SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Designing and Building an Active Monitoring Platform with Prometheus Ecosystem

Jih-Wei Liang

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Designing and Building an Active Monitoring Platform with Prometheus Ecosystem

Entwerfen und Aufbauen einer aktiven Überwachungsplattform mit dem Prometheus-Ökosystem

Author:	Jih-Wei Liang
Supervisor:	Michael Gerndt; Prof. Dr.-Ing.
Advisor:	Samuel Torre Vinuesa
Submission Date:	15.03.2024

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.03.2024 Jih-Wei Liang

Acknowledgments

Abstract

Contents

1 Introduction

System monitoring is essential in the modern Information Technology (IT) industry. It offers observability to both users and developers, enabling rapid detection and notification of issues. For Amadeus's extensive network of interlinked applications, observability is vital to swiftly identifying, diagnosing, and resolving problems, ensuring our customers receive the highest quality service.

The Amadeus Observability Platform accommodates a variety of monitoring data, including metrics, logs, structured logs, and events. It is built on the Prometheus ecosystem and integrates with Splunk. Additionally, Amadeus actively contributes to the Prometheus ecosystem. A prime example is Perses, a new CNCF project developed and maintained by Amadeus that aims at observability data visualization.

Recently, demands for probing on kinds of applications in Amadeus Cloud Service (ACS) by customized requests, such as those of Hypertext Transfer Protocol (HTTP)/Hypertext Transfer Protocol Secure (HTTPS), Domain Name System (DNS), Internet Control Message Protocol (ICMP), Simple Network Management Protocol (SNMP), etc, have increased, which stemmed from the increasing focus on observability. This kind of monitoring actively sends simulated requests and observes the response without internal knowledge, which could benefit observability with low cost and fast reaction. As part of the overall strategy for rock-solid operations, Amadeus needs this kind of monitoring on scale.

As mentioned above, the tendency brings about the demand for introducing active monitoring or Black Box Monitoring to Amadeus. By definition, active monitoring features the simulation of requests and observation of responses, covering the scope of Black Box monitoring. However, the two monitoring methods have slight differences in concepts. Active monitoring emphasizes proactively watching the reactions to simulated behaviors [Spl23]. On the other side, Black Box Monitoring focuses on ignoring the internal mechanism, only caring about the input and output of the application [Bey+16]. In conclusion, although the two monitoring methods have different names and concepts, they still feature similar behaviors and fit the latest monitoring requirements in Amadeus.

Following alignment meetings with Platform Product Management and Tech Senior Leaders, a consensus was reached regarding the requirements for the final solution in active monitoring. The solution must be highly scalable, reliable, and cloud-ready to meet the needs of a modern IT environment. It should support multiple protocols such as HTTP, Simple Object Access Protocol (SOAP), Representational State Transfer (REST), Electronic Data Interchange for Administration, Commerce and Transport (EDIFACT), etc. Additionally, the system must be modular to facilitate the easy addition of new protocols.

Additionally, other essential features concerning the Operating Model should be considered. For example, the active monitoring platform must be self-service and fully support "as-code" configurations such as GitOps. Also, it should support deploying probers and selecting their origin and target with an easy interface. Lastly, seamless integration with the Amadeus Observability Platform, including the Event Management stack, is crucial to ensure a cohesive and efficient monitoring ecosystem.

Nowadays, there are renowned tools to implement system monitoring, such as Prometheus, Datadog, Nagios, and so on. These tools aim to collect various metrics from targets for monitoring computational resources. Generally, they provide an agent to scrape desired data and then wait for the server's request or actively push these data to the responsible server. Most of them feature great professions in monitoring, offering enterprise-level services in reaction to kinds of requirements [NVP21].

This project considers three solutions to align with the Amadeus Observability Platform. The first is a customized application with scheduled probings, probing targets, and a centralized user interface. The second option involves deploying a Prometheus Agent and Blackbox Exporter, accompanied by an additional control plane. The third solution explores using the Prometheus Operator, Open Policy Agent, and GitOps.

The first solution comprises three main components: the Operator, the Scheduler, and the Prober. The Operator is responsible for handling the CRD that defines the service and related information for monitoring. The Scheduler is tasked with scheduling monitoring requests based on the CRD and collecting data from the Probers. Lastly, the Prober is a custom worker designed for various monitoring purposes. It sends target requests and then returns the data required to the Scheduler. The significant advantage of this method is full customization; however, the disadvantages are the heavy maintenance load and the inconsistency with the open-source framework.

The second solution is similar to the first one. The difference is that the scheduler is replaced with the Prometheus Agent, and general probers are replaced with the Blackbox Exporter, except for some customized probers of specialized protocols in Amadeus. With these changes mentioned above, a well-designed open-source framework is employed, decreasing the maintenance load and hidden danger of homemade solutions.

The third solution introduces the open-source implementation for managing different components in the Prometheus ecosystem with the Operator Pattern, the Prometheus

Operator. Like the operator in the first solution, the Operator Pattern allows users to automate configurations and management in container orchestrators like Kubernetes and OpenShift [Kub]. Therefore, the Prometheus Operator helps users deploy and manage the components in the ecosystem, which are needed for active monitoring, achieving sharding and auto scalability without additional effort [Ope20]. On the other hand, users must understand and utilize official CRDs to conduct settings, which brings some learning curve to the promoting.

In selecting the final solution, the third option involving the Prometheus Operator stands out as the preferred choice for several reasons. Firstly, the Prometheus Operator can be seen as an enhanced and official version of the first solution, offering improved compatibility. Additionally, while the second solution requires extra effort to attain scalability and availability, the Prometheus Operator naturally includes automated configuration management and sharding capabilities. Lastly, the Prometheus Operator holds an advantage over the other two solutions regarding the potential for increased support from other Prometheus-related projects.

This thesis is organized as follows: Section 2 provides background on key concepts related to Monitoring, Prometheus, and Operator. Section 3 outlines the design of the active monitoring Platform at Amadeus, including system design, data workflow, and system integrations. Section 4 discusses design criteria and proposed solutions, such as the employment of Prometheus Agent and the combination of load balancers, highlighting their value and design decisions. Section 5 details the implementation process, focusing on integrating the Prometheus Operator into the active monitoring Platform and employing the load balancer for probers. Section 6 presents and evaluates experiments, examining the efficiency and benefits of the proposed solution. Section 7 reviews related work in the field, comparing this thesis to other projects and highlighting distinctions. Section 8 summarizes the implementation and results, underscoring the thesis's contributions. Finally, Section 9 suggests potential avenues for future research, exploring promising areas for further investigation.

2 Background

2.1 Active Monitoring and Black Box Monitoring

Monitoring IT systems brings insight into systems in terms of observability, becoming increasingly critical to improving systems. Based on simulation data generated synthetically, active monitoring proactively monitors the performance of networks, applications, and infrastructure [Spl23]. Another well-known monitoring method is black box monitoring. Like active monitoring, black box monitoring evaluates a system's functionality based on its responses to given inputs while abstracting away the system's internal workings [JD21].

