezinsky, S.V. Bulanov, V.A. Dogiel, and V.S. Ptuskin. *Astrophysics of cosmic rays*; 1990.
147. Luke O'C. Drury and Andrew W. Strong. Power requirements for cosmic ray propagation models involving diffusive reacceleration; estimates and implications for the damping of interstellar turbulence. *Astron. Astrophys.* **2017**, *597*, A117, [arXiv:astro-ph.HE/1608.04227]. doi:10.1051/0004-6361/201629526.
148. C. Evoli et al. Cosmic-ray propagation with DRAGON2: I. numerical solver and astrophysical ingredients. *JCAP* **2017**, *2*, 015, [arXiv:astro-ph.HE/1607.07886]. doi:10.1088/1475-7516/2017/02/015.
149. R. Silberberg and C.H. Tsao. Spallation processes and nuclear interaction products of cosmic rays. *Physics Reports* **1990**, *191*, 351 – 408. doi:https://doi.org/10.1016/0370-1573(90)90109-F.
150. A. W. Strong and I. V. Moskalenko. Models for galactic cosmic ray propagation. *Adv. Space Res.* **2001**, *27*, 717–726, [astro-ph/0101068]. doi:10.1016/S0273-1177(01)00112-0.
151. S. G. Mashnik, A. J. Sierk, K. A. Van Riper, and W. B. Wilson. Production and validation of isotope production cross-section libraries for neutrons and protons to 1.7-GeV. 4th Workshop on Simulating Accelerator Radiation Environments (SARE4) (A Satellite Meeting of ACCAPP 98), 1998, [nucl-th/9812071].
152. I. V. Moskalenko, S. G. Mashnik, and A. W. Strong. New calculation of radioactive secondaries in cosmic rays. 27th International Cosmic Ray Conference, 2001, Vol. 5, p. 1836, [astro-ph/0106502].
153. Igor V. Moskalenko and S. G. Mashnik. Evaluation of production cross sections of Li, Be, B in CR. 28th International Cosmic Ray Conference, 2003, pp. 1969–1972, [astro-ph/0306367].
154. C. Evoli et al. Cosmic-ray propagation with DRAGON2: II. Nuclear interactions with the interstellar gas. *ArXiv e-prints* **2017**, [arXiv:astro-ph.HE/1711.09616].
155. F. Lebrun et al. Gamma-rays from atomic and molecular gas in the first galactic quadrant. ApJ **1983**, *274*, 231–236. doi:10.1086/161440.
156. Emma Storm, Christoph Weniger, and Francesca Calore. SkyFACT: High-dimensional modeling of gamma-ray emission with adaptive templates and penalized likelihoods. *JCAP* **2017**, *08*, 022, [arXiv:astro-ph.HE/1705.04065]. doi:10.1088/1475-7516/2017/08/022.

157. Dmitry Malyshev. Spectral components analysis of diffuse emission processes. *arXiv e-prints* **2012**, p. arXiv:1202.1034, [arXiv:astro-ph.IM/1202.1034].
158. Wim de Boer, Léo Bosse, Iris Gebauer, Alexander Neumann, and Peter L. Biermann. Molecular clouds as origin of the Fermi gamma-ray GeV excess. Phys. Rev. D **2017**, *96*, 043012, [arXiv:astro-ph.HE/1707.08653]. doi:10.1103/PhysRevD.96.043012.
159. Marco Selig, Valentina Vacca, Niels Oppermann, and Torsten A. Enßlin. The denoised, deconvolved, and decomposed Fermi γ-ray sky - An application of the D^{3}PO algorithm. *Astron. Astrophys.* **2015**, *581*, A126, [arXiv:astro-ph.HE/1410.4562]. doi:10.1051/0004-6361/201425172.
160. A. W. Strong and G. Youssefi. Propagation Models For Cr Nucleons And Electrons And Predictions Of The Galactic Gamma-ray Spectrum. *International Cosmic Ray Conference* **1995**, *3*, 48.
161. A. W. Strong and I. V. Moskalenko. Propagation of cosmic-ray nucleons in the galaxy. *Astrophys. J.* **1998**, *509*, 212–228, [arXiv:astro-ph/astro-ph/9807150]. doi:10.1086/306470.
162. I. V. Moskalenko and A. W. Strong. Production and propagation of cosmic ray positrons and electrons. *Astrophys. J.* **1998**, *493*, 694–707, [arXiv:astro-ph/astro-ph/9710124]. doi:10.1086/305152.
163. Carmelo Evoli, Daniele Gaggero, Dario Grasso, and Luca Maccione. Cosmic-Ray Nuclei, Antiprotons and Gamma-rays in the Galaxy: a New Diffusion Model. *JCAP* **2008**, *0810*, 018, [arXiv:astro-ph/0807.4730]. doi:10.1088/1475-7516/2008/10/018.
164. Daniele Gaggero, Luca Maccione, Giuseppe Di Bernardo, Carmelo Evoli, and Dario Grasso. Three-Dimensional Model of Cosmic-Ray Lepton Propagation Reproduces Data from the Alpha Magnetic Spectrometer on the International Space Station. *Phys. Rev. Lett.* **2013**, *111*, 021102, [arXiv:astro-ph.HE/1304.6718]. doi:10.1103/PhysRevLett.111.021102.
165. Ralf Kissmann. PICARD: A novel code for the Galactic Cosmic Ray propagation problem. *Astropart. Phys.* **2014**, *55*, 37–50, [arXiv:astro-ph.HE/1401.4035]. doi:10.1016/j.astropartphys.2014.02.002.
166. M. Werner, R. Kissmann, A. W. Strong, and O. Reimer. Spiral Arms as Cosmic Ray Source Distributions. *Astropart. Phys.* **2015**, *64*, 18–33, [arXiv:astro-ph.HE/1410.5266]. doi:10.1016/j.astropartphys.2014.10.005.
167. D. Maurin, F. Donato, R. Taillet, and P. Salati. Cosmic rays below z=30 in a diffusion model: new constraints on propagation parameters. *Astrophys. J.* **2001**, *555*, 585–596, [arXiv:astro-ph/astro-ph/0101231]. doi:10.1086/321496.
168. M. Aguilar et al. Precision Measurement of Cosmic-Ray Nitrogen and its Primary and Secondary Components with the Alpha Magnetic Spectrometer on the International Space Station. *Phys. Rev. Lett.* **2018**, *121*, 051103. doi:10.1103/PhysRevLett.121.051103.
169. Oscar Adriani et al. An anomalous positron abundance in cosmic rays with energies 1.5-100 GeV. *Nature* **2009**, *458*, 607–609, [arXiv:astro-ph/0810.4995]. doi:10.1038/nature07942.
170. M. Aguilar et al. First Result from the Alpha Magnetic Spectrometer on the International Space Station: Precision Measurement of the Positron Fraction in Primary Cosmic Rays of 0.5–350 GeV. *Phys. Rev. Lett.* **2013**, *110*, 141102. doi:10.1103/PhysRevLett.110.141102.
171. Alessandro Cuoco, Michael Krämer, and Michael Korsmeier. Novel Dark Matter Constraints from Antiprotons in Light of AMS-02. *Phys. Rev. Lett.* **2017**, *118*, 191102, [arXiv:astro-ph.HE/1610.03071]. doi:10.1103/PhysRevLett.118.191102.
172. Alessandro Cuoco, Jan Heisig, Lukas Klamt, Michael Korsmeier, and Michael Krämer. Scrutinizing the evidence for dark matter in cosmic-ray antiprotons. *Phys. Rev. D* **2019**, *99*, 103014, [arXiv:astro-ph.HE/1903.01472]. doi:10.1103/PhysRevD.99.103014.
173. R. Kissmann, M. Werner, O. Reimer, and A.W. Strong. Propagation in 3D spiral-arm cosmic-ray source distribution models and secondary particle production using *PICARD*. *Astropart. Phys.* **2015**, *70*, 39–53, [arXiv:astro-ph.HE/1504.08249]. doi:10.1016/j.astropartphys.2015.04.003.
174. R. Kissmann, F. Niederwanger, O. Reimer, and A.W. Strong. Diffuse gamma rays in 3D galactic cosmic-ray propagation models. *AIP Conf. Proc.* **2017**, *1792*, 070011, [arXiv:astro-ph.HE/1701.07285]. doi:10.1063/1.4969008.
175. Troy A. Porter, Gudlaugur Johannesson, and Igor V. Moskalenko. High-Energy Gamma Rays from the Milky Way: Three-Dimensional Spatial Models for the Cosmic-Ray and Radiation Field Densities in the Interstellar Medium. *Astrophys. J.* **2017**, *846*, 67, [arXiv:astro-ph.HE/1708.00816]. doi:10.3847/1538-4357/aa844d.
176. Carmelo Evoli, Daniele Gaggero, Dario Grasso, and Luca Maccione. Common Solution to the Cosmic Ray Anisotropy and Gradient Problems. Phys. Rev. Lett. **2012**, *108*, 211102, [arXiv:astro-ph.HE/1203.0570]. doi:10.1103/PhysRevLett.108.211102.
177. G. Jóhannesson et al. Bayesian analysis of cosmic-ray propagation: evidence against homogeneous diffusion. *Astrophys. J.* **2016**, *824*, 16, [arXiv:astro-ph.HE/1602.02243]. doi:10.3847/0004-637X/824/1/16.
178. Silvio Sergio Cerri, Daniele Gaggero, Andrea Vittino, Carmelo Evoli, and Dario Grasso. A signature of anisotropic cosmic-ray transport in the gamma-ray sky. *JCAP* **2017**, *10*, 019, [arXiv:astro-ph.HE/1707.07694]. doi:10.1088/1475-7516/2017/10/019.
179. D.G. Wentzel. Cosmic-ray propagation in the galaxy: collective effects. *Ann. Rev. Astron. Astrophys.* **1974**, *12*, 71–96. doi:10.1146/annurev.aa.12.090174.000443.
180. Alison J. Farmer and Peter Goldreich. Wave damping by MHD turbulence and its effect upon cosmic ray propagation in the ISM. *Astrophys. J.* **2004**, *604*, 671–674, [astro-ph/0311400]. doi:10.1086/382040.
181. Pasquale Blasi, Elena Amato, and Pasquale D. Serpico. Spectral breaks as a signature of cosmic ray induced turbulence in the Galaxy. *Phys. Rev. Lett.* **2012**, *109*, 061101, [arXiv:astro-ph.HE/1207.3706]. doi:10.1103/PhysRevLett.109.061101.

182. Carmelo Evoli, Pasquale Blasi, Giovanni Morlino, and Roberto Aloisio. Origin of the Cosmic Ray Galactic Halo Driven by Advected Turbulence and Self-Generated Waves. *Phys. Rev. Lett.* **2018**, *121*, 021102, [arXiv:astro-ph.HE/1806.04153]. doi:10.1103/PhysRevLett.121.021102.
183. S. Sridhar and P. Goldreich. Toward a Theory of Interstellar Turbulence. I. Weak Alfvenic Turbulence. ApJ **1994**, *432*, 612. doi:10.1086/174600.
184. P. Goldreich and S. Sridhar. Toward a Theory of Interstellar Turbulence. II. Strong Alfvenic Turbulence. ApJ **1995**, *438*, 763. doi:10.1086/175121.
185. Benjamin D. G. Chandran. Scattering of Energetic Particles by Anisotropic Magnetohydrodynamic Turbulence with a Goldreich-Sridhar Power Spectrum. Phys. Rev. Lett. **2000**, *85*, 4656–4659, [arXiv:astro-ph/astro-ph/0008498]. doi:10.1103/PhysRevLett.85.4656.
186. Huirong Yan and A. Lazarian. Scattering of Cosmic Rays by Magnetohydrodynamic Interstellar Turbulence. *Phys. Rev. Lett.* **2002**, *89*, 281102. doi:10.1103/PhysRevLett.89.281102.
187. Huirong Yan and Alex Lazarian. Cosmic ray scattering and streaming in compressible magnetohydrodynamic turbulence. *Astrophys. J.* **2004**, *614*, 757–769, [astro-ph/0408172]. doi:10.1086/423733.
188. Huirong Yan and A. Lazarian. Cosmic Ray Propagation: Nonlinear Diffusion Parallel and Perpendicular to Mean Magnetic Field. *Astrophys. J.* **2008**, *673*, 942, [arXiv:astro-ph/0710.2617]. doi:10.1086/524771.
189. Carmelo Evoli and Huirong Yan. Cosmic ray propagation in galactic turbulence. *Astrophys. J.* **2014**, *782*, 36, [arXiv:astro-ph.HE/1310.5732]. doi:10.1088/0004-637X/782/1/36.
190. Ottavio Fornieri, Daniele Gaggero, Silvio Sergio Cerri, Pedro De La Torre Luque, and Stefano Gabici. The theory of cosmic-ray scattering on pre-existing MHD modes meets data. MNRAS **2021**, [arXiv:astro-ph.HE/2011.09197]. doi:10.1093/mnras/stab355.
191. Heshou Zhang et al. Identification of plasma modes in Galactic turbulence with synchrotron polarization. *Nature Astronomy* **2020**, *4*, 1001–1008, [arXiv:physics.plasm-ph/1808.01913]. doi:10.1038/s41550-020-1093-4.
192. V. S. Ptuskin, F. C. Jones, E. S. Seo, and R. Sina. Effect of random nature of cosmic ray sources Supernova remnants on cosmic ray intensity fluctuations, anisotropy, and electron energy spectrum. *Advances in Space Research* **2006**, *37*, 1909–1912. doi:10.1016/j.asr.2005.08.036.
193. I. Büsching et al. Cosmic-Ray Propagation Properties for an Origin in Supernova Remnants. ApJ **2005**, *619*, 314–326, [arXiv:astro-ph/astro-ph/0410352]. doi:10.1086/426537.
194. Pasquale Blasi and Elena Amato. Diffusive propagation of cosmic rays from supernova remnants in the Galaxy. I: spectrum and chemical composition. J. Cosmology Astropart. Phys. **2012**, *2012*, 010, [arXiv:astro-ph.HE/1105.4521]. doi:10.1088/1475-7516/2012/01/010.
195. Carmelo Evoli, Giovanni Morlino, Pasquale Blasi, and Roberto Aloisio. AMS-02 beryllium data and its implication for cosmic ray transport. Phys. Rev. D **2020**, *101*, 023013, [arXiv:astro-ph.HE/1910.04113]. doi:10.1103/PhysRevD.101.023013.
196. A. W. Strong. Source population synthesis and the Galactic diffuse gamma-ray emission. Ap&SS **2007**, *309*, 35–41, [arXiv:astro-ph/astro-ph/0609359]. doi:10.1007/s10509-007-9480-1.
197. S. Casanova and B. L. Dingus. Constraints on the TeV source population and its contribution to the galactic diffuse TeV emission. *Astroparticle Physics* **2008**, *29*, 63–69, [arXiv:astro-ph/0711.2753]. doi:10.1016/j.astropartphys.2007.11.008.
198. Constantin Steppa and Kathrin Egberts. Modelling the Galactic very-high-energy γ-ray source population. A&A **2020**, *643*, A137, [arXiv:astro-ph.HE/2010.03305]. doi:10.1051/0004-6361/202038172.
199. Richard Bartels, Suraj Krishnamurthy, and Christoph Weniger. Strong Support for the Millisecond Pulsar Origin of the Galactic Center GeV Excess. Phys. Rev. Lett. **2016**, *116*, 051102, [arXiv:astro-ph.HE/1506.05104]. doi:10.1103/PhysRevLett.116.051102.
200. Rebecca K. Leane and Tracy R. Slatyer. Spurious Point Source Signals in the Galactic Center Excess. Phys. Rev. Lett. **2020**, *125*, 121105, [arXiv:astro-ph.HE/2002.12370]. doi:10.1103/PhysRevLett.125.121105.
201. Jean-Marc Casandjian. Local H i Emissivity Measured with Fermi-LAT and Implications for Cosmic-Ray Spectra. ApJ **2015**, *806*, 240, [arXiv:astro-ph.HE/1506.00047]. doi:10.1088/0004-637X/806/2/240.
202. Planck Collaboration et al. Planck intermediate results. XXVIII. Interstellar gas and dust in the Chamaeleon clouds as seen by Fermi LAT and Planck. A&A **2015**, *582*, A31, [arXiv:astro-ph.HE/1409.3268]. doi:10.1051/0004-6361/201424955.
203. Q. Remy, I. A. Grenier, D. J. Marshall, and J. M. Casand jian. Cosmic rays, gas and dust in nearby anticentre clouds. I. CO-to-H_2 conversion factors and dust opacities. A&A **2017**, *601*, A78, [arXiv:astro-ph.HE/1703.05237]. doi:10.1051/0004-6361/201629632.
204. T. Joubaud, I. A. Grenier, J. M. Casandjian, T. Tolksdorf, and R. Schlickeiser. The cosmic-ray content of the Orion-Eridanus superbubble. A&A **2020**, *635*, A96, [arXiv:astro-ph.HE/2001.10139]. doi:10.1051/0004-6361/201937205.
205. A. A. Abdo et al. Fermi LAT Observation of Diffuse Gamma Rays Produced Through Interactions Between Local Interstellar Matter and High-energy Cosmic Rays. ApJ **2009**, *703*, 1249–1256, [arXiv:astro-ph.HE/0908.1171]. doi:10.1088/0004-637X/703/2/1249.
206. A. A. Abdo et al. Fermi Observations of Cassiopeia and Cepheus: Diffuse Gamma-ray Emission in the Outer Galaxy. ApJ **2010**, *710*, 133–149, [arXiv:astro-ph.HE/0912.3618]. doi:10.1088/0004-637X/710/1/133.
207. M. Ackermann et al. Constraints on the Cosmic-ray Density Gradient Beyond the Solar Circle from Fermi γ-ray Observations of the Third Galactic Quadrant. ApJ **2011**, *726*, 81, [arXiv:astro-ph.HE/1011.0816]. doi:10.1088/0004-637X/726/2/81.