Active monitoring, also called synthetic monitoring, mainly identifies potential application issues, including health checks, Application Programming Interface (API) endpoint tests, etc. In addition, active monitoring also includes evaluating the performance of hardware resources and benchmarking the network performance [Spl23]. For example, the Quality of Service (QoS) requirements of the network like latency and bandwidth, the CPU utilization, and the memory utilization.

Black box monitoring is a well-established approach to system analysis, commonly applied in software testing and network monitoring. In contrast to white box monitoring, which baesd on metrics exposed by the internals of the system, it focuses on external behavior rather than internal metrics [Bey+16]. This technique metaphorically views the system as a "black box" in Figure 2.1, signifying that the internal mechanisms are not visible or accounted for in the evaluation process [Mye04].

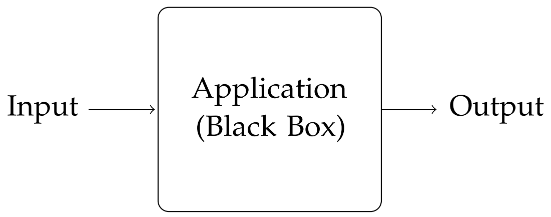

Figure 2.1: An example of a black-box monitoring.

Active monitoring and black box monitoring share some similar features but embody distinct concepts. Both involve simulating requests to target APIs or other endpoints, such as those of Transmission Control Protocol (TCP), HTTP, or other customized protocols. Specifically, These symptom-oriented techniques mimic user behavior, thereby reflecting real issues [Bey+16]. By definition, active monitoring can encompass black box monitoring but focuses on different aspects. Active monitoring emphasizes proactive check-ups, actively testing and querying systems, while black box monitoring is more of an observational approach, focusing on the outcomes and behaviors of systems without delving into their internal mechanics. In summary, despite of slight difference, both methods are valuable in system monitoring and could contribute to the comprehensive view of system health and performance.

In system monitoring, the "Four Golden Signals" are essential: Latency, Traffic, Errors, and Saturation. Latency signifies response times, differentiating between successful and failed requests. Traffic gauges demand, such as HTTP requests per second. Errors identify the rate of failures, whether explicit, implicit, or policy-driven. Saturation looks at resource utilization and potential bottlenecks [Bey+16]. Bridging these metrics, active monitoring proactively tests systems for performance and reliability, while black box monitoring observes system behavior without internal visibility. The combination enhanced the effectiveness of the "Four Golden Signals," ensuring comprehensive system monitoring and optimal system performance.

As the IT landscape becomes more complex with microservices and cloud-based applications, understanding the internals of every service can be overwhelming. In such a scenario, active monitoring and black box monitoring's ability to provide a holistic view of system behavior can be highly beneficial. Moreover, in light of growing privacy regulations and data protection measures, the non-invasive approach aligns with the current trends. Simulating test inputs and focusing on system outputs rather than internals could minimize privacy or data protection concerns.

In conclusion, active monitoring and black box monitoring are vital tools for system testing and monitoring. Its future appears increasingly integrated with cloud technologies and aligned with user-centric design principles. Nevertheless, they should not be considered a stand-alone solution but rather a component of a comprehensive monitoring strategy that incorporates various techniques to manage system performance effectively.

2.2 Prometheus Agent and Blackbox Exporter

The renowned Prometheus ecosystem provides two important components: the Prometheus Agent and the Blackbox Exporter. These technologies are extensively employed in various enterprises for monitoring applications in testing and production environments. Their widespread adoption is a testament to the reliability and robustness of the Prometheus platform in handling diverse monitoring needs.

The Prometheus Agent is a specialized mode of a standard Prometheus instance aimed at efficient data scraping and remote write as in Figure 2.2. Unlike a full-fledged Prometheus server with functionalities like data storage and querying, the Prometheus Agent is simplified for specific tasks, making it an ideal choice in distributed systems where resource optimization is crucial. This mode is invaluable when a full Prometheus server setup is unnecessary, facilitating a more resource-efficient deployment. It is particularly effective in large-scale environments where managing the overhead and complexity of multiple complete Prometheus instances would be impractical [Prob].

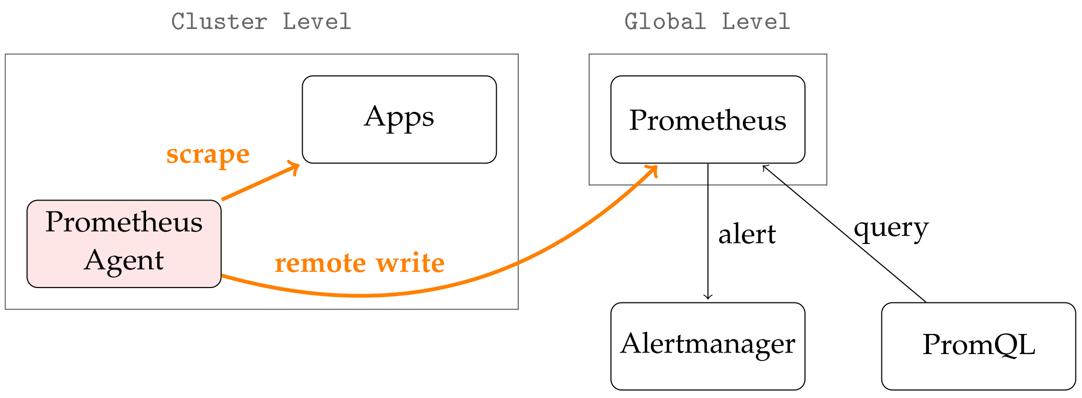

Figure 2.2: Prometheus Agent with scrape and remote write.

The Blackbox Exporter, an essential element of the Prometheus toolkit, is designed to probe external endpoints across multiple protocols, including HTTP/HTTPS, TCP, and ICMP. This capability is fundamental to black box monitoring, enabling the assessment of system health and performance from an external viewpoint without necessitating internal access to the monitored system [Pro23]. The Blackbox Exporter is instrumental in situations where internal monitoring is either not feasible or insufficient, such as in third-party services or in environments where internal metrics are not available or reliable.

Integrating the Prometheus Agent with the Blackbox Exporter fosters a distributed scheduler-executor architecture. This approach allows for a more granular and distributed monitoring framework, which is highly adaptable and can be effectively implemented across various Platform as a Service (PaaS) clusters with latencies [Proc]. Such an architecture bolsters the scalability and resilience of the monitoring system, rendering it suitable for large-scale and intricate environments. This distributed nature enhances the system's ability to scale and ensures a more resilient monitoring setup capable of withstanding node failures and network partitions [Prob].

Achieving optimal availability and scalability with the Prometheus Agent and Blackbox Exporter needs additional configuration and management. For the Prometheus Agent, employing relabeling is a crucial horizontal scaling or sharding strategy [Proa]. This method distributes the workload across multiple Prometheus Agent instances with a hidden label with hashed value, augmenting the system's capacity to handle substantial data volumes efficiently. Regarding the Blackbox Exporter, scalability is further enhanced by integrating a load balancer with properly configured scaling parameters, ensuring an even distribution of the probing load and maintaining system performance, even under high demand [Prob].

Overall, the active and extensive community surrounding Prometheus plays a significant role in the continuous improvement and reliability of the Prometheus Agent and Blackbox Exporter. This support ensures consistent updates and maintenance, effectively addressing the evolving needs and challenges in the monitoring domain. However, utilizing the Prometheus Agent and Blackbox Exporter to achieve scalability and manageability requires additional configurations and operations. Consequently, automating the scaling and management operations for both the Prometheus Agent and Blackbox Exporter emerges as a critical issue.

2.3 Operator Pattern and Prometheus Operator

The Operator Pattern, defined by the Cloud Native Computing Foundation (CNCF), represents a paradigm shift in Kubernetes, enabling users to automate the maintenance and configuration of applications [Kub]. The Prometheus Operator embodies this pattern, serving as an implementation that addresses the practical needs of deploying and managing Prometheus components. Through reconciliation, the Prometheus Operator aligns the deployment's actual state with the user's desired state, streamlining its lifecycle in Kubernetes [Ope20].

The Operator Pattern extends Kubernetes' native capabilities by introducing the Custom Resource (CR) and controllers. The operator, that is, the controller, is the essential software extension that utilizes the Kubernetes control plane and API to create, configure, and manage instances of complex stateful applications on behalf of a Kubernetes user [DW]. As shown in Figure 2.3, the controller continuously reconciles the current state with the desired state, encapsulated in custom resources, using a loop to transition watched objects to their target state. In conclusion, the Operator Pattern encapsulates operational knowledge, automating the management tasks typically requiring manual operations [CNC].

The Prometheus Operator is tailored to simplify the deployment and management of Prometheus within Kubernetes environments. It enables Kubernetes-native deployment

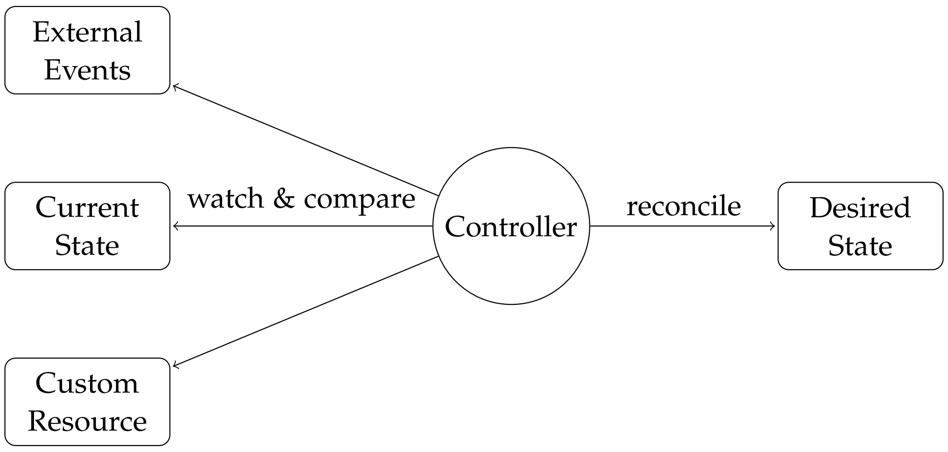

Figure 2.3: Illustration of the Operator Pattern.

and automated management of Prometheus and its associated monitoring components. Key features of the operator include Kubernetes CRs for deploying and managing Prometheus, Alertmanager, and so on, streamlined deployment configurations for setting up Prometheus essentials like versions and retention policies, and automatic generation of monitoring target configurations. This approach facilitates easy installation and version upgrades, simplifies configuration management, and ensures seamless integration with existing Kubernetes resources [Ope20].

In terms of current advancements in black box monitoring, the Prometheus Operator supports two CRs: Probe and PrometheusAgent. The Probe resource defines monitoring for a set of static targets or ingresses, such as specifying the target for the Blackbox Exporter and the module to be used. On the other hand, PrometheusAgent is responsible for defining a Prometheus agent deployment. This includes configurations like the number of replicas, ScrapeConfigs, and advanced features like sharding [Opeb]. Notably, it incorporates the ProbeSelector feature, which links to the previously defined Probe resource, enhancing its functionality and integration [Opea].

Despite these advancements, a significant gap persists in the management and configuration of the Blackbox Exporter within the Prometheus Operator framework. Specifically, there is no support for using a custom resource definition to manage the Blackbox Exporter. Consequently, users are required to deploy their own instances and modules of the Blackbox Exporter [Pro23]. This limitation underscores the need for more integrated solutions that can simplify the deployment and scaling of the Blackbox Exporter, ensuring it meets the evolving requirements of black box monitoring in complex environments.

In conclusion, while the Prometheus Operator has made significant strides in enhancing the ease and efficiency of deploying Prometheus in Kubernetes environments, there is still room for improvement, particularly in the realm of black box monitoring. Addressing the current limitations in the management of the Blackbox Exporter will be crucial in realizing the full potential of Prometheus as a comprehensive monitoring solution.

3 Overview

3.1 System Goals

This project aims to develop a scalable, reliable, and cloud-ready monitoring platform within the Prometheus ecosystem, designed for seamless integration into the Amadeus Observability Platform. To maintain compatibility and streamline integration with the existing platform, it will also support various probers based on custom protocols, including the Transport Cluster-Interconnect-Logic (TCIL) protocol used at Amadeus.

To ensure scalability, the platform should examine the bottleneck and consider the automated horizontal scaling among all components. Specifically, the design must keep all components stateless as much as possible. As for the inevitable stateful components, the maintenance of states should be handled properly for scaling. Meanwhile, distinct strategies like load balancing and sharding should be considered appropriately in reaction to different circumstances.

Reliability decides the trustworthiness of the Active Monitoring Platform, and the critical point to assure reliability is fault tolerance. To elaborate, as a prober deployed responsible for probing targets is down, another prober could take over and continue operating.

For cloud readiness, the design must be compatible with cloud infrastructure. Amadeus operates thousands of applications across numerous OpenShift clusters, necessitating that the Active Monitoring Platform be tailored for cloud platforms. Additionally, artifact versioning is crucial in cloud readiness, offering a centralized and managed system package for cloud deployment and the prerequisite for GitOps.

Overall, the proposed design addresses the demands for active monitoring among thousands of applications deployed and distributed in dozens of clusters. With a user-friendly and efficient solution, the existing Amadeus Observability Platform could seamlessly integrate the active monitoring results and achieve high availability and efficiency.