208. M. Ackermann et al. The cosmic-ray and gas content of the Cygnus region as measured in γ-rays by the Fermi Large Area Telescope. A&A **2012**, *538*, A71, [arXiv:astro-ph.HE/1110.6123]. doi:10.1051/0004-6361/201117539.
209. M. Ackermann et al. Gamma-Ray Observations of the Orion Molecular Clouds with the Fermi Large Area Telescope. ApJ **2012**, *756*, 4, [arXiv:astro-ph.HE/1207.0616]. doi:10.1088/0004-637X/756/1/4.
210. M. Ackermann et al. Fermi Large Area Telescope Study of Cosmic Rays and the Interstellar Medium in nearby Molecular Clouds. ApJ **2012**, *755*, 22, [arXiv:astro-ph.HE/1207.6275]. doi:10.1088/0004-637X/755/1/22.
211. T. Mizuno et al. Quantifying the Interstellar Medium and Cosmic Rays in the MBM 53, 54, and 55 Molecular Clouds and the Pegasus Loop Using Fermi-LAT Gamma-ray Observations. ApJ **2016**, *833*, 278, [arXiv:astro-ph.HE/1610.08596]. doi:10.3847/1538-4357/833/2/278.
212. T. Mizuno et al. Study of the Cosmic Rays and Interstellar Medium in Local H I Clouds Using Fermi-LAT Gamma-Ray Observations. ApJ **2020**, *890*, 120, [arXiv:astro-ph.HE/2002.01118]. doi:10.3847/1538-4357/ab6a99.
213. M. J. Boschini et al. Inference of the Local Interstellar Spectra of Cosmic-Ray Nuclei $Z \leq 28$ with the GALPROP-HELMOD Framework. ApJS **2020**, *250*, 27, [arXiv:astro-ph.HE/2006.01337]. doi:10.3847/1538-4365/aba901.
214. C. Corti, V. Bindi, C. Consolandi, and K. Whitman. Solar Modulation of the Local Interstellar Spectrum with Voyager 1, AMS-02, PAMELA, and BESS. ApJ **2016**, *829*, 8. doi:10.3847/0004-637X/829/1/8.
215. Andrew W. Strong, Igor V. Moskalenko, and Olaf Reimer. Diffuse Continuum Gamma Rays from the Galaxy. ApJ **2000**, *537*, 763–784, [arXiv:astro-ph/astro-ph/9811296]. doi:10.1086/309038.
216. O. Adriani et al. PAMELA Measurements of Cosmic-Ray Proton and Helium Spectra. *Science* **2011**, *332*, 69, [arXiv:astro-ph.HE/1103.4055]. doi:10.1126/science.1199172.
217. J. J. Engelmann et al. Charge composition and energy spectra of cosmic-ray nuclei for elements from Be to Ni - Results from HEAO-3-C2. A&A **1990**, *233*, 96–111.
218. S. Ostapchenko. Monte Carlo treatment of hadronic interactions in enhanced Pomeron scheme: QGSJET-II model. Phys. Rev. D **2011**, *83*, 014018, [arXiv:hep-ph/1010.1869]. doi:10.1103/PhysRevD.83.014018.
219. A. W. Strong. Local interstellar cosmic-ray spectra derived from gamma-ray emissivities. *arXiv e-prints* **2015**, p. arXiv:1507.05006, [arXiv:astro-ph.HE/1507.05006].
220. Erik Bertram, Simon C. O. Glover, Paul C. Clark, Sarah E. Ragan, and Ralf S. Klessen. Synthetic observations of molecular clouds in a galactic centre environment - I. Studying maps of column density and integrated intensity. MNRAS **2016**, *455*, 3763–3778, [arXiv:astro-ph.GA/1509.06649]. doi:10.1093/mnras/stv2619.
221. J. Skilling and A. W. Strong. Cosmic ray exclusion from dense molecular clouds. A&A **1976**, *53*, 253–258.
222. John E. Everett and Ellen G. Zweibel. The Interaction of Cosmic Rays with Diffuse Clouds. ApJ **2011**, *739*, 60, [arXiv:astro-ph.GA/1107.1243]. doi:10.1088/0004-637X/739/2/60.
223. M. Padovani and D. Galli. Effects of magnetic fields on the cosmic-ray ionization of molecular cloud cores. A&A **2011**, *530*, A109, [arXiv:astro-ph.SR/1104.5445]. doi:10.1051/0004-6361/201116853.
224. Rui-zhi Yang, Emma de Oña Wilhelmi, and Felix Aharonian. Probing cosmic rays in nearby giant molecular clouds with the Fermi Large Area Telescope. A&A **2014**, *566*, A142, [arXiv:astro-ph.HE/1303.7323]. doi:10.1051/0004-6361/201321044.
225. Andrii Neronov, Denys Malyshev, and Dmitri V. Semikoz. Cosmic-ray spectrum in the local Galaxy. A&A **2017**, *606*, A22, [arXiv:astro-ph.HE/1705.02200]. doi:10.1051/0004-6361/201731149.
226. Vardan Baghmanyan, Giada Peron, Sabrina Casanova, Felix Aharonian, and Roberta Zanin. Evidence of Cosmic-Ray Excess from Local Giant Molecular Clouds. ApJ **2020**, *901*, L4, [arXiv:astro-ph.HE/2009.08893]. doi:10.3847/2041-8213/abb5f8.
227. A. Albert et al. Probing the Sea of Cosmic Rays by Measuring Gamma-Ray Emission from Passive Giant Molecular Clouds. *arXiv e-prints* **2021**, p. arXiv:2101.08748, [arXiv:astro-ph.HE/2101.08748].
228. A. W. Strong et al. The radial distribution of galactic gamma rays. IV. The whole galaxy. A&A **1988**, *207*, 1–15.
229. G. F. Bignami, C. E. Fichtel, D. A. Kniffen, and D. J. Thompson. High-energy galactic gamma radiation from cosmic rays concentrated in spiral arms. ApJ **1975**, *199*, 54–60. doi:10.1086/153663.
230. F. Acero et al. Development of the Model of Galactic Interstellar Emission for Standard Point-source Analysis of Fermi Large Area Telescope Data. ApJS **2016**, *223*, 26, [arXiv:astro-ph.HE/1602.07246]. doi:10.3847/0067-0049/223/2/26.
231. Ruizhi Yang, Felix Aharonian, and Carmelo Evoli. Radial distribution of the diffuse γ-ray emissivity in the Galactic disk. *Phys. Rev. D* **2016**, *93*, 123007, [arXiv:astro-ph.HE/1602.04710]. doi:10.1103/PhysRevD.93.123007.
232. Mart Pothast, Daniele Gaggero, Emma Storm, and Christoph Weniger. On the progressive hardening of the cosmic-ray proton spectrum in the inner Galaxy. *JCAP* **2018**, *10*, 045, [arXiv:astro-ph.HE/1807.04554]. doi:10.1088/1475-7516/2018/10/045.
233. F. Acero et al. Fermi Large Area Telescope Third Source Catalog. ApJS **2015**, *218*, 23, [arXiv:astro-ph.HE/1501.02003]. doi:10.1088/0067-0049/218/2/23.
234. Felix Aharonian, Giada Peron, Ruizhi Yang, Sabrina Casanova, and Roberta Zanin. Probing the sea of galactic cosmic rays with Fermi-LAT. Phys. Rev. D **2020**, *101*, 083018. doi:10.1103/PhysRevD.101.083018.