The current system for active monitoring features lightweight scheduling and probing with an operator managing configurations as CRDs. Substituting the scheduler with the Prometheus Agent is critical, improving efficiency with the remote write feature and decreasing peak requests by spreading requests over a scrape interval. Next, employing the Prometheus Operator breeds further enhancements, including deployment management, automatic scalability, configuration sharding, etc. In brief, implementing the above two aspects successfully meets Amadeus's active monitoring goals.

3.2 System Overview

The design proposal utilizes container orchestrators, including Kubernetes and OpenShift. Within Amadeus, the OpenShift Container Platform is predominantly employed, often set up as separate and autonomous clusters. The system design described in the upcoming paragraphs is illustrated in Figure 3.1.

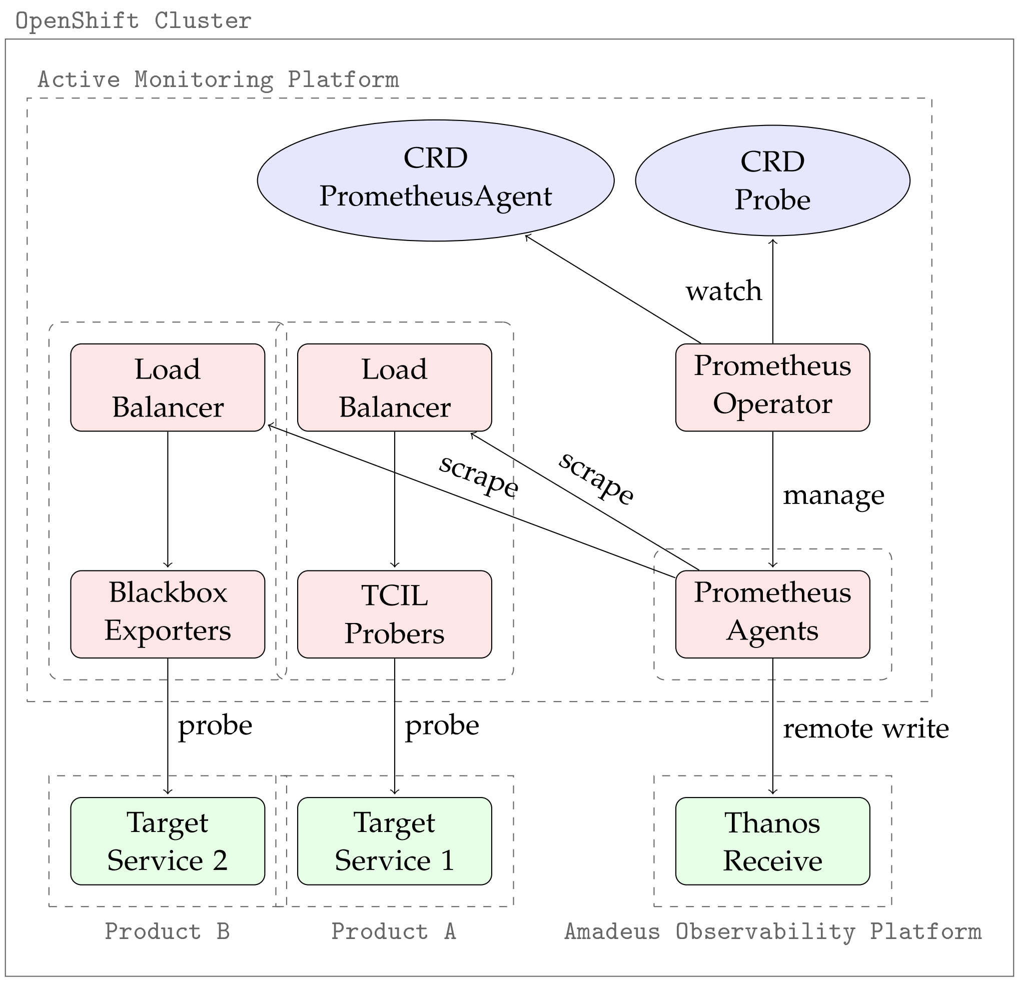

Figure 3.1: System Design with Prometheus Operator, Prometheus Agent and Probers.

Inside the cluster, a slightly modified Prometheus Operator would be installed, responsible for deploying and managing Promenteus Agents and probe configurations. Next, Blackbox Exporters and customized probers like the TCIL Pinger, which is still in development, would be deployed with load balancers, leveraging horizontal scalability for better load distribution.

At first, the deployed Prometheus Operator will monitor the CRDs created for deploying Prometheus Agents and probe configurations. It will maintain its state, offering scaling or sharding as required. Subsequently, the Prometheus Agent will handle scheduling probes and remotely transmit metrics to the Prometheus remote endpoint or Thanos Receive.

Secondly, deploying probers with the load balancer enables the Prometheus Agent to target the same host, leveraging load distribution and enhancing reliability without extra effort. Additionally, with Kubernetes or OpenShift, the inherent scalability and the ingress or load balancer provide valuable and reliable support, significantly boosting this design.

Finally, users with active monitoring requirements across various clusters and namespaces can leverage the official CRD: Probe to customize their specific requests. In summary, this system architecture offers a more efficient, seamlessly integrated, and highly available solution for active monitoring for the Amadeus Observability Platform.

3.3 System Workflow

Three workflows have to be discussed to illustrate this design's system workflow. They are responsible for maintaining Prometheus Agent, configuring probe settings, and scheduling probes.

Firstly, for the active monitoring platform's initialization, the admin must deploy Prometheus Agent via the official CRD: PrometheusAgent. As shown in Figure 3.2, the admin creates or updates the CR and receives the successful response. Then, the Prometheus Operator that keeps watching the CR retrieves the configuration and generates a new template to create or update the deployment of the Prometheus Agent.

Secondly, the workflow to create or modify probe configurations is similar to the above one, as they both leverage the Prometheus Operator for management and execution. In Figure 3.3, the user utilizes the CR: Probe, specifying the target Uniform Resource Locator (URL), scrape interval, and timeout interval, etc. As mentioned above, the Prometheus Operator would take over it, reloading the configuration of Prometheus Agent and bringing about the desired scheduling of probes.

Lastly, the third workflow is fully automatic, carrying out probes over different kinds of targets and operated by the Prometheus Agent as in Figure 3.4. With proper

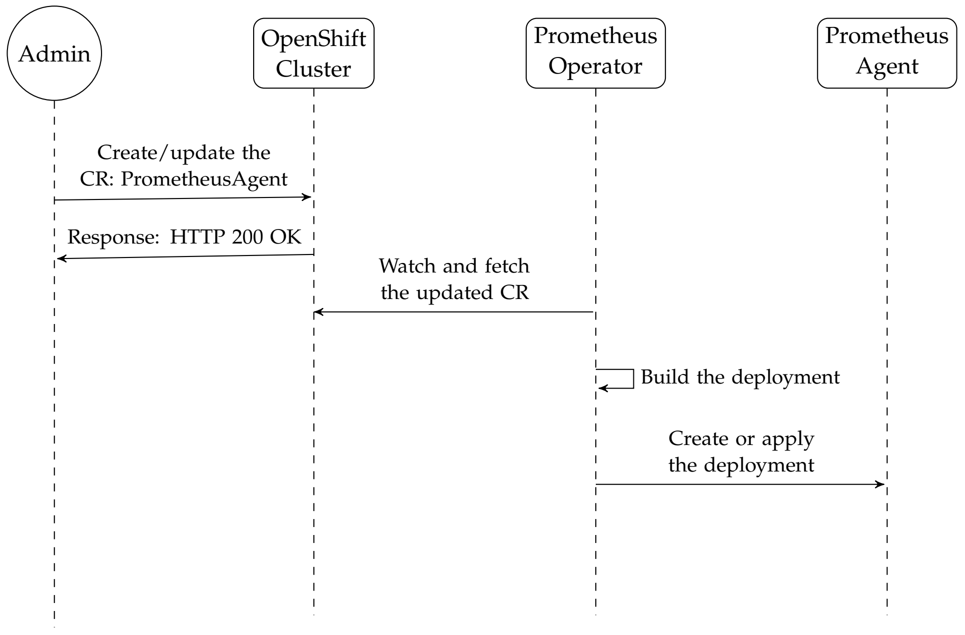

Figure 3.2: Deploying or updating Prometheus Agent by admin.

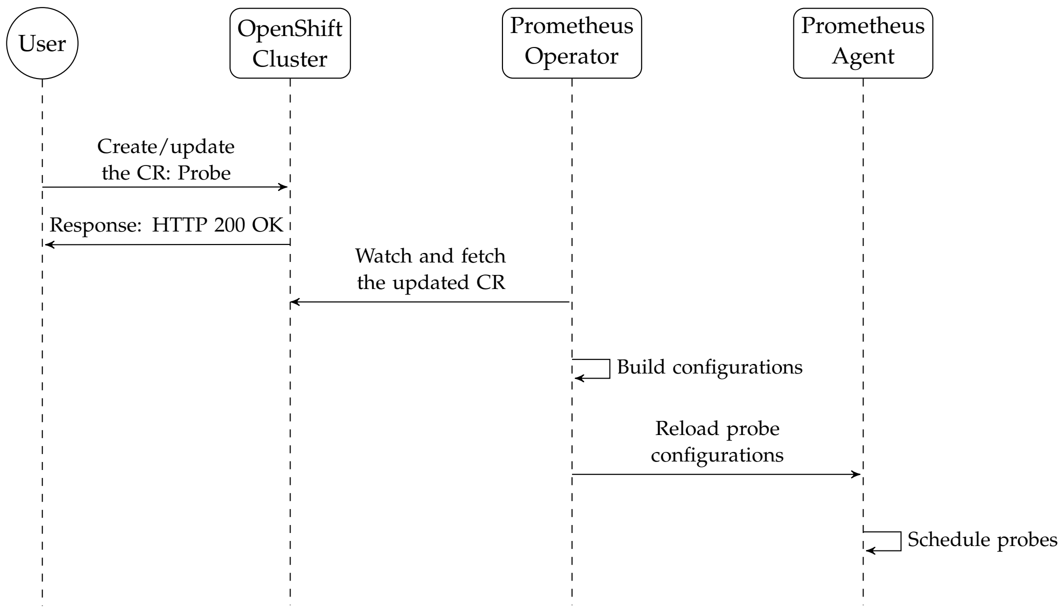

Figure 3.3: Creating or updating probe configurations by user.

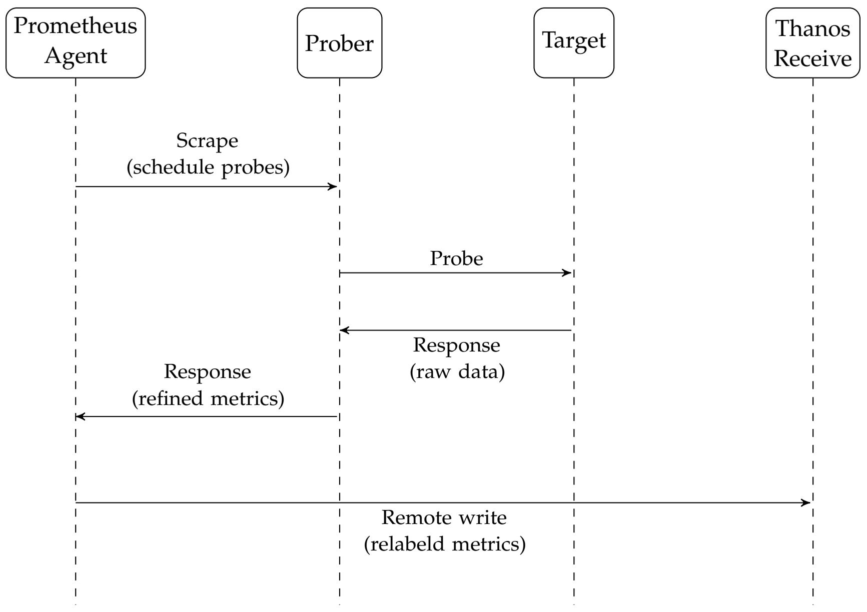

Figure 3.4: Scheduling probes by Prometheus Agent.

configurations of Probes, which the Prometheus Operator should load with the CR: Probe as mentioned in the previous paragraph, The Prometheus Agent would keep scraping the Probe with specified labels, parameters, modules, etc., and the Prober would probe targets accordingly, and then respond to the Prometheus Agent with collected raw information. Finally, after some operation on the metrics like relabeling or filtering, the Prometheus Agent sends these data to the Thanos Receive via remote write.

3.4 Integration with GitOps

So far, the design successfully addresses all requirements regarding active monitoring. However, Amadeus has plenty of clusters and namespaces, and numerous developers or users would deploy their own CR in their clusters and namespaces. So, managing deployed CRs becomes another crucial issue.

GitOps is a new approach to automate the management of the cloud infrastructure. Specifically, GitOps employs the Git repository, based on its modifications, triggering automatic reconfiguration and synchronization of the infrastructure as shown in Figure 3.5. Therefore, GitOps could also achieve Infrastructure as Code (IaC) by managing configurations in a Git repository.