235. Giada Peron, Felix Aharonian, Sabrina Casanova, Ruizhi Yang, and Roberta Zanin. Probing the Cosmic-Ray Density in the Inner Galaxy. ApJ **2021**, *907*, L11. doi:10.3847/2041-8213/abcaa9.
236. M. R. Issa, I. MacLaren, and A. W. Wolfendale. Dust-to-gas ratio and metallicity variations in nearby galaxies. A&A **1990**, *236*, 237.
237. S. Boissier, A. Boselli, V. Buat, J. Donas, and B. Milliard. The radial extinction profiles of late-type galaxies. A&A **2004**, *424*, 465–476, [arXiv:astro-ph/astro-ph/0407617]. doi:10.1051/0004-6361:20047024.
238. J. C. Muñoz-Mateos et al. Radial Distribution of Stars, Gas, and Dust in Sings Galaxies. II. Derived Dust Properties. ApJ **2009**, *701*, 1965–1991, [arXiv:astro-ph.CO/0909.2658]. doi:10.1088/0004-637X/701/2/1965.
239. M. Ackermann et al. Fermi-LAT Observations of the Diffuse γ-Ray Emission: Implications for Cosmic Rays and the Interstellar Medium. ApJ **2012**, *750*, 3, [arXiv:astro-ph.HE/1202.4039]. doi:10.1088/0004-637X/750/1/3.
240. Troy A. Porter, Igor V. Moskalenko, Andrew W. Strong, Elena Orlando, and L. Bouchet. Inverse Compton Origin of the Hard X-Ray and Soft Gamma-Ray Emission from the Galactic Ridge. ApJ **2008**, *682*, 400–407, [arXiv:astro-ph/0804.1774]. doi:10.1086/589615.
241. Laurent Bouchet et al. Diffuse Emission Measurement with the SPectrometer on INTEGRAL as an Indirect Probe of Cosmic-Ray Electrons and Positrons. ApJ **2011**, *739*, 29, [arXiv:astro-ph.HE/1107.0200]. doi:10.1088/0004-637X/739/1/29.
242. Eugene Churazov et al. INTEGRAL results on the electron-positron annihilation radiation and X-ray & Gamma-ray diffuse emission of the Milky Way. New A Rev. **2020**, *90*, 101548. doi:10.1016/j.newar.2020.101548.
243. A. A. Abdo et al. A Measurement of the Spatial Distribution of Diffuse TeV Gamma-Ray Emission from the Galactic Plane with Milagro. ApJ **2008**, *688*, 1078–1083, [arXiv:astro-ph/0805.0417]. doi:10.1086/592213.
244. B. Bartoli et al. Study of the Diffuse Gamma-Ray Emission from the Galactic Plane with ARGO-YBJ. ApJ **2015**, *806*, 20, [arXiv:astro-ph.IM/1507.06758]. doi:10.1088/0004-637X/806/1/20.
245. H. Zhou, C. Dong Rho, G. Vianello, and HAWC Collaboration. Probing Galactic Diffuse TeV Gamma-Ray Emission with the HAWC Observatory. 35th International Cosmic Ray Conference (ICRC2017), 2017, Vol. 301, *International Cosmic Ray Conference*, p. 689, [arXiv:astro-ph.HE/1709.03619].
246. M. Amenomori et al. First Detection of sub-PeV Diffuse Gamma Rays from the Galactic Disk: Evidence for Ubiquitous Galactic Cosmic Rays beyond PeV Energies. Phys. Rev. Lett. **2021**, *126*, 141101, [arXiv:astro-ph.HE/2104.05181]. doi:10.1103/PhysRevLett.126.141101.
247. A. Neronov and D. Semikoz. Galactic diffuse gamma-ray emission at TeV energy. A&A **2020**, *633*, A94, [arXiv:astro-ph.HE/1907.06061]. doi:10.1051/0004-6361/201936368.
248. G. Pagliaroli, M. Cataldo, V. Vecchiotti, and F. L. Villante. Can the TeV gamma-ray sky probe the galactic cosmic ray distribution? Journal of Physics Conference Series, 2020, Vol. 1468, *Journal of Physics Conference Series*, p. 012099. doi:10.1088/1742-6596/1468/1/012099.
249. A. Neronov and D. Semikoz. Mapping large-scale diffuse γ-ray emission in the 10-100 TeV band with Cherenkov telescopes. A&A **2020**, *637*, A44, [arXiv:astro-ph.HE/2001.00922]. doi:10.1051/0004-6361/202037608.
250. Andrii Neronov and Dmitri Semikoz. LHAASO telescope sensitivity to diffuse gamma-ray signals from the Galaxy. Phys. Rev. D **2020**, *102*, 043025, [arXiv:astro-ph.HE/2001.11881]. doi:10.1103/PhysRevD.102.043025.
251. Daniele Gaggero, Alfredo Urbano, Mauro Valli, and Piero Ullio. Gamma-ray sky points to radial gradients in cosmic-ray transport. *Phys. Rev. D* **2015**, *91*, 083012, [arXiv:astro-ph.HE/1411.7623]. doi:10.1103/PhysRevD.91.083012.
252. S. Recchia, P. Blasi, and G. Morlino. On the radial distribution of Galactic cosmic rays. *Mon. Not. Roy. Astron. Soc.* **2016**, *462*, L88–L92, [arXiv:astro-ph.HE/1604.07682]. doi:10.1093/mnrasl/slw136.
253. Katia M. Ferrière. The interstellar environment of our galaxy. *Reviews of Modern Physics* **2001**, *73*, 1031–1066, [arXiv:astro-ph/astro-ph/0106359]. doi:10.1103/RevModPhys.73.1031.
254. Daniel De Marco, Pasquale Blasi, and Todor Stanev. Numerical propagation of high energy cosmic rays in the Galaxy. I. Technical issues. *JCAP* **2007**, *06*, 027, [arXiv:astro-ph/0705.1972]. doi:10.1088/1475-7516/2007/06/027.
255. A. P. Snodin, A. Shukurov, G. R. Sarson, P. J. Bushby, and L. F. S. Rodrigues. Global diffusion of cosmic rays in random magnetic fields. *Mon. Not. Roy. Astron. Soc.* **2016**, *457*, 3975–3987, [arXiv:astro-ph.HE/1509.03766]. doi:10.1093/mnras/stw217.
256. Andrej Dundovic, Oreste Pezzi, Pasquale Blasi, Carmelo Evoli, and William H. Matthaeus. Novel aspects of cosmic ray diffusion in synthetic magnetic turbulence. *Phys. Rev. D* **2020**, *102*, 103016, [arXiv:astro-ph.HE/2007.09142]. doi:10.1103/PhysRevD.102.103016.
257. D. Gaggero, D. Grasso, A. Marinelli, M. Taoso, and A. Urbano. Diffuse cosmic rays shining in the Galactic center: A novel interpretation of H.E.S.S. and Fermi-LAT gamma-ray data. *Phys. Rev. Lett.* **2017**, *119*, 031101, [arXiv:astro-ph.HE/1702.01124]. doi:10.1103/PhysRevLett.119.031101.
258. Paolo Lipari and Silvia Vernetto. Diffuse Galactic gamma ray flux at very high energy. *Phys. Rev. D* **2018**, *98*, 043003, [arXiv:astro-ph.HE/1804.10116]. doi:10.1103/PhysRevD.98.043003.
259. Daniele Gaggero, Dario Grasso, Antonio Marinelli, Alfredo Urbano, and Mauro Valli. The gamma-ray and neutrino sky: A consistent picture of Fermi-LAT, Milagro, and IceCube results. *Astrophys. J. Lett.* **2015**, *815*, L25, [arXiv:astro-ph.HE/1504.00227]. doi:10.1088/2041-8205/815/2/L25.

260. G. Pagliaroli and F. L. Villante. A multi-messenger study of the total galactic high-energy neutrino emission. *JCAP* **2018**, *08*, 035, [arXiv:hep-ph/1710.01040]. doi:10.1088/1475-7516/2018/08/035.
261. A. Albert et al. Joint Constraints on Galactic Diffuse Neutrino Emission from the ANTARES and IceCube Neutrino Telescopes. *Astrophys. J. Lett.* **2018**, *868*, L20, [arXiv:astro-ph.HE/1808.03531]. doi:10.3847/2041-8213/aaeecf.
262. M. G. Aartsen et al. Search for Sources of Astrophysical Neutrinos Using Seven Years of IceCube Cascade Events. *Astrophys. J.* **2019**, *886*, 12, [arXiv:astro-ph.HE/1907.06714]. doi:10.3847/1538-4357/ab4ae2.
263. M. Ackermann et al. Deep view of the Large Magellanic Cloud with six years of Fermi-LAT observations. A&A **2016**, *586*, A71, [arXiv:astro-ph.HE/1509.06903]. doi:10.1051/0004-6361/201526920.
264. Victoria Scowcroft et al. The Carnegie Hubble Program: The Distance and Structure of the SMC as Revealed by Mid-infrared Observations of Cepheids. ApJ **2016**, *816*, 49, [arXiv:astro-ph.GA/1502.06995]. doi:10.3847/0004-637X/816/2/49.
265. A. A. Abdo et al. Detection of the Small Magellanic Cloud in gamma-rays with Fermi/LAT. A&A **2010**, *523*, A46, [arXiv:astro-ph.HE/1008.2127]. doi:10.1051/0004-6361/201014855.
266. R. Caputo et al. Search for gamma-ray emission from dark matter annihilation in the Small Magellanic Cloud with the Fermi Large Area Telescope. Phys. Rev. D **2016**, *93*, 062004, [arXiv:astro-ph.HE/1603.00965]. doi:10.1103/PhysRevD.93.062004.
267. A. A. Abdo et al. Observations of the Large Magellanic Cloud with Fermi. A&A **2010**, *512*, A7+. doi:10.1051/0004-6361/200913474.
268. M. Ackermann et al. Observations of M31 and M33 with the Fermi Large Area Telescope: A Galactic Center Excess in Andromeda? ApJ **2017**, *836*, 208, [arXiv:astro-ph.HE/1702.08602]. doi:10.3847/1538-4357/aa5c3d.
269. N. Weinrich et al. Galactic halo size in the light of recent AMS-02 data. A&A **2020**, *639*, A74, [arXiv:astro-ph.HE/2004.00441]. doi:10.1051/0004-6361/202038064.
270. M. Dahlem, R. J. Dettmar, and E. Hummel. Spatially correlated diffuse Hα and radio continuum emission from the halo of NGC 891. A&A **1994**, *290*, 384–392.
271. L. Tibaldo et al. Fermi-LAT Observations of High- and Intermediate-velocity Clouds: Tracing Cosmic Rays in the Halo of the Milky Way. ApJ **2015**, *807*, 161, [arXiv:astro-ph.HE/1505.04223]. doi:10.1088/0004-637X/807/2/161.
272. B. P. Wakker. Distances and Metallicities of High- and Intermediate-Velocity Clouds. ApJS **2001**, *136*, 463–535, [arXiv:astro-ph/astro-ph/0102147]. doi:10.1086/321783.
273. T. Joubaud, I. A. Grenier, J. Ballet, and J. D. Soler. Gas shells and magnetic fields in the Orion-Eridanus superbubble. A&A **2019**, *631*, A52, [arXiv:astro-ph.HE/1909.10083]. doi:10.1051/0004-6361/201936239.
274. Gaia Collaboration et al. Gaia Data Release 2. Summary of the contents and survey properties. A&A **2018**, *616*, A1, [arXiv:astro-ph.GA/1804.09365]. doi:10.1051/0004-6361/201833051.
275. M. S. Pshirkov, V. V. Vasiliev, and K. A. Postnov. Evidence of Fermi bubbles around M31. MNRAS **2016**, *459*, L76–L80, [arXiv:astro-ph.HE/1603.07245]. doi:10.1093/mnrasl/slw045.
276. Christopher M. Karwin, Simona Murgia, Sheldon Campbell, and Igor V. Moskalenko. Fermi-LAT Observations of γ-Ray Emission toward the Outer Halo of M31. ApJ **2019**, *880*, 95, [arXiv:astro-ph.HE/1812.02958]. doi:10.3847/1538-4357/ab2880.
277. S. Recchia, S. Gabici, F. A. Aharonian, and V. Niro. Giant cosmic ray halos around M31 and the Milky Way. *arXiv e-prints* **2021**, p. arXiv:2101.05016, [arXiv:astro-ph.HE/2101.05016].
278. S. Recchia, P. Blasi, and G. Morlino. Cosmic ray driven Galactic winds. MNRAS **2016**, *462*, 4227–4239, [arXiv:astro-ph.HE/1603.06746]. doi:10.1093/mnras/stw1966.
279. F. Holguin, M. Ruszkowski, A. Lazarian, R. Farber, and H. Y. K. Yang. Role of cosmic-ray streaming and turbulent damping in driving galactic winds. MNRAS **2019**, *490*, 1271–1282, [arXiv:astro-ph.HE/1807.05494]. doi:10.1093/mnras/stz2568.
280. V. A. Dogiel, A. V. Ivlev, D. O. Chernyshov, and C. M. Ko. Formation of the Cosmic-Ray Halo: Galactic Spectrum of Primary Cosmic Rays. ApJ **2020**, *903*, 135, [arXiv:astro-ph.HE/2009.08799]. doi:10.3847/1538-4357/abba31.
281. Robert Feldmann, Dan Hooper, and Nickolay Y. Gnedin. Circum-galactic Gas and the Isotropic Gamma-Ray Background. ApJ **2013**, *763*, 21, [arXiv:astro-ph.HE/1205.0249]. doi:10.1088/0004-637X/763/1/21.
282. Pasquale Blasi and Elena Amato. Escape of Cosmic Rays from the Galaxy and Effects on the Circumgalactic Medium. Phys. Rev. Lett. **2019**, *122*, 051101, [arXiv:astro-ph.HE/1901.03609]. doi:10.1103/PhysRevLett.122.051101.
283. M. Ackermann et al. GeV Observations of Star-forming Galaxies with the Fermi Large Area Telescope. ApJ **2012**, *755*, 164, [arXiv:astro-ph.HE/1206.1346]. doi:10.1088/0004-637X/755/2/164.
284. Gregory Dobler, Douglas P. Finkbeiner, Ilias Cholis, Tracy R. Slatyer, and Neal Weiner. The Fermi Haze: A Gamma-Ray Counterpart to the Microwave Haze. *Astrophys. J.* **2010**, *717*, 825–842, [arXiv:astro-ph.HE/0910.4583]. doi:10.1088/0004-637X/717/2/825.
285. Meng Su, Tracy R. Slatyer, and Douglas P. Finkbeiner. Giant Gamma-ray Bubbles from Fermi-LAT: AGN Activity or Bipolar Galactic Wind? *Astrophys. J.* **2010**, *724*, 1044–1082, [arXiv:astro-ph.HE/1005.5480]. doi:10.1088/0004-637X/724/2/1044.
286. M. Ackermann et al. The Spectrum and Morphology of the *Fermi* Bubbles. *Astrophys. J.* **2014**, *793*, 64, [arXiv:astro-ph.HE/1407.7905]. doi:10.1088/0004-637X/793/1/64.
287. H. Y. Yang, Mateusz Ruszkowski, and Ellen Zweibel. Unveiling the Origin of the Fermi Bubbles. *Galaxies* **2018**, *6*, 29, [arXiv:astro-ph.HE/1802.03890]. doi:10.3390/galaxies6010029.