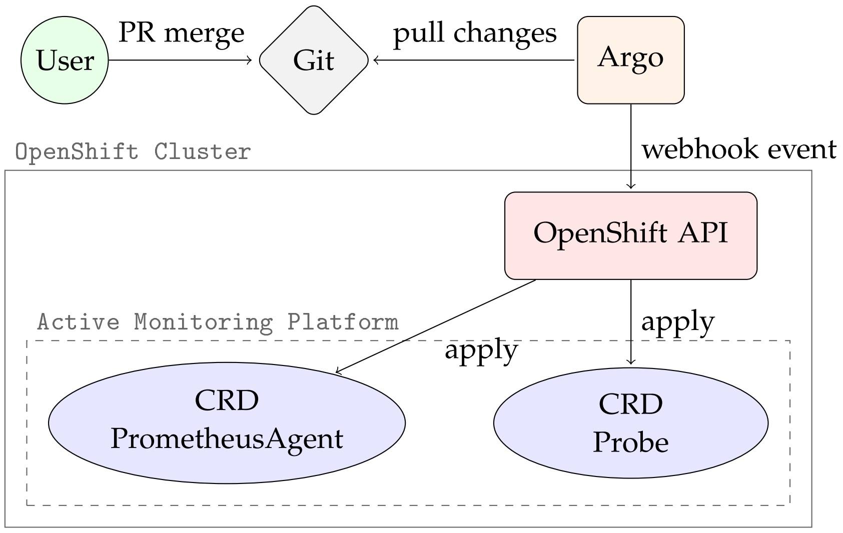

Figure 3.5: Utilizing GitOps to manage CRDs.

3.5 User Interface with Grafana

The implementations described above enable users to achieve scalable, reliable, and cloud-ready active monitoring. However, non-visualized results can be challenging for intuitive human understanding. Therefore, a vital final step involves using Grafana, an open-source analytics and visualization project, to create a dedicated dashboard for active monitoring. Since the Amadeus Observability Platform has already deployed the complete Grafana web application, no further deployment is necessary, apart from configuring the dashboard.

4 Design

4.1 Prometheus Operator

Prometheus Operator is an open-source tool designed to simplify the deployment, configuration, and management of Prometheus components in the Kubernetes ecosystem. Prometheus has grown in popularity due to its effectiveness in cloud monitoring, making it a cornerstone for developers focusing on reliability and performance. The Prometheus Operator, developed under the support of the CNCF, offers cloud deployment and management, aligning seamlessly with the principles and practices of cloud-native computing.

The primary goal of the Prometheus Operator is to make running Prometheus on the cloud platform as straightforward and efficient as possible. It leverages the container orchestrator's APIs to offer a scalable and highly available solution that fits the dynamic nature of cloud computing. The operator automates the complex processes of deploying, configuring, and managing Prometheus instances, making it easier for teams that don't have deep expertise in monitoring systems. By introducing CRDs representing distinct Prometheus components, such as Alertmanager, PrometheusAgent, and Prometheus itself, the integration lets users define their monitoring requirements with YAML configuration files declaratively.

Based on the orchestrator's API, one of the critical features of the Prometheus Operator is to dynamically discover and manage configurations, such as monitored targets. This dynamic management is crucial in environments where services and workloads constantly change. The operator automates updating Prometheus configurations in response to changes in the cluster, such as adding or removing pods, services, and endpoints. This ensures that monitoring is consistently aligned with the current state of the cluster, providing instant and accurate insights into applications and infrastructure.

In addition, to enhance the user experience, the Prometheus Operator also supports features like high availability, sharding, etc. To elaborate, for Prometheus and Alertmanager, it is ensured that monitoring and alerting systems remain operational even if individual components fail. For Prometheus Agent, the configuration file is divided into several configurations for multiple agent instances using the hash function, achieving Prometheus Agent sharding for distributing loads. Moreover, the operator also facilitates easy backup and restoration of Prometheus data, integrating with Kubernetes'

RBAC model to provide fine-grained access control over monitoring resources.

In conclusion, the Prometheus Operator enhances cloud monitoring by automating the deployment, configuration, and management of Prometheus. Its dynamic configuration adaptation, high availability, and sharding features increase efficiency and reliability. Therefore, utilizing the Prometheus Operator for robust cloud monitoring solutions in the Amadeus Observability Platform is valuable.

4.2 Prometheus Agent

The Prometheus Agent represents a simplified, focused approach to monitoring distributed systems, particularly in large-scale and cloud-native environments. As a lightweight variant of the Prometheus instance, the Prometheus Agent is designed to be a highly efficient data collector optimized for forwarding metrics to the Prometheus server or a compatible remote receiver.

One of the core advantages of the Prometheus Agent lies in its simplicity. Unlike an entire Prometheus server, which includes data storage, querying, and alerting functionalities, the agent is only responsible for scraping metrics and sending them out. This reduction leads to a smaller memory and CPU footprint, critical in resource-constrained environments, making it ideal for edge computing, microservices architectures, and multi-cluster environments.

In addition, the Prometheus Agent maintains excellent compatibility with the Prometheus ecosystem. This is a crucial consideration for organizations invested in Prometheus-based monitoring. It can scrape metrics in the same Prometheus exposition format, ensuring users integrate the agent seamlessly into the existing monitoring infrastructure. Furthermore, the agent's ability to integrate with service discovery mechanisms in orchestration platforms ensures its dynamically adapting to changes in the monitored environment.

Finally, the Prometheus Agent enhances the scalability and reliability of monitoring systems. The Prometheus Agent can achieve more efficient horizontal scaling depending on scrape configurations as shown in Figure 4.1 and focusing on scraping and forwarding metrics. This is particularly beneficial in large-scale environments, where a central Prometheus server can aggregate and process data from multiple agents, reducing duplication of storage and computation. Also, this architecture enhances the resilience of the monitoring system, as the failure of a single agent has a limited impact, ensuring reliable monitoring of the infrastructure.

Overall, the Prometheus Agent is an inevitable addition to the design of the active monitoring platform, offering a scalable and reliable solution for the existing monitoring system.