288. Simona Murgia. The Fermi?LAT Galactic Center Excess: Evidence of Annihilating Dark Matter? *Annual Review of Nuclear and Particle Science* **2020**, *70*, 455–483. doi:10.1146/annurev-nucl-101916-123029.
289. T. A. Porter, G. Jóhannesson, and I. V. Moskalenko. Deciphering Residual Emissions: Time-dependent Models for the Nonthermal Interstellar Radiation from the Milky Way. ApJ **2019**, *887*, 250, [arXiv:astro-ph.HE/1909.02223]. doi:10.3847/1538-4357/ab5961.
290. Marta D'Angelo, Pasquale Blasi, and Elena Amato. Grammage of cosmic rays around Galactic supernova remnants. Phys. Rev. D **2016**, *94*, 083003, [arXiv:astro-ph.HE/1512.05000]. doi:10.1103/PhysRevD.94.083003.
291. Marta D'Angelo, Giovanni Morlino, Elena Amato, and Pasquale Blasi. Diffuse gamma-ray emission from self-confined cosmic rays around Galactic sources. MNRAS **2018**, *474*, 1944–1954, [arXiv:astro-ph.HE/1710.10937]. doi:10.1093/mnras/stx2828.
292. Guđlaugur Jóhannesson, Troy A. Porter, and Igor V. Moskalenko. Cosmic-Ray Propagation in Light of the Recent Observation of Geminga. ApJ **2019**, *879*, 91, [arXiv:astro-ph.HE/1903.05509]. doi:10.3847/1538-4357/ab258e.
293. A. U. Abeysekara et al. Extended gamma-ray sources around pulsars constrain the origin of the positron flux at Earth. *Science* **2017**, *358*, 911–914, [arXiv:astro-ph.HE/1711.06223]. doi:10.1126/science.aan4880.
294. Stefano Profumo, Javier Reynoso-Cordova, Nicholas Kaaz, and Maya Silverman. Lessons from HAWC pulsar wind nebulae observations: The diffusion constant is not a constant; pulsars remain the likeliest sources of the anomalous positron fraction; cosmic rays are trapped for long periods of time in pockets of inefficient diffusion. Phys. Rev. D **2018**, *97*, 123008, [arXiv:astro-ph.HE/1803.09731]. doi:10.1103/PhysRevD.97.123008.
295. Silvia Manconi, Mattia Di Mauro, and Fiorenza Donato. Contribution of pulsars to cosmic-ray positrons in light of recent observation of inverse-Compton halos. Phys. Rev. D **2020**, *102*, 023015, [arXiv:astro-ph.HE/2001.09985]. doi:10.1103/PhysRevD.102.023015.
296. L. Nava, S. Gabici, A. Marcowith, G. Morlino, and V. S. Ptuskin. Non-linear diffusion of cosmic rays escaping from supernova remnants - I. The effect of neutrals. MNRAS **2016**, *461*, 3552–3562, [arXiv:astro-ph.HE/1606.06902]. doi:10.1093/mnras/stw1592.
297. L. Nava et al. Non-linear diffusion of cosmic rays escaping from supernova remnants - II. Hot ionized media. MNRAS **2019**, *484*, 2684–2691, [arXiv:astro-ph.HE/1903.03193]. doi:10.1093/mnras/stz137.
298. Benedikt Schroer, Oreste Pezzi, Damiano Caprioli, Colby Haggerty, and Pasquale Blasi. Dynamical effects of cosmic rays leaving their sources. *arXiv e-prints* **2020**, p. arXiv:2011.02238, [arXiv:astro-ph.HE/2011.02238].
299. V. S. Ptuskin and V. N. Zirakashvili. On the spectrum of high-energy cosmic rays produced by supernova remnants in the presence of strong cosmic-ray streaming instability and wave dissipation. A&A **2005**, *429*, 755–765, [arXiv:astro-ph/astro-ph/0408025]. doi:10.1051/0004-6361:20041517.
300. L. O'c. Drury. Escaping the accelerator: how, when and in what numbers do cosmic rays get out of supernova remnants? MNRAS **2011**, *415*, 1807–1814, [arXiv:astro-ph.GA/1009.4799]. doi:10.1111/j.1365-2966.2011.18824.x.
301. A. R. Bell, K. M. Schure, B. Reville, and G. Giacinti. Cosmic-ray acceleration and escape from supernova remnants. MNRAS **2013**, *431*, 415–429, [arXiv:astro-ph.HE/1301.7264]. doi:10.1093/mnras/stt179.
302. A. M. W. Mitchell, G. P. Rowell, S. Celli, and S. Einecke. Using interstellar clouds to search for Galactic PeVatrons: gamma-ray signatures from supernova remnants. MNRAS **2021**, *503*, 3522–3539, [arXiv:astro-ph.HE/2103.01787]. doi:10.1093/mnras/stab667.
303. Pierre Cristofari, Pasquale Blasi, and Elena Amato. The low rate of Galactic pevatrons. *Astroparticle Physics* **2020**, *123*, 102492, [arXiv:astro-ph.HE/2007.04294]. doi:10.1016/j.astropartphys.2020.102492.
304. M. A. Malkov, P. H. Diamond, R. Z. Sagdeev, F. A. Aharonian, and I. V. Moskalenko. Analytic Solution for Self-regulated Collective Escape of Cosmic Rays from Their Acceleration Sites. ApJ **2013**, *768*, 73, [arXiv:astro-ph.HE/1207.4728]. doi:10.1088/0004-637X/768/1/73.
305. L. Brahimi, A. Marcowith, and V. S. Ptuskin. Nonlinear diffusion of cosmic rays escaping from supernova remnants: Cold partially neutral atomic and molecular phases. A&A **2020**, *633*, A72, [arXiv:astro-ph.HE/1909.04530]. doi:10.1051/0004-6361/201936166.
306. G. Giacinti, M. Kachelrieß, and D. V. Semikoz. Anisotropic cosmic ray diffusion and its implications for gamma-ray astronomy. Phys. Rev. D **2013**, *88*, 023010, [arXiv:astro-ph.HE/1306.3209]. doi:10.1103/PhysRevD.88.023010.
307. H. E. S. S. Collaboration et al. H.E.S.S. observations of RX J1713.7-3946 with improved angular and spectral resolution: Evidence for gamma-ray emission extending beyond the X-ray emitting shell. A&A **2018**, *612*, A6, [arXiv:astro-ph.HE/1609.08671]. doi:10.1051/0004-6361/201629790.
308. Yudong Cui, Paul K. H. Yeung, P. H. Thomas Tam, and Gerd Pühlhofer. Leaked GeV CRs from a Broken Shell: Explaining 9 Years of Fermi-LAT Data of SNR W28. ApJ **2018**, *860*, 69, [arXiv:astro-ph.HE/1805.03372]. doi:10.3847/1538-4357/aac37b.
309. Yasunobu Uchiyama, Roger D. Blandford, Stefan Funk, Hiroyasu Tajima, and Takaaki Tanaka. Gamma-ray Emission from Crushed Clouds in Supernova Remnants. ApJ **2010**, *723*, L122–L126, [arXiv:astro-ph.HE/1008.1840]. doi:10.1088/2041-8205/723/1/L122.
310. S. Abdollahi et al. Fermi Large Area Telescope Fourth Source Catalog. ApJS **2020**, *247*, 33, [arXiv:astro-ph.HE/1902.10045]. doi:10.3847/1538-4365/ab6bcb.
311. S. Gabici, F. A. Aharonian, and P. Blasi. Gamma rays from molecular clouds. Ap&SS **2007**, *309*, 365–371, [arXiv:astro-ph/0610032]. doi:10.1007/s10509-007-9427-6.
312. Y. Hanabata et al. Detailed Investigation of the Gamma-Ray Emission in the Vicinity of SNR W28 with FERMI-LAT. ApJ **2014**, *786*, 145, [arXiv:astro-ph.HE/1403.6878]. doi:10.1088/0004-637X/786/2/145.

313. S. Celli, G. Morlino, S. Gabici, and F. A. Aharonian. Exploring particle escape in supernova remnants through gamma rays. MNRAS **2019**, *490*, 4317–4333, [arXiv:astro-ph.HE/1906.09454]. doi:10.1093/mnras/stz2897.
314. MAGIC Collaboration et al. Study of the GeV to TeV morphology of the γ-Cygni SNR (G78.2+2.1) with MAGIC and Fermi-LAT. *arXiv e-prints* **2020**, p. arXiv:2010.15854, [arXiv:astro-ph.HE/2010.15854].
315. F. Acero et al. The First Fermi LAT Supernova Remnant Catalog. ApJS **2016**, *224*, 8, [arXiv:astro-ph.HE/1511.06778]. doi:10.3847/0067-0049/224/1/8.
316. Yasunobu Uchiyama et al. Fermi Large Area Telescope Discovery of GeV Gamma-Ray Emission from the Vicinity of SNR W44. ApJ **2012**, *749*, L35, [arXiv:astro-ph.HE/1203.3234]. doi:10.1088/2041-8205/749/2/L35.
317. H. Abdalla et al. The H.E.S.S. Galactic plane survey. A&A **2018**, *612*, A1, [arXiv:astro-ph.HE/1804.02432]. doi:10.1051/0004-6361/201732098.
318. J. Eagle, S. Marchesi, D. Castro, M. Ajello, and A. Vendrasco. Gamma-Ray Emission Revealed at the Western Edge of SNR G344.7-0.1. ApJ **2020**, *904*, 123, [arXiv:astro-ph.HE/2006.08757]. doi:10.3847/1538-4357/abbe08.
319. J. Eagle et al. 2FHL J0826.1-4500: Discovery of a Possible Shock-Cloud Interaction on the Western Edge of the Vela Supernova Remnant. ApJ **2019**, *870*, 35, [arXiv:astro-ph.HE/1811.05898]. doi:10.3847/1538-4357/aaf0ff.
320. Xiaping Tang. Gamma-ray emission from middle-aged supernova remnants interacting with molecular clouds: the challenge for current models. MNRAS **2019**, *482*, 3843–3856, [arXiv:astro-ph.HE/1707.00958]. doi:10.1093/mnras/sty1042.
321. M. Ackermann et al. Search for Extended Sources in the Galactic Plane Using Six Years of Fermi-Large Area Telescope Pass 8 Data above 10 GeV. ApJ **2017**, *843*, 139, [arXiv:astro-ph.HE/1702.00476]. doi:10.3847/1538-4357/aa775a.
322. G. Pivato et al. Fermi LAT and WMAP Observations of the Supernova Remnant HB 21. ApJ **2013**, *779*, 179, [arXiv:astro-ph.HE/1311.0393]. doi:10.1088/0004-637X/779/2/179.
323. L. Ambrogi et al. Spectral and morphological study of the gamma radiation of the middle-aged supernova remnant HB 21. A&A **2019**, *623*, A86, [arXiv:astro-ph.HE/1902.06064]. doi:10.1051/0004-6361/201833985.
324. Bryan M. Gaensler and Patrick O. Slane. The Evolution and Structure of Pulsar Wind Nebulae. ARA&A **2006**, *44*, 17–47, [arXiv:astro-ph/astro-ph/0601081]. doi:10.1146/annurev.astro.44.051905.092528.
325. Niccolò Bucciantini, Barbara Olmi, and Luca Del Zanna. 3D Relativistic MHD Simulations of Pulsar Bow Shock Nebulae. Journal of Physics Conference Series, 2020, Vol. 1623, *Journal of Physics Conference Series*, p. 012002, [arXiv:astro-ph.HE/2005.14079]. doi:10.1088/1742-6596/1623/1/012002.
326. G. Giacinti et al. On the TeV Halo Fraction in gamma-ray bright Pulsar Wind Nebulae. *arXiv e-prints* **2019**, p. arXiv:1907.12121, [arXiv:astro-ph.HE/1907.12121].
327. Tim Linden et al. Using HAWC to discover invisible pulsars. Phys. Rev. D **2017**, *96*, 103016, [arXiv:astro-ph.HE/1703.09704]. doi:10.1103/PhysRevD.96.103016.
328. Mattia Di Mauro, Silvia Manconi, and Fiorenza Donato. Evidences of low-diffusion bubbles around Galactic pulsars. *arXiv e-prints* **2019**, p. arXiv:1908.03216, [arXiv:astro-ph.HE/1908.03216].
329. Takahiro Sudoh, Tim Linden, and John F. Beacom. TeV halos are everywhere: Prospects for new discoveries. Phys. Rev. D **2019**, *100*, 043016, [arXiv:astro-ph.HE/1902.08203]. doi:10.1103/PhysRevD.100.043016.
330. Dan Hooper and Tim Linden. Millisecond pulsars, TeV halos, and implications for the Galactic Center gamma-ray excess. Phys. Rev. D **2018**, *98*, 043005, [arXiv:astro-ph.HE/1803.08046]. doi:10.1103/PhysRevD.98.043005.
331. Tim Linden and Benjamin J. Buckman. Pulsar TeV Halos Explain the Diffuse TeV Excess Observed by Milagro. Phys. Rev. Lett. **2018**, *120*, 121101, [arXiv:astro-ph.HE/1707.01905]. doi:10.1103/PhysRevLett.120.121101.
332. Mattia Di Mauro, Silvia Manconi, and Fiorenza Donato. Detection of a γ -ray halo around Geminga with the Fermi-LAT data and implications for the positron flux. Phys. Rev. D **2019**, *100*, 123015, [arXiv:astro-ph.HE/1903.05647]. doi:10.1103/PhysRevD.100.123015.
333. G. Giacinti et al. Halo fraction in TeV-bright pulsar wind nebulae. A&A **2020**, *636*, A113, [arXiv:astro-ph.HE/1907.12121]. doi:10.1051/0004-6361/201936505.
334. Carmelo Evoli, Tim Linden, and Giovanni Morlino. Self-generated cosmic-ray confinement in TeV halos: Implications for TeV γ -ray emission and the positron excess. Phys. Rev. D **2018**, *98*, 063017, [arXiv:astro-ph.HE/1807.09263]. doi:10.1103/PhysRevD.98.063017.
335. Rubén López-Coto and Gwenael Giacinti. Constraining the properties of the magnetic turbulence in the Geminga region using HAWC γ-ray data. MNRAS **2018**, *479*, 4526–4534, [arXiv:astro-ph.HE/1712.04373]. doi:10.1093/mnras/sty1821.
336. G. Ferrand and A. Marcowith. On the shape of the spectrum of cosmic rays accelerated inside superbubbles. A&A **2010**, *510*, A101, [arXiv:astro-ph.HE/0911.4457]. doi:10.1051/0004-6361/200913520.
337. T. Montmerle. On gamma-ray sources, supernova remnants, OB associations, and the origin of cosmic rays. ApJ **1979**, *231*, 95–110. doi:10.1086/157166.
338. A. M. Bykov. Particle Acceleration and Nonthermal Phenomena in Superbubbles. *Space Science Reviews* **2001**, *99*, 317–326.
339. E. Parizot, A. Marcowith, E. van der Swaluw, A. M. Bykov, and V. Tatischeff. Superbubbles and energetic particles in the Galaxy. I. Collective effects of particle acceleration. A&A **2004**, *424*, 747–760, [arXiv:astro-ph/0405531]. doi:10.1051/0004-6361:20041269.

340. Jeongbhin Seo, Hyesung Kang, and Dongsu Ryu. The Contribution of Stellar Winds to Cosmic Ray Production. *Journal of Korean Astronomical Society* **2018**, *51*, 37–48, [arXiv:astro-ph.HE/1804.07486]. doi:10.5303/JKAS.2018.51.2.37.
341. G. Morlino, P. Blasi, E. Peretti, and P. Cristofari. Particle acceleration in winds of star clusters. MNRAS **2021**, [arXiv:astro-ph.HE/2102.09217]. doi:10.1093/mnras/stab690.
342. T. Tolksdorf, I. A. Grenier, T. Joubaud, and R. Schlickeiser. Cosmic Rays in Superbubbles. ApJ **2019**, *879*, 66. doi:10.3847/1538-4357/ab24c6.
343. Andrei M. Bykov et al. High-Energy Particles and Radiation in Star-Forming Regions. Space Sci. Rev. **2020**, *216*, 42, [arXiv:astro-ph.HE/2003.11534]. doi:10.1007/s11214-020-00663-0.
344. A. M. Bykov, D. C. Ellison, P. E. Gladilin, and S. M. Osipov. Supernovae in compact star clusters as sources of high-energy cosmic rays and neutrinos. *Advances in Space Research* **2018**, *62*, 2764–2772, [arXiv:astro-ph.HE/1706.01135]. doi:10.1016/j.asr.2017.05.043.
345. W. R. Binns et al. Observation of the ^{60}Fe nucleosynthesis-clock isotope in galactic cosmic rays. *Science* **2016**, *352*, 677–680. doi:10.1126/science.aad6004.
346. M. H. Israel et al. Elemental Composition at the Cosmic-Ray Source Derived from the ACE-CRIS Instrument. I. $_6$C to $_{28}$Ni. ApJ **2018**, *865*, 69. doi:10.3847/1538-4357/aad867.
347. Nicholas W. Melena, Philip Massey, Nidia I. Morrell, and Amanda M. Zangari. The Massive Star Content of NGC 3603. AJ **2008**, *135*, 878–891, [arXiv:astro-ph/0712.2621]. doi:10.1088/0004-6256/135/3/878.
348. N. Furukawa et al. Molecular Clouds Toward RCW49 and Westerlund 2: Evidence for Cluster Formation Triggered by Cloud-Cloud Collision. ApJ **2009**, *696*, L115–L119, [arXiv:astro-ph.GA/0904.0286]. doi:10.1088/0004-637X/696/2/L115.
349. Carlos A. Vargas Álvarez et al. The Distance to the Massive Galactic Cluster Westerlund 2 from a Spectroscopic and HST Photometric Study. AJ **2013**, *145*, 125, [arXiv:astro-ph.SR/1302.0863]. doi:10.1088/0004-6256/145/5/125.
350. Q. Nguyen Luong et al. W43: the closest molecular complex of the Galactic bar? A&A **2011**, *529*, A41, [arXiv:astro-ph.SR/1102.3460]. doi:10.1051/0004-6361/201016271.
351. M. Ackermann et al. A Cocoon of Freshly Accelerated Cosmic Rays Detected by Fermi in the Cygnus Superbubble. *Science* **2011**, *334*, 1103–. doi:10.1126/science.1210311.
352. L. Saha et al. Morphological and Spectral Study of 4FGL J1115.1-6118 in the Region of the Young Massive Stellar Cluster NGC 3603. ApJ **2020**, *897*, 131, [arXiv:astro-ph.HE/2006.00274]. doi:10.3847/1538-4357/ab9ac2.
353. S. Ohm, J. A. Hinton, and R. White. γ-ray emission from the Westerlund 1 region. MNRAS **2013**, *434*, 2289–2294, [arXiv:astro-ph.HE/1306.5642]. doi:10.1093/mnras/stt1170.
354. Rui-zhi Yang, Emma de Oña Wilhelmi, and Felix Aharonian. Diffuse γ-ray emission in the vicinity of young star cluster Westerlund 2. A&A **2018**, *611*, A77, [arXiv:astro-ph.HE/1710.02803]. doi:10.1051/0004-6361/201732045.
355. Rui-Zhi Yang and Yuan Wang. The diffuse gamma-ray emission toward the Galactic mini starburst W43. A&A **2020**, *640*, A60, [arXiv:astro-ph.HE/2007.15295]. doi:10.1051/0004-6361/202037518.
356. B. Bartoli et al. Identification of the TeV Gamma-Ray Source ARGO J2031+4157 with the Cygnus Cocoon. ApJ **2014**, *790*, 152, [arXiv:astro-ph.HE/1406.6436]. doi:10.1088/0004-637X/790/2/152.
357. A. U. Abeysekara et al. HAWC observations of the acceleration of very-high-energy cosmic rays in the Cygnus Cocoon. *Nature Astronomy* **2021**, [arXiv:astro-ph.HE/2103.06820]. doi:10.1038/s41550-021-01318-y.
358. A. Abramowski et al. Discovery of extended VHE γ-ray emission from the vicinity of the young massive stellar cluster Westerlund 1. A&A **2012**, *537*, A114, [arXiv:astro-ph.HE/1111.2043]. doi:10.1051/0004-6361/201117928.
359. A. H. E. S. S. Collaborationand Abramowski et al. Revisiting the Westerlund 2 field with the HESS telescope array. A&A **2011**, *525*, A46, [arXiv:astro-ph.HE/1009.3012]. doi:10.1051/0004-6361/201015290.
360. HESS Collaboration et al. Acceleration of petaelectronvolt protons in the Galactic Centre. Nature **2016**, *531*, 476–479, [arXiv:astro-ph.HE/1603.07730]. doi:10.1038/nature17147.
361. A. Abramowski et al. The exceptionally powerful TeV γ-ray emitters in the Large Magellanic Cloud. *Science* **2015**, *347*, 406–412, [arXiv:astro-ph.HE/1501.06578]. doi:10.1126/science.1261313.
362. J. Ballet, T. H. Burnett, S. W. Digel, and B. Lott. Fermi Large Area Telescope Fourth Source Catalog Data Release 2. *arXiv e-prints* **2020**, p. arXiv:2005.11208, [arXiv:astro-ph.HE/2005.11208].
363. M. Ackermann et al. The Search for Spatial Extension in High-latitude Sources Detected by the Fermi Large Area Telescope. ApJS **2018**, *237*, 32, [arXiv:astro-ph.HE/1804.08035]. doi:10.3847/1538-4365/aacdf7.
364. Bing Liu, Rui-zhi Yang, Xiao-na Sun, Felix Aharonian, and Yang Chen. The GeV Emission in the Field of the Star-forming Region W30 Revisited. ApJ **2019**, *881*, 94, [arXiv:astro-ph.HE/1907.00180]. doi:10.3847/1538-4357/ab2df8.
365. Xiao-Na Sun et al. Diffuse γ-ray emission toward the massive star-forming region, W40. A&A **2020**, *639*, A80, [arXiv:astro-ph.HE/2006.00879]. doi:10.1051/0004-6361/202037580.
366. T. Ergin et al. Probing the star formation origin of gamma-rays from 3FHL J1907.0+0713. MNRAS **2021**, *501*, 4226–4237, [arXiv:astro-ph.HE/2012.07357]. doi:10.1093/mnras/staa3817.

367. J. Katsuta, Y. Uchiyama, and S. Funk. Extended Gamma-Ray Emission from the G25.0+0.0 Region: A Star-forming Region Powered by the Newly Found OB Association? ApJ **2017**, *839*, 129, [arXiv:astro-ph.HE/1704.06110]. doi:10.3847/1538-4357/aa6aa3.
368. Felix Aharonian, Ruizhi Yang, and Emma de Oña Wilhelmi. Massive stars as major factories of Galactic cosmic rays. *Nature Astronomy* **2019**, *3*, 561–567, [arXiv:astro-ph.HE/1804.02331]. doi:10.1038/s41550-019-0724-0.
369. L. Jouvin, A. Lemière, and R. Terrier. Does the SN rate explain the very high energy cosmic rays in the central 200 pc of our Galaxy? MNRAS **2017**, *467*, 4622–4630, [arXiv:astro-ph.HE/1703.10398]. doi:10.1093/mnras/stx361.
370. L. Jouvin, A. Lemière, and R. Terrier. Time-dependent escape of cosmic rays from supernova remnants potentially at the origin of the very-high-energy cosmic-ray gradient of the Galactic center. A&A **2020**, *644*, A113, [arXiv:astro-ph.HE/2011.00214]. doi:10.1051/0004-6361/202039069.
371. G. Maurin, A. Marcowith, N. Komin, F. Krayzel, and G. Lamanna. Embedded star clusters as sources of high-energy cosmic rays . Modelling and constraints. A&A **2016**, *591*, A71, [arXiv:astro-ph.HE/1605.04202]. doi:10.1051/0004-6361/201628465.
372. M. Ackermann et al. An extremely bright gamma-ray pulsar in the Large Magellanic Cloud. *Science* **2015**, *350*, 801–805. doi:10.1126/science.aac7400.
373. H. J. Völk, F. A. Aharonian, and D. Breitschwerdt. The Nonthermal Energy Content and Gamma-Ray Emission of Starburst Galaxies and Clusters of Galaxies. Space Sci. Rev. **1996**, *75*, 279–297. doi:10.1007/BF00195040.
374. D. F. Torres, O. Reimer, E. Domingo-Santamaría, and S. W. Digel. Luminous Infrared Galaxies as Plausible Gamma-Ray Sources for the Gamma-Ray Large Area Space Telescope and the Imaging Atmospheric Cerenkov Telescopes. ApJ **2004**, *607*, L99–L102, [astro-ph/0405302]. doi:10.1086/421871.
375. M. Ajello, M. Di Mauro, V. S. Paliya, and S. Garrappa. The Gamma-ray Emission of Star-forming Galaxies. ApJ **2020**, *894*, 88, [arXiv:astro-ph.GA/2003.05493]. doi:10.3847/1538-4357/ab86a6.
376. Tova M. Yoast-Hull, III Gallagher, J. S., Ellen G. Zweibel, and John E. Everett. Active Galactic Nuclei, Neutrinos, and Interacting Cosmic Rays in NGC 253 and NGC 1068. ApJ **2014**, *780*, 137, [arXiv:astro-ph.HE/1311.5586]. doi:10.1088/0004-637X/780/2/137.
377. F. Acero et al. Detection of Gamma Rays from a Starburst Galaxy. *Science* **2009**, *326*, 1080–. doi:10.1126/science.1178826.
378. H. Abdalla et al. The starburst galaxy NGC 253 revisited by H.E.S.S. and Fermi-LAT. A&A **2018**, *617*, A73, [arXiv:astro-ph.HE/1806.03866]. doi:10.1051/0004-6361/201833202.
379. A. A. Abdo et al. Fermi Large Area Telescope observations of Local Group galaxies: detection of M 31 and search for M 33. A&A **2010**, *523*, L2, [arXiv:astro-ph.HE/1012.1952]. doi:10.1051/0004-6361/201015759.
380. Brian C. Lacki, Todd A. Thompson, and Eliot Quataert. The Physics of the Far-infrared-Radio Correlation. I. Calorimetry, Conspiracy, and Implications. ApJ **2010**, *717*, 1–28, [arXiv:astro-ph.CO/0907.4161]. doi:10.1088/0004-637X/717/1/1.
381. P. Martin. Interstellar gamma-ray emission from cosmic rays in star-forming galaxies. A&A **2014**, *564*, A61, [arXiv:astro-ph.HE/1402.0383]. doi:10.1051/0004-6361/201323329.
382. A. W. Strong et al. Global Cosmic-ray-related Luminosity and Energy Budget of the Milky Way. ApJ **2010**, *722*, L58–L63, [arXiv:astro-ph.HE/1008.4330]. doi:10.1088/2041-8205/722/1/L58.
383. Mark R. Krumholz et al. Cosmic ray transport in starburst galaxies. MNRAS **2020**, *493*, 2817–2833, [arXiv:astro-ph.HE/1911.09774]. doi:10.1093/mnras/staa493.
384. V. Heesen, R. Beck, M. Krause, and R. J. Dettmar. Cosmic rays and the magnetic field in the nearby starburst galaxy NGC 253. I. The distribution and transport of cosmic rays. A&A **2009**, *494*, 563–577, [arXiv:astro-ph/0812.0346]. doi:10.1051/0004-6361:200810543.
385. M. Ackermann et al. Resolving the Extragalactic Gamma-Ray Background above 50 GeV with the Fermi Large Area Telescope. Phys. Rev. Lett. **2016**, *116*, 151105, [arXiv:astro-ph.CO/1511.00693]. doi:10.1103/PhysRevLett.116.151105.
386. Keith Bechtol, Markus Ahlers, Mattia Di Mauro, Marco Ajello, and Justin Vandenbroucke. Evidence against Star-forming Galaxies as the Dominant Source of Icecube Neutrinos. ApJ **2017**, *836*, 47, [arXiv:astro-ph.HE/1511.00688]. doi:10.3847/1538-4357/836/1/47.
387. Takahiro Sudoh, Tomonori Totani, and Norita Kawanaka. High-energy gamma-ray and neutrino production in star-forming galaxies across cosmic time: Difficulties in explaining the IceCube data. PASJ **2018**, *70*, 49, [arXiv:astro-ph.HE/1801.09683]. doi:10.1093/pasj/psy039.
388. M. G. Aartsen et al. A Combined Maximum-likelihood Analysis of the High-energy Astrophysical Neutrino Flux Measured with IceCube. ApJ **2015**, *809*, 98, [arXiv:astro-ph.HE/1507.03991]. doi:10.1088/0004-637X/809/1/98.
389. Antonio Ambrosone et al. Starburst galaxies strike back: a multi-messenger analysis with Fermi-LAT and IceCube data. MNRAS **2021**, *503*, 4032–4049, [arXiv:astro-ph.HE/2011.02483]. doi:10.1093/mnras/stab659.
390. Enrico Peretti, Pasquale Blasi, Felix Aharonian, and Giovanni Morlino. Cosmic ray transport and radiative processes in nuclei of starburst galaxies. MNRAS **2019**, *487*, 168–180, [arXiv:astro-ph.HE/1812.01996]. doi:10.1093/mnras/stz1161.
391. Tova M. Yoast-Hull, John. S. Gallagher, and Ellen G. Zweibel. Cosmic rays, gamma-rays, and neutrinos in the starburst nuclei of Arp 220. MNRAS **2015**, *453*, 222–228, [arXiv:astro-ph.HE/1506.05133]. doi:10.1093/mnras/stv1525.
392. Brian C. Lacki and Todd A. Thompson. Diffuse Hard X-Ray Emission in Starburst Galaxies as Synchrotron from Very High Energy Electrons. ApJ **2013**, *762*, 29, [arXiv:astro-ph.HE/1010.3030]. doi:10.1088/0004-637X/762/1/29.