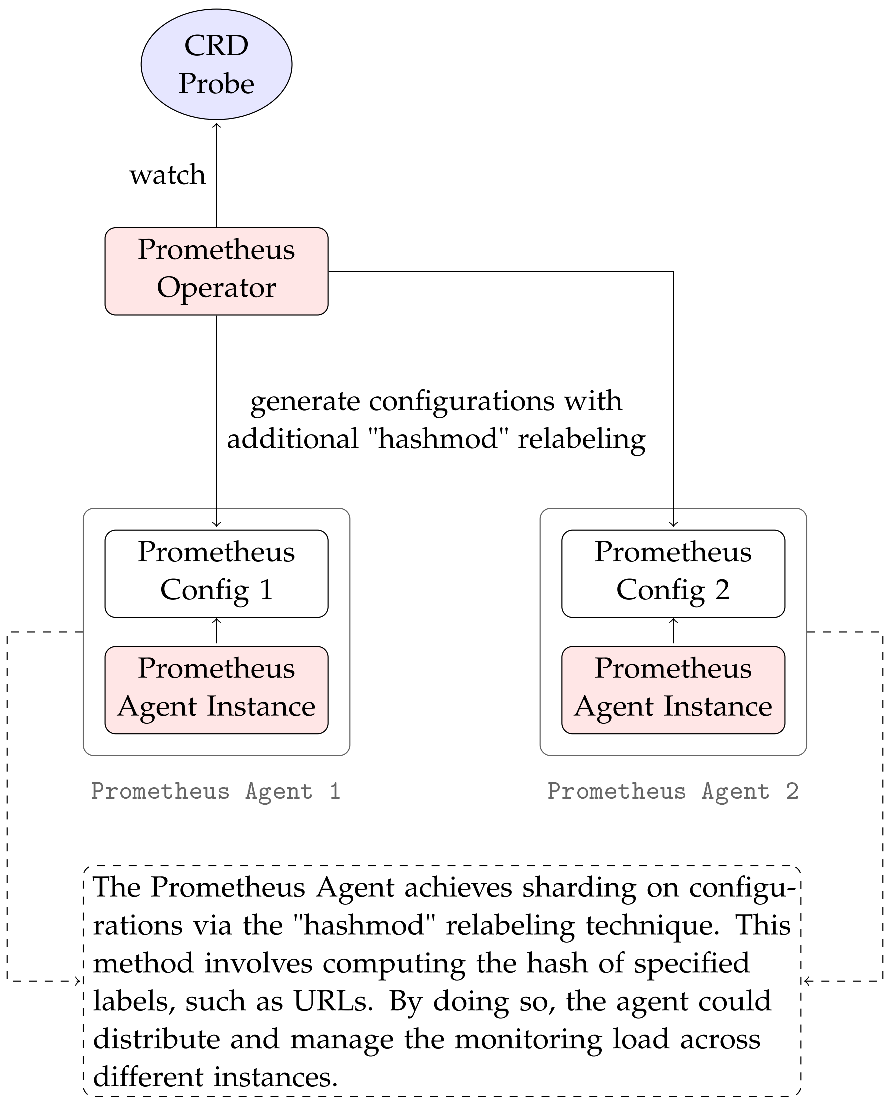
CRD
Probe
watch
Prometheus
Operator
generate configurations with
additional "hashmod" relabeling
Prometheus
Config 1
Prometheus
Agent Instance
Prometheus Agent 1
Prometheus
Config 2
Prometheus
Agent Instance
Prometheus Agent 2
The Prometheus Agent achieves sharding on configurations via the "hashmod" relabeling technique. This method involves computing the hash of specified labels, such as URLs. By doing so, the agent could distribute and manage the monitoring load across different instances.

Figure 4.1: Prometheus Agent Sharding Mechanism.

4.3 Blackbox Exporter and Load Balancing

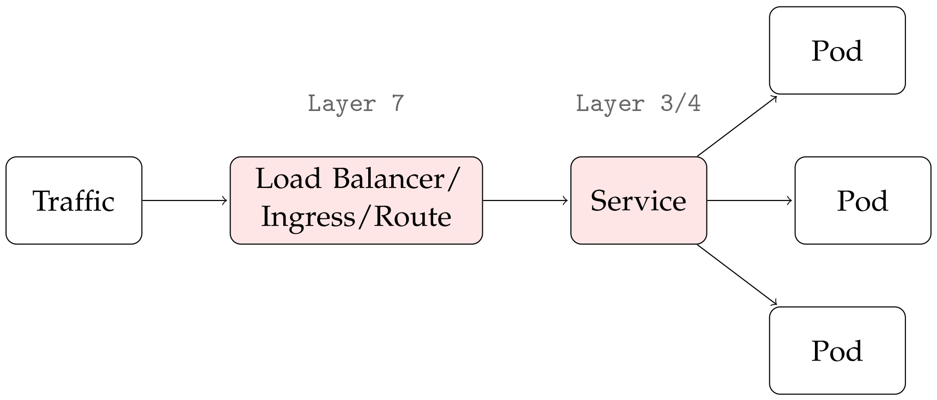

Figure 4.2: Route is the resource responsible for Load Balancing in Openshift.

Blackbox Exporter, a probing tool for the Prometheus monitoring system, is essential for assessing the performance and availability of services in network environments. It plays an important role in monitoring and diagnosing systems designed for probing endpoints over various protocols like HTTP/HTTPS, DNS, TCP, and ICMP.

In complex infrastructures hosting multiple services, it is crucial to integrate the Blackbox Exporter with Layer 7 load-balancing solutions such as built-in Load Balancers, OpenShift Routes, Kubernetes Ingress, or Layer 3 and 4 solutions like the Service, as depicted in Figure Figure 4.2. This integration enables scalable and efficient monitoring, ensuring the reliability of the monitoring system as the service count increases.

Moreover, combined with automatic horizontal scaling, resource utilization efficiency could be significantly enhanced. Load balancers optimize traffic flow to the Blackbox Exporter instances; the orchestrator scales the number of the Blackbox Exporter instances, ensuring that the monitoring process does not overuse or run out of resources.

Another benefit is the high availability and fault tolerance. If an exporter instance fails, the load balancer can redirect traffic to other operational instances, ensuring uninterrupted monitoring. On the other hand, in cloud-native environments where services often change, this setup supports dynamic service discovery natively, reducing the need for manual configuration and simplifying overall management.

In conclusion, integrating the Blackbox Exporter with load-balancing solutions enhances scalability and reliability in performance monitoring, simplifying management. Load balancing evenly distributes monitoring loads, leading to more accurate performance metrics and easier issue identification. This integration is essential for effective maintenance of network services, crucial in large-scale deployment and management.

5 Implementation

5.1 Deployment of the Prometheus Operator

This section aims to detail the preparations and procedures for the setup of the Prometheus Operator. As mentioned in previous sections, the Prometheus Operator takes responsibility for deploying and managing Prometheus Agents and probe configurations, enhancing users' experience in active monitoring. The following subsections outline and explain the modifications and deployment of the Prometheus Operator.

5.1.1 Context

Red Hat, Inc. has been actively promoting the integration of the Prometheus Operator into OpenShift. However, the most recent distribution hasn't incorporated the Prometheus Agent. Consequently, to avoid affecting the native deployment of the Prometheus Operator in OpenShift, a modified version of the operator, along with the associated CRDs, is required at this stage. This adaptation ensures compatibility with current OpenShift configurations and employs the Prometheus Agent's advanced monitoring features.

5.1.2 Rebuild the Prometheus Operator and CRDs

To deploy a distinct Prometheus Operator for specific use cases and to prevent conflicts in environments where the native Prometheus Operator is already in use, it's essential to modify the group name of the targeted CRDs. This process consists of implementing necessary adjustments and constructing a custom version. The steps involved are as follows:

1. **Edit the Group Name in the CRD Manifests:** Modify the "apiVersion" field in each CRD manifest by changing the group name part. This unique group name helps to distinguish the custom Prometheus Operator from the default installation.

2. **Update the Operator Code:** Besides the CRD manifests, updating references to the group name in the Prometheus Operator's codebase is necessary. This ensures that the operator correctly interacts with the modified CRDs.
3. **Build the Prometheus Operator Image with the Modified Group Name:** Once the changes are complete, build a new Docker image of the Prometheus Operator.
4. **Push the New Image to a Private Container Registry:** After building the new image, push it to your private container registry that the cloud platform can access.