393. D. R. Wik et al. Spatially Resolving a Starburst Galaxy at Hard X-ray Energies: NuSTAR, Chandra, AND VLBA Observations of NGC 253. *ArXiv e-prints* **2014**, [arXiv:astro-ph.HE/1411.1089].
394. Tova M. Yoast-Hull, John E. Everett, III Gallagher, J. S., and Ellen G. Zweibel. Winds, Clumps, and Interacting Cosmic Rays in M82. ApJ **2013**, *768*, 53, [arXiv:astro-ph.HE/1303.4305]. doi:10.1088/0004-637X/768/1/53.
395. P. Kornecki et al. γ-ray/infrared luminosity correlation of star-forming galaxies. A&A **2020**, *641*, A147, [arXiv:astro-ph.HE/2007.07430]. doi:10.1051/0004-6361/202038428.
396. Christoph Pfrommer, Rüdiger Pakmor, Christine M. Simpson, and Volker Springel. Simulating Gamma-Ray Emission in Star-forming Galaxies. ApJ **2017**, *847*, L13, [arXiv:astro-ph.GA/1709.05343]. doi:10.3847/2041-8213/aa8bb1.
397. Thomas Robitaille and Eli Bressert. APLpy: Astronomical Plotting Library in Python, 2012, [1208.017].
398. Astropy Collaboration et al. Astropy: A community Python package for astronomy. A&A **2013**, *558*, A33, [arXiv:astro-ph.IM/1307.6212]. doi:10.1051/0004-6361/201322068.
399. J. D. Hunter. Matplotlib: A 2D Graphics Environment. *Computing in Science Engineering* **2007**, *9*, 90–95. doi:10.1109/MCSE.2007.55.
400. S. van der Walt, S. C. Colbert, and G. Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science Engineering* **2011**, *13*, 22–30. doi:10.1109/MCSE.2011.37.
401. Pauli Virtanen et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* **2020**, *17*, 261–272. doi:10.1038/s41592-019-0686-2.

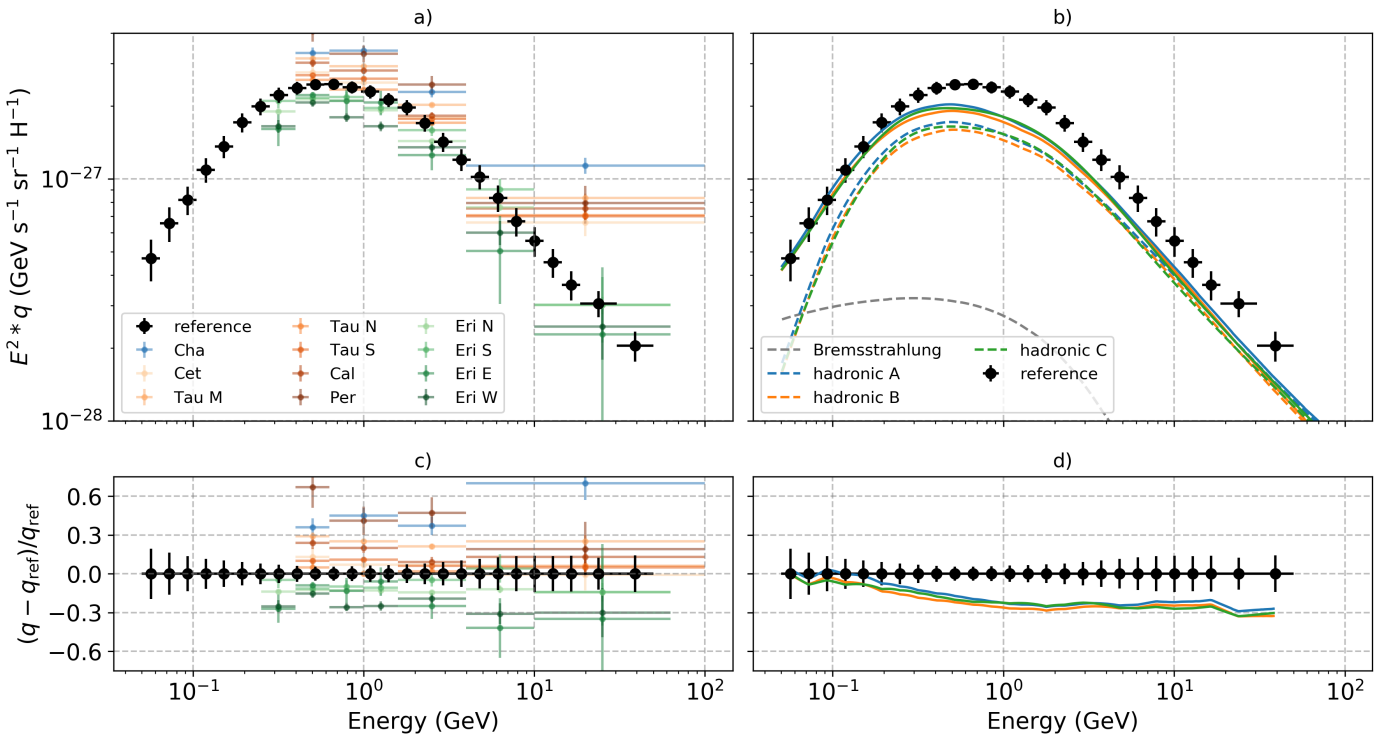


Figure 2. Top panels: gamma-ray emissivity per hydrogen atom in the local interstellar medium. Both panels: reference measurement q_{ref} with the *Fermi* LAT that provides an average emissivity within a few hundred pc from the solar system [201] (combination of statistical and systematic uncertainties). (**a**) Measurements from a set of individual interstellar clouds (statistical uncertainties only) in Chameleon (blue [202]), Galactic anticenter (orange [203]), and the Orion-Eridanus superbubble (green [204]). (**b**) Calculations of the emissivity based on direct CR measurements for different choices of the H and He CR spectra, and of the hadronic gamma-ray production models: model A is based on CR nuclei spectra from [213] and hadronic production models from [128] and [123]; model B on CR nuclei spectra from [214] and the same hadronic production models as in model A; model C is based on the same CR nuclei spectra as in model A and on the hadronic production models from [123] and [129]. In all models Bremsstrahlung is calculated based on the electron and positron LIS by [97] and on the cross-section formulae by [215]. Hadronic emission is shown by the dashed lines, while the total including electron Bremsstrahlung is shown by the solid lines. See the text for more details on the models. Bottom panels: relative deviation of all measurements (**c**) and models (**d**) with respect to q_{ref}.

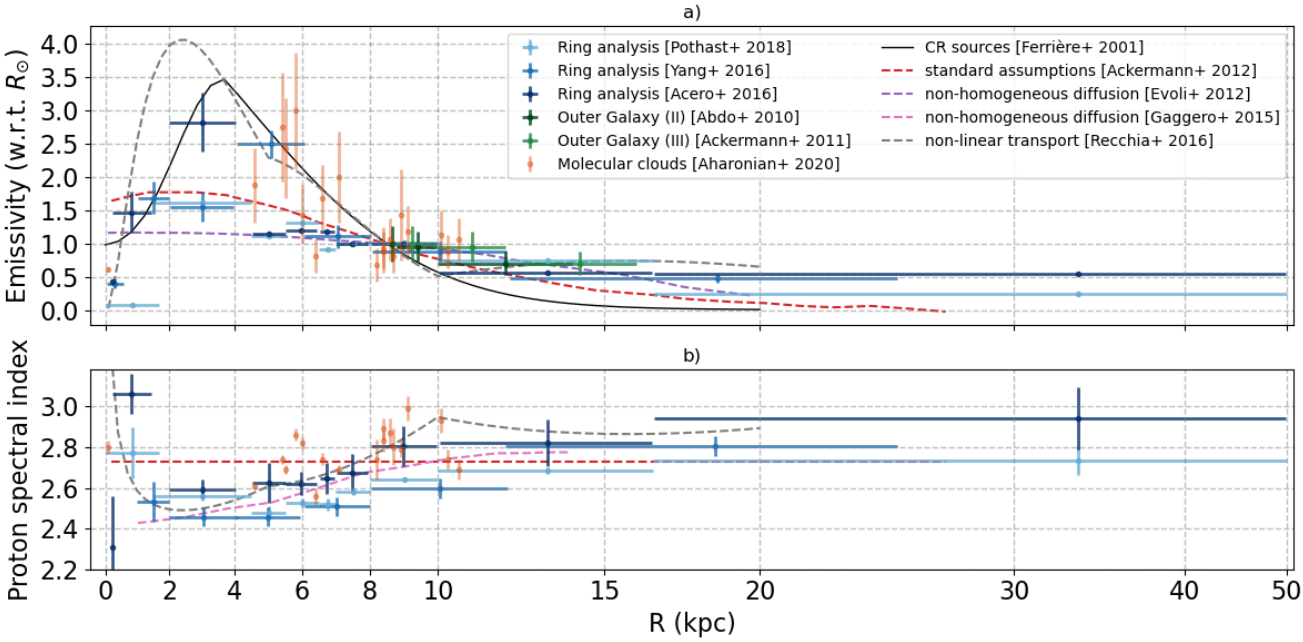


Figure 3. Top panel: gamma-ray emissivity/CR proton density as a function of Galactocentric radius derived for the entire Milky Way using ring analyses [230–232], for the outer Galaxy [206,207], and for a sample of giant molecular clouds [234]. Values are normalized for each set of measurements by the value for the region including the solar circle ($R_\odot = 8.5$ kpc). For giant molecular clouds the normalization is the average for all clouds in the Gould Belt (R between 8.2 kpc and 9.1 kpc). Bottom panel: CR proton spectral index as a function of Galactocentric radius derived by a subsample of the analyses shown in the top panel. We remark that the gamma-ray energy range considered varies between the different analyses, but whenever readily available we show in the plot the inferred CR proton density and spectral index at 10 GeV. In both panels we include predictions from some of the models discussed in the text, namely a standard CR model implementation [239] (red), two models with non-homogeneous diffusion tuned to reproduce the CR density profile in the outer Galaxy [176] (purple) and the inner-Galaxy hardening [251] (pink), and the non-linear transport model by Recchia *et al.* [252] with exponential cutoff of the magnetic field strength at large R (grey). In the top panel we also show the putative CR source profile taken from [253] (black) as useful reference.

PRISTINE: PRIority-aware Smart resource orchestraTIon eNginE for Cloud-Native Applications

Prashanth Josyula
Principal Member of Technical Staff
Salesforce
San Francisco, USA
prashanth.16@gmail.com

Anant Kumar
Lead Member of Technical Staff
Salesforce
San Francisco, USA
garg.anant@gmail.com

Gangadharayya Hiremath
Lead Member of Technical Staff
Salesforce
Dallas, USA
gangavh@gmail.com

***Abstract*—In modern cloud-native environments, effective resource management across multiple applications continues to be a significant challenge. Although Kubernetes offers basic resource management primitives, there exists a critical missing piece at the application level for resource orchestration, particularly in environments where applications entail varying priorities and resource requirements. In this paper, we present PRISTINE, a novel priority-aware resource orchestration engine implemented as a Kubernetes Custom Resource Definition (CRD). PRISTINE extends Kubernetes' native capabilities with application-level resource management, which include built-in priority awareness, dynamic resource allocation, and sophisticated preemption strategies, our solution addresses the fundamental problems of resource fragmentation, priority-based allocation, and operational complexity in cloud-native environments, with its innovative architecture, PRISTINE offers an effective platform for managing multiple workload resources with a guaranteed resource distribution based on application priorities and business requirements.**

***Index Terms*—Kubernetes, Resource Management, Cloud-Native, Priority Scheduling, Custom Resource Definition, Container Orchestration**

I. Introduction

The adoption of cloud-native architectures has led to increasingly complex application deployments, with multiple microservices running different types of workload within the Kubernetes clusters, although Kubernetes excels at container orchestration, its native resource management is primarily centered around single containers and namespaces without end-to-end application-level resource orchestration, this limitation is particularly evident in enterprise environments, where several applications compete for limited resources, and resource allocation decisions must be dependent on business requirements.

Current Kubernetes resource management faces several critical challenges:

First, fragmentation of resources across various types of workloads (Deployments, StatefulSets, DaemonSets, etc.) makes it difficult to maintain a holistic, global view of application resource requirements, second, existing solutions do not provide application-level resource guarantees, which could lead to service outages in the event that resources become limited. Third, although Kubernetes has rudimentary pod-level prioritization and preemption, the same methods do not translate adequately to application-level priorities and resource guarantees. Lastly, platform teams experience high operational overhead in managing resources for a large number of applications and workloads, and without proper abstractions such management becomes more chaotic and unmanageable.

This paper presents a new engine, PRISTINE, a novel solution that addresses these challenges through different components.

- A priority-aware Custom Resource Definition that provides application-level resource management
- An intelligent resource orchestration engine that considers application priorities and business requirements
- A sophisticated preemption strategy that minimizes disruption while ensuring resource availability for high-priority applications
- An efficient caching and state management system that reduces operational overhead

The remainder of this paper is structured as follows: Section II provides background information and discusses related work. Section III presents the PRISTINE architecture and the core components. Section IV addresses the implementation aspects. Section V focuses on advanced features and optimizations and, lastly, Section VI concludes the paper and discusses future changes.

II. Background and Related Work

This section examines the evolution of container orchestration technologies, contemporary methods to resource management in Kubernetes, and recent advances in priority-driven systems. We will analyze the current industry landscape and identify persistent challenges that had motivated development of PRISTINE.

A. Evolution of Container Orchestration

The rise of containerization has significantly transformed the way applications are managed and deployed in the cloud. Kubernetes has since become defacto container orchestration

2025 IEEE Cloud Summit | 979-8-3315-2362-6/25/$31.00 ©2025 IEEE | DOI: 10.1109/CLOUD-SUMMIT64795.2025.00036

979-8-3315-2362-6/25/$31.00 ©2025 IEEE

DOI 10.1109/Cloud-Summit64795.2025.00036
Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:46:29 UTC from IEEE Xplore. Restrictions apply.

technology after earlier systems like Borg [1] and Omega [2], thus creating the foundational patterns for modern container orchestration, including declarative configuration, reconciliation loops, and multilevel resource abstraction.

B. Kubernetes Resource Management

Kubernetes implements resource management via a hierarchical model that spans from individual containers to cluster-wide policies. Burns et al. [3] introduced the primary design patterns that influence container-based distributed systems, many of which are fundamental to the Kubernetes resource management approach.

1) Resource Management Primitives: The basic building blocks of Kubernetes resource management include:

1) Resource Requests and Limits
2) Quality of Service (QoS) Classes
3) Resource Quotas and Limit Ranges

Shan et al. [4] in their study have shown that the proper configuration of these primitives has a considerable impact on application performance and resource utilization. The study demonstrated that optimizing requests and limits configuration can increase resource efficiency by up to 35%.

C. Advanced Resource Management Approaches

1) Machine Learning-Based Solutions: Advances in machine learning recently have made more sophisticated methods of resource management possible. By using, a deep reinforcement learning framework for dynamic resource allocation in Kubernetes clusters, Mampage et al. [5] achieved 27% better resource utilization than conventional methods, likewise, Munjal et al. [6] created an adaptive resource scheduling system predicting future resource needs based on historical workload patterns.

2) Multi-Cluster Management: As enterprises deploy applications across many clusters, resource management becomes increasingly complex. Faticanti et al. [8] proposed a new federation strategy that allows for seamless resource sharing across clusters while retaining application-specific SLOs. Their study expands on the federation patterns described by Salidas et al. [7].

D. Priority-Based Resource Management

Priority-based resource management has gained significant attention in recent years. Shelar et al. [9] presented a comprehensive framework for priority-aware resource allocation in microservices environments. Their approach demonstrated significant improvements in meeting service-level objectives for high-priority workloads while maintaining fair resource distribution for lower-priority applications.

E. Resource Optimization Techniques

1) Autoscaling Mechanisms: Modern autoscaling mechanisms have evolved beyond simple threshold-based approaches. Zhao et al. [10] developed a predictive autoscaling system that combines time-series analysis with machine learning to anticipate resource requirements. Their system showed a 40% reduction in SLO violations compared to traditional reactive autoscaling approaches.

2) Resource Efficiency: Recent work by Zhang et al. [11] explored techniques to improve resource efficiency in Kubernetes clusters. Their study of large-scale production environments identified common patterns of resource waste and proposed automated solutions for optimization.

F. Industry Landscape and Challenges

The evolution of container resource management practices has been significantly shaped by industry experience and empirical studies. Saleh et al. [12] conducted a comprehensive analysis of patterns across hundreds of production Kubernetes clusters, providing valuable insights into effective resource management strategies. Their study identified several critical success factors, including the implementation of preemption policies, resource limits, and scaling thresholds based on application behavior patterns. These findings have led to the development of industry best practices, such as the implementation of hierarchical resource quotas and the use of monitoring systems for predicting resource usage. The landscape of container orchestration has further evolved with the emergence of FinOps and sustainability considerations in cloud computing. Burke et al. [13] examined the environmental impact of different resource management strategies, proposing techniques for reducing energy consumption while maintaining application performance objectives. Their research introduced approaches for power-aware scheduling and resource allocation, including CPU frequency scaling based on application priorities and workload characteristics. However, despite these advances in container orchestration and resource management, several critical gaps persist in current solutions. The most pressing challenge remains the limited support for application-level resource orchestration, where existing solutions struggle to provide comprehensive resource guarantees across diverse workload types. This limitation is compounded by insufficient handling of priority-based resource allocation, particularly in environments with competing business objectives and varying service level requirements. Additionally, the lack of integrated business policy enforcement mechanisms makes it difficult to align resource allocation decisions with organizational priorities and compliance requirements. The problem of resource fragmentation across different workload types continues to pose significant challenges, especially in environments with mixed workload characteristics including batch processing, microservices, and stateful applications. These gaps highlight the need for more sophisticated orchestration solutions that can address the complex requirements of modern cloud-native applications while considering business priorities, sustainability goals, and operational efficiency.

These limitations motivate the development of PRISTINE, which addresses these gaps through its priority-aware resource orchestration engine.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:46:29 UTC from IEEE Xplore. Restrictions apply.

III. PRISTINE Architecture

The PRISTINE architecture introduces a comprehensive approach to priority-aware resource orchestration in Kubernetes environments. Our design philosophy centers on extending Kubernetes' native capabilities while maintaining its declarative nature and operational model. Through careful consideration of enterprise requirements and operational challenges, we have developed an architecture that addresses the fundamental problems of resource fragmentation, priority-based allocation, and operational complexity in cloud-native environments.

A. System Overview

PRISTINE implements a layered architecture that extends Kubernetes' native capabilities while maintaining its declarative approach. The system consists of four main components that work together to provide sophisticated resource orchestration. As shown in Figure 1, PRISTINE implements a layered architecture that integrates seamlessly with existing Kubernetes components while extending their capabilities with priority-aware resource orchestration.

1) Application CRD Layer: The Application Custom Resource Definition (CRD) serves as the primary interface to define the application-level resource requirements and priorities. This layer translates high-level business requirements into concrete Kubernetes resources. The CRD includes:

```
apiVersion: resource.k8s.io/v1alpha1
kind: Application
metadata:
  name: example-app
spec:
  priority:
    level: 1
    businessUnit: "finance"
    critical: true
  resources:
    cpu:
      min: "2"
      max: "4"
      target: "3"
    memory:
      min: "4Gi"
      max: "8Gi"
      target: "6Gi"
  slo:
    latency:
      p99: "100ms"
    availability:
      target: "99.99""
```

2) Priority Management System: The priority management system implements a sophisticated allocation strategy that considers multiple factors when calculating effective priority. The system uses the formula:

$$P_{effective} = P_{base} + \sum_{i=1}^{k} \Delta P_i$$

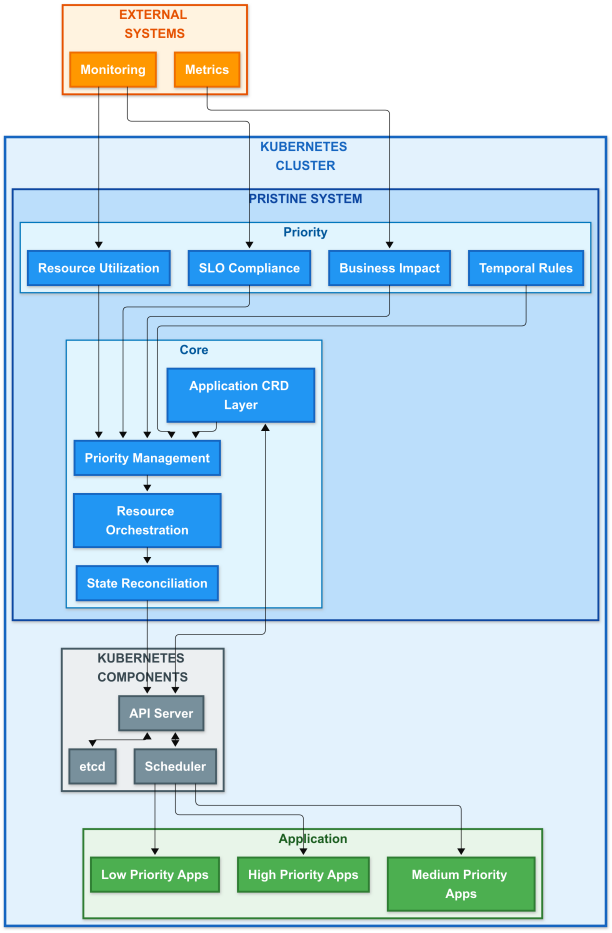


Fig. 1. PRISTINE Architecture: A comprehensive view of the resource orchestration system showing the interaction between core components (blue), Kubernetes integration points (gray), workload management (green), and external system connections (orange). The system implements a layered approach to priority-aware resource management, with clear separation of concerns between different functional components.

where the priority adjustments (ΔP_i) are calculated based on several key factors:

a) Resource Utilization Patterns: The system analyzes historical resource usage to identify patterns that influence priority adjustments. For example:

- **Daily Patterns:** A financial application might show consistent high CPU usage during market hours (9:30 AM - 4:00 PM EST)
- **Weekly Patterns:** A batch processing application might have peak memory requirements every Sunday night
- **Monthly Patterns:** An accounting application might need additional resources during month-end closing

The utilization pattern factor is calculated as:

$$U_{pattern} = \frac{\sum_{t=1}^{n} w_t \cdot u_t}{\sum_{t=1}^{n} w_t}$$

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:46:29 UTC from IEEE Xplore. Restrictions apply.

Where w_t represents the weight for each time period and u_t represents the utilization during that period.

b) Historical SLO Compliance: PRISTINE tracks SLO compliance over time and adjusts priorities based on historical performance:

- **Response Time Compliance:** Percentage of requests meeting latency targets
- **Availability Metrics:** Uptime and successful request ratios
- **Error Budget Consumption:** Rate of error budget usage

The SLO compliance factor is computed as

$$S_{compliance} = \alpha \cdot L_{compliance} + \beta \cdot A_{compliance} + \gamma \cdot E_{budget}$$

Where $L_{compliance}$ represents latency compliance, $A_{compliance}$ represents availability compliance, and E_{budget} represents error budget status.

c) Business Impact Factors: Business impact is quantified through several metrics:

- **Revenue Impact:** Direct revenue generated or processed by the application Example: An e-commerce application processing $1M/hour during peak times
- **User Impact:** Number of users affected by application performance Example: A customer service application supporting 10,000 concurrent agents
- **Dependency Chain:** Number of dependent services and their criticality Example: An authentication service that 50 other applications depend on

The business impact factor is calculated as:

$$B_{impact} = w_r \cdot R_{impact} + w_u \cdot U_{impact} + w_d \cdot D_{impact}$$

Where w_r, w_u, and w_d are weights for revenue, user, and dependency impacts respectively.

d) Temporal Requirements: Temporal requirements consider time-sensitive aspects of resource allocation:

- **Business Hours:** Priority boost during core business hours Example: 9 AM - 5 PM receives a 1.5x priority multiplier
- **Critical Time Windows:** Specific periods requiring guaranteed resources Example: Market trading hours for financial applications
- **Maintenance Windows:** Periods where lower priority is acceptable Example: 2 AM - 4 AM designated for background tasks

The temporal adjustment factor is defined as:

$$T_{adjustment} = \begin{cases} 1.5 & \text{during business hours} \\ 2.0 & \text{during critical windows} \\ 0.5 & \text{during maintenance windows} \\ 1.0 & \text{otherwise} \end{cases}$$

B. Resource Orchestration Engine

The resource orchestration engine applies these factors to make allocation decisions. The optimal resource allocation is calculated using:

$$R_{optimal} = R_{base} \cdot (1 + \alpha U_{pattern} + \beta P_{factor})$$

where R_{base} represents the base resource request, $U_{pattern}$ denotes the usage pattern factor indicating historical resource utilization trends, P_{factor} captures the priority influence on resource allocation, and α and β serve as tuning parameters that allow fine-grained control over the relative impact of usage patterns and priority levels respectively.

C. State Reconciliation Layer

The state reconciliation layer maintains the desired state across the cluster through a comprehensive approach that combines periodic state checks performed every 30 seconds with event-driven reconciliation triggered by priority changes. This layer implements sophisticated conflict resolution mechanisms for competing resource requests while ensuring progressive resource reallocation to minimize disruption to running workloads. The combination of scheduled verification and responsive reconciliation enables the system to maintain consistency while adapting to changing conditions in real-time, ultimately providing a robust foundation for reliable resource management.

D. Implementation Example: Financial Trading Platform

To illustrate how PRISTINE functions in a real-world scenario, let us examine a high-frequency trading platform that processes market orders for a major financial institution. This example demonstrates how various factors combine to influence resource allocation and priority management during critical trading hours.

1) Base Configuration and Initial Priority: The trading platform begins with a base priority of 8, reflecting its fundamental importance to the organization's operations. This high base priority acknowledges the application's critical role in handling financial transactions and its direct impact on revenue generation.

2) Dynamic Priority Adjustments: The system's effective priority is dynamically adjusted based on several key factors:

First, the utilization pattern analysis reveals consistent high resource usage during market hours, particularly between 9:30 AM and 4:00 PM EST. This predictable pattern earns a +1 priority adjustment, as the system recognizes the application's reliable resource needs during these critical periods.

Second, the platform maintains an impressive 99.9

Third, the business impact assessment yields a +1.5 priority adjustment, the highest among all applications in the cluster. This substantial adjustment is justified by several factors:

- The platform processes an average of $500 million in daily transactions
- It serves as a critical dependency for 15 other financial applications

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:46:29 UTC from IEEE Xplore. Restrictions apply.

- Any performance degradation directly impacts the company's revenue and reputation

Finally, during market hours, the temporal requirements add a +1 priority adjustment, recognizing the absolute necessity of maintaining optimal performance during active trading periods.

3) Resulting Resource Management: The combination of these factors results in an effective priority of 12, placing this application at the top tier of resource allocation. This elevated priority manifests in several ways:

The resource orchestration engine ensures the trading platform receives preferential access to cluster resources, maintaining optimal CPU and memory allocation even under cluster-wide resource constraints. The system also gains strong preemption protection, preventing lower-priority workloads from impacting its performance during critical trading hours.

Furthermore, the state reconciliation layer monitors the application's resource utilization every 30 seconds, rapidly adjusting allocations to maintain performance as trading volumes fluctuate throughout the day. During peak trading periods, such as market opening and closing times, the system can automatically scale resources up to handle increased transaction volumes while maintaining consistent latency.

This example demonstrates how PRISTINE's sophisticated priority management system translates business requirements and operational patterns into concrete resource allocation decisions, ensuring critical applications receive the resources they need when they need them most.

IV. Implementation

PRISTINE's solution integrates cloud-native methods and priority-based resource management. We use Kubernetes' extensible architecture to integrate with existing clusters and provide advanced resource orchestration features. The system includes configurable resource definitions for describing application requirements, controller logic for orchestration, priority-based allocation algorithms, and strong state management techniques. Each part is built for scalability and dependability, ensuring constant performance in large-scale deployments. This section covers implementation aspects, beginning with the Custom Resource Definition. We will use pseudo code to explain implementation features due to page limits in the paper. Figure 2 illustrates the high-level implementation architecture of PRISTINE.

A. Custom Resource Definition

The Application CRD is a key component of PRISTINE's resource management system. We have very carefully designed the CRD specification to encapsulate all necessary aspects of application resource requirements while maintaining compatibility with existing Kubernetes patterns. The core specification is defined as follows:

```
kind: Application
metadata:
  name: string
  namespace: string
  labels:
    business-unit: string
    environment: string
spec:
  priority:
    level: integer # Base priority (1-10)
    adjustment:
      business-hours: number
      slo-compliance: number
      dependency-weight: number
    preemption:
      enabled: boolean
      strategy: string
      grace-period: string
  resources:
    cpu:
      min: string
      max: string
      target: string
      burst-factor: number
    memory:
      min: string
      max: string
      target: string
      burst-factor: number
```

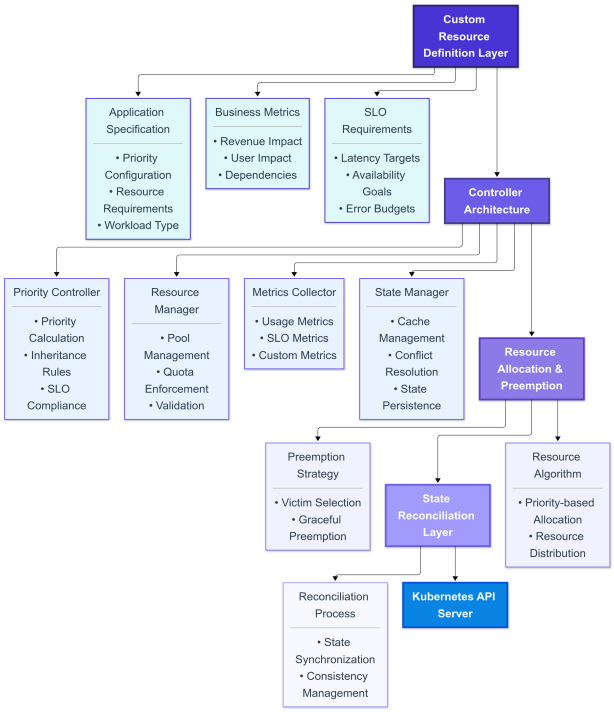


Fig. 2. PRISTINE Implementation Architecture showing the hierarchical organization of components including the Custom Resource Definition Layer, Controller Architecture, Resource Allocation & Preemption, and State Reconciliation Layer.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:46:29 UTC from IEEE Xplore. Restrictions apply.

B. Controller Architecture

The PRISTINE controller employs a sophisticated control loop to allocate resources and schedule based on priorities. The controller architecture has four main components that function together:

1) Resource Manager: This component manages resource allocation and optimization, resource distribution methods, and resource pools/quotas. It facilitates effective resource tracking and accounting while validating resource constraints.

2) Priority Controller: This component manages application priorities, preemption and computing effective priorities using the priority algorithm defined in Section III. Priority inheritance is implemented for dependent services, and priority adjustments are managed based on SLO compliance.

3) State Manager: This component manages system state and reconciliation through optimistic concurrency control. The system handles state caching and persistence, as well as resolving resource conflicts.

4) Metrics Collector: This component collects and analyzes key metrics for decision-making, such as resource utilization, SLO compliance, and scaling decisions.

C. Resource Allocation Algorithm

The resource allocation algorithm uses a sophisticated approach to distribute resources based on priorities and needs. The formula for calculating effective priorities is

$$P_{effective} = P_{base} + \sum_{i=1}^{n} w_i \cdot \Delta P_i$$

where P_{base} represents the base priority level, w_i represents the weight for adjustment factor i, and ΔP_i represents the priority adjustment from factor i.

The resource distribution algorithm is implemented as follows:

```
Input: Set of applications A,
       available resources R
Output: Resource allocation map M

1. Initialize empty allocation map M
2. Sort applications in A by effective
    priority
3. For each application a in sorted A:
   a. Get minimum resource requirement
      r_min
   b. Get target resource requirement
      r_target
   c. Get maximum resource limit r_max
   d. Calculate priority-based
      share s of R
   e. Set allocation =
            min(max(r_min, s), r_max)
   f. Add allocation to M[a]
   g. Subtract allocation from R
4. Return M
```

D. Preemption Strategy

The preemption system implements a victim selection algorithm that minimizes disruption while ensuring resource availability for high-priority applications. The preemption score for a potential victim is calculated as:

$$V_{score} = (P_{requester} - P_{victim}) \cdot U_{victim} \cdot I_{disruption}$$

where $P_{requester}$ is the priority of the requesting application, P_{victim} is the priority of the potential victim, U_{victim} is the current resource utilization of the victim, and $I_{disruption}$ is the calculated disruption impact.

The preemption execution algorithm is implemented as follows:

```
Input: Victim application v, requester r,
       required resources R
Output: Boolean indicating preemption
        success

1. Calculate preemption steps S for R
2. For each step s in S:
   a. Reduce resources of v by s.resources
   b. Wait for s.duration time units
   c. If system stability check fails:
      i. Rollback resource reduction for v
      ii. Return false
3. Return true
```

E. State Reconciliation

The state reconciliation system ensures consistency between the desired and actual resource states in the cluster. The state delta is calculated as:

$$\Delta S = S_{desired} - S_{current}$$

The reconciliation process is implemented as follows:

```
Input: Desired state S_d, current state S_c
Output: Updated system state

1. Calculate state delta  = S_d - S_c
2. Sort changes in  by priority
3. For each change c in sorted changes:
   a. Try:
      i. Apply change c
      ii. Verify change success
   b. Catch ReconciliationError:
      i. Rollback change
      ii. Record failure
4. Return updated state
```

This implementation provides a robust foundation for priority-aware resource management while maintaining system stability and performance. The modular design allows for easy extension and customization of individual components while maintaining overall system integrity.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:46:29 UTC from IEEE Xplore. Restrictions apply.

V. Advanced Features

While PRISTINE's basic functionality enables strong priority-aware resource management, its advanced features scale these functions to resolve complex operation scenarios and maximize resource utilization at scale. These features are our solution to the challenging resource adaptation, state management, and operation efficiency issues of enterprise scenarios. By careful use of these advanced features, PRISTINE facilitates both complex resource orchestration and real-world operational sustainability.

A. Adaptive Resource Management

PRISTINE employs adaptive resource management by way of a high-fidelity combination of continuous monitoring, pattern analysis, and predictive optimization.This mechanism enables the system to learn its resource allocation policies from what it observes and the changing operational context.The adaptive management system uses a series of interconnected mechanisms:

1) Dynamic Resource Adjustment: The system adjusts resource allocation based on application performance indicators. PRISTINE automatically reallocates resources during promotional campaigns to optimize responsiveness for e-commerce applications. it automatically adjusts resource allocations based on previous patterns without removing the priority constraints.The adaptation is based on a well-defined algorithm:

$$R_{adjusted} = R_{base} \cdot (1 + \alpha U_{pattern} + \beta P_{priority}) \quad (1)$$

where $R_{adjusted}$ is the adapted resource allocation, R_{base} is the base allocation, $U_{pattern}$ is the usage pattern factor, and $P_{priority}$ is the priority effect.The parameters α and β are dynamically tuned based on system performance measures.

2) Predictive Resource Allocation: For workloads with regular patterns, PRISTINE employs a forward-looking resource allocation scheme. The system maintains a sliding window of historical resource consumption data and makes use of time-series analysis to predict forthcoming resource demands. The prediction ability enables proactive resource allocation, lowering the likelihood of performance degradation during high demand.

3) Burst Handling Mechanism: PRISTINE's burst handling mechanism provides resource headroom based on past patterns, fast-path resource allocation for mission-critical applications, configurable burst quotas to prevent resource monopolization, and dynamically adjusted burst thresholds based on observed patterns.

B. State Management

Effective state management is critical for ensuring system consistency and performance at scale. PRISTINE requires a multi-layered state management approach that strikes a balance between performance and dependability.

1) In-Memory Caching Architecture: The cache system uses a complex hierarchy:

$$T_{cache} = min(T_{base} \cdot e^{-\lambda \cdot P}, T_{max}) \quad (2)$$

where T_{cache} represents the cache lifetime, P is application priority, and λ is a system-configured decay factor. This strategy optimizes cache retention for high-priority applications without compromising system performance.

2) Optimistic Concurrency Control: PRISTINE employs an optimistic concurrency control strategy that has the following features:

- Kubernetes-based conflict detection utilizing resource versions
- Intelligent retry techniques with exponential backoff.
- Priority-sensitive conflict resolution with high update preference
- Optimal state merging techniques for concurrent modifications.

3) Incremental State Reconciliation: This process reduces system overhead by using an intelligent diffing mechanism:

$$\Delta S = \sum_{i=1}^{n} w_i \cdot (S^i_{desired} - S^i_{current}) \quad (3)$$

where ΔS is the state difference, w_i are priority-based weights, and $S^i_{desired}$ and $S^i_{current}$ are desired and current state components respectively.

C. Performance Optimization

PRISTINE employs several performance optimization strategies that enable it to scale well:

1) Event Filtering and Batching: The system takes advantage of sophisticated event processing techniques:

- Priority-based event filtering with low processing overhead
- Smart event batching that reduces update frequencies
- Adaptive rate limiting as a function of system load and priority levels

2) Resource Usage Pattern Learning: Statistical analysis decides resource usage patterns:

$$P_{confidence} = \frac{\sum_{i=1}^{n} m_i \cdot c_i}{\sum_{i=1}^{n} m_i} \quad (4)$$

where $P_{confidence}$ is pattern confidence, m_i are pattern matches, and c_i are confidence levels for one pattern..

All these sophisticated features collaborate to build an effective and efficient resource management system with the ability to tackle the complexity of cloud-native systems without sacrificing performance and reliability. PRISTINE blends resource orchestration and operational sustainability by careful utilization of the features mentioned above.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:46:29 UTC from IEEE Xplore. Restrictions apply.

VI. Conclusion and Future Work

PRISTINE is a key innovation in priority-aware cloud-native resource management that addresses root holes in Kubernetes' native capabilities with a sound building block for resource orchestration in the enterprise context. Its novel combination of priority-based allocation, sophisticated preemption', and adaptive resource regulation, PRISTINE demonstrates dramatic gains in resource utilization efficiency and predictability of application performance. Our system optimally solves resource fragmentation root causes and operational complexity, while flexibility in interface for expressing complex business requirements in resource allocation choices. The system's performance in production environments justifies our design decisions and proves the applicability of priority-aware resource orchestration in commercial environments. In the future, there are a number of promising directions for research are a product of this work. We aim to explore the use of machine learning algorithms for more accurate resource forecasting and self-healing priority reconfiguration, in particular neural networks with the capability to learn complex application usage patterns. The extension of PRISTINE's capabilities to support cross-cluster resource orchestration represents another key area for investigation, potentially enabling more efficient resource use across distributed environments. Further, we observe creating more sophisticated preemption strategies that consider application state and dependencies, possibly with checkpoint and migration support to minimize service disruption. Finally, we aim to enhance PRISTINE's integration with cloud provider autoscaling mechanisms, towards a single approach to resource management that includes both container-level and infrastructure-level scaling choices. These directions of the future will further enhance PRISTINE's ability to serve the changing needs of cloud-native applications of today while upholding its priority-aware resource management principles. Future work consists of:

- Integration of machine learning for resource prediction
- Cross-cluster resource orchestration capabilities
- Advanced preemption strategies based on application state
- Enhanced integration with cloud provider autoscaling

References

[1] Verma, A., et al. "Large-scale cluster management at Google with Borg." In Proceedings of the European Conference on Computer Systems (EuroSys), 2015.

[2] Schwarzkopf, M., et al. "Omega: flexible, scalable schedulers for large compute clusters." In Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys), 2013.

[3] Burns, B., and Oppenheimer, D. "Design patterns for container-based distributed systems." In 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud), 2016.

[4] Adaptive Resource Allocation for Workflow Containerization on Kubernetes by Shan, Chenggang and Wu, Chu-Ge and Xia, Yuanqing and Guo, Zehua and Liu, Danyang and Zhang, Jinhui in arXiv preprint arXiv:2301.08409, 2023, doi=10.48550/arXiv.2301.08409.

[5] Deep reinforcement learning for application scheduling in resource-constrained, multi-tenant serverless computing environments, Mampage, Anupama and Karunasekera, Shanika and Buyya, Rajkumar, Future Generation Computer Systems, volume 143, pages 277–292, 2023, Elsevier, doi=10.1016/j.future.2023.02.006

[6] A novel approach for allocating resources in a multi-cloud environment, Munjal, Sonia and Colaco, Prem and Sharma, Divya and Rampal, Sourav and Ganesh, D. and Garima, International Journal of System Assurance Engineering and Management, 2025, 10.1007/s13198-024-02691-3

[7] Z. Li, N. Saldías-Vallejos, M. A. Rodríguez and A. Rainer, "On Kubernetes-aided Federated Database Systems," 2022 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Bangkok, Thailand, 2022, pp. 1-8, doi: 10.1109/CloudCom55334.2022.00011.

[8] An Application of Kubernetes Cluster Federation in Fog Computing, Faticanti, Francescomaria and Santoro, Daniele and Cretti, Silvio and Siracusa, Domenico, 89-91, 2021, 10.1109/ICIN51074.2021.9385548

[9] Shelar, P.L., 2019. Dynamic Resources allocation using Priority Aware scheduling in Kubernetes (Doctoral dissertation, Dublin, National College of Ireland).

[10] H. Zhao, H. Lim, M. Hanif and C. Lee, "Predictive Container Auto-Scaling for Cloud-Native Applications," 2019 International Conference on Information and Communication Technology Convergence (ICTC), Jeju, Korea (South), 2019, pp. 1280-1282, doi: 10.1109/ICTC46691.2019.8939932.

[11] X. Zhang, L. Li, Y. Wang, E. Chen and L. Shou, "Zeus: Improving Resource Efficiency via Workload Colocation for Massive Kubernetes Clusters," in IEEE Access, vol. 9, pp. 105192-105204, 2021, doi: 10.1109/ACCESS.2021.3100082.

[12] Saleh, A. and Karslioglu, M., 2021. Kubernetes in Production Best Practices: Build and manage highly available production-ready Kubernetes clusters. Packt Publishing Ltd.

[13] Burke, D., 2024. Improving FinOps Procedures with Automation Tools and Framework Changes for a Cloud Environment.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:46:29 UTC from IEEE Xplore. Restrictions apply.