These steps ensure that the modified Prometheus Operator functions independently of the native Openshift Prometheus Operator, thus allowing for customized configurations and deployments.

5.1.3 Deploy the Rebuilt Prometheus Operator

Deploying a modified Prometheus Operator requires careful steps to ensure that it integrates seamlessly and functions correctly. This process not only involves applying the modified CRDs but also deploying the Operator itself. The steps are as follows:

1. **Check the Targeted OpenShift Cluster:** Examine the targeted cluster to ensure it is ready for deployment. This includes checking for sufficient resources, access rights, etc.
2. **Apply the Modified CRDs:** Deploy modified CRDs to the cluster.
3. **Deploy the Prometheus Operator:** Use an OpenShift template or a Helm chart to deploy the Prometheus Operator. Ensure that the location of the rebuilt image is specified in the private container registry in the deployment configuration.
4. **Verify the Deployment:** Check the status of the Prometheus Operator to confirm the successful deployment by console or Command-line Interface (CLI) tools. It is essential to ensure that the Operator runs without errors and can manage Prometheus instances as expected.

These steps help identify potential issues with the modified Prometheus Operator and ensure its compatibility and functionality within the OpenShift ecosystem. By following these detailed steps, the deployment of the modified Prometheus Operator can be conducted smoothly.

5.2 Deployment of the Blackbox Exporter

As an extension for the Prometheus ecosystem, the Blackbox Exporter has to be deployed additionally as the terminal to raise probes. This section will introduce the setup and steps for the Blackbox Exporter.

5.2.1 Build the Blackbox Exporter Image with Necessary Modules

In Amadeus, two fundamental checks - the existence check and the certificate check - require the creation of specific modules for the Blackbox Exporter:

- *Existence Check Module:* Confirms network connectivity through a HTTP GET request to an endpoint, ensuring a successful response. Configurable parameters include the URL, anticipated HTTP status codes, and timeout settings.
- *Certificate Check Module:* Validate the existence and Transport Layer Security (TLS) certificates of HTTPS endpoints to ensure the presence, validity, and signing. Configuration includes the URL, expiry thresholds, and CA verification details.

These modules are used to build a customized Blackbox Exporter image, which is subsequently deployed in the environment. The construction process involves the following steps:

1. **Create the Configuration of Two Modules:** Create a YAML configuration file for the Blackbox Exporter that includes these two modules. For example:

```
modules:
  existence_check:
    prober: http
    timeout: 30s
    http:
      preferred_ip_protocol: "ip4"
  certificate_check:
    prober: http
    timeout: 30s
    http:
      fail_if_not_ssl: true
      tls_config:
        ca_file: "/certs/ca.crt"
      preferred_ip_protocol: "ip4"
```

2. **Build the Blackbox Exporter Image:** Once the configuration file is ready, incorporate it into the Blackbox Exporter Docker image for a new build. This involves creating a Dockerfile that uses the Blackbox Exporter base image, adds the configuration file, and sets necessary environment variables. Here is the Dockerfile example:

```
FROM prom/blackbox-exporter:v0.24.0
COPY blackbox-config.yml /etc/blackbox_exporter/config.yml
ADD certs /certs
EXPOSE 9115
ENTRYPOINT [ "/bin/blackbox_exporter" ]
CMD [ "--config.file=/etc/blackbox_exporter/config.yml" ]
```

3. **Push the Image to the Private Container Registry:** After building the image, push it to a private container registry for deployment. Ensure that the cluster is authorized to pull images from this private registry.

Following these steps, a customized Blackbox Exporter with specific modules for existence and certificate checks is created, aiming at the needs of Amadeus's infrastructure monitoring. This ensures targeted and efficient monitoring, specifically designed for the network and certificate validation requirements.

5.2.2 Deploy the Blackbox Exporter

Deploying the customized Blackbox Exporter is straightforward, with the image already prepared with the necessary configurations. To ensure a successful deployment with the monitoring platform, follow these steps:

1. **Deploy the Blackbox Exporter:** Depending on the preference and the existing setup, the deployment can be done using either an OpenShift template or a Helm chart.

2. **Verify the Deployment:** After the deployment, ensure it functions correctly by checking its status via console or CLI. Necessary checks include the pod's running status, resource utilization, log outputs for errors, and connectivity to targets. This step is essential to verify the operational readiness for monitoring.

By following these steps, the deployment of the Blackbox Exporter can be ensured to provide reliable monitoring capabilities for the cluster.

5.2.3 Load Balancing via the Service

In container orchestrators, Service for load balancing is optimal when Layer 7 functionalities are unnecessary and reducing load overhead is essential. The Service functions as a layer above the pods, creating a consolidated access point that distributes traffic evenly among different Blackbox Exporter pod instances. This method leverages kube-proxy's iptables mode for efficient traffic distribution. The following steps guide setting up load balancing with the Service:

1. **Create the Service Resource for the Blackbox Exporter:** Define a Service resource for the Blackbox Exporter. This Service will route traffic to the selected pods.

2. **Check the Connectivity of the Service Resource:** Once the Service is created, it gets assigned a ClusterIP and a DNS record within the cluster. Check the connectivity to this DNS record to ensure that the Service is accessible. You can do this by running a DNS lookup or a simple curl command from a pod within the same cluster:

   ```
   $ nslookup blackbox-exporter-service
   $ curl http://blackbox-exporter-service/
   ```

The created Service of the Blackbox Exporter serves as a central gateway for Prometheus Agent requests. The iptables proxy mode of the kube-proxy naturally directs traffic to available pods in a random and balanced way. This configuration ensures efficient and direct routing for monitoring traffic.

5.2.4 Scalability via the Horizontal Pod Autoscaler (HPA)

The HPA boosts elasticity and cost efficiency by enabling dynamic adjustment of pod replicas in response to varying workloads, thus enhancing application scalability. It scales the number of pod replicas based on metrics like CPU usage. This section explains how to implement an HPA for the Blackbox Exporter, allowing it to scale dynamically according to workload changes:

1. **Ensure the Metrics Server:** The HPA requires metric data to make scaling decisions. Ensure that the Metrics Server responsible for collecting resource usage data is deployed in the cluster.

2. **Create the HPA Resource:** Define an HPA resource in YAML format. Specify the deployment to scale, the metrics to be used for decisions, and the desired target values for those metrics.

3. **Apply the HPA Configuration:** Apply the HPA configuration to your Kubernetes cluster.

4. **Adjust HPA Parameters:** Based on the observed performance, you might need to adjust the HPA parameters, such as the target CPU utilization or the maximum number of replicas. Update the HPA configuration file and reapply it as necessary.

Implementing an HPA for the Blackbox Exporter ensures that the deployment can adapt to changing demands automatically, maintaining optimal performance and resource utilization.

5.3 Deployment of the Prometheus Agent

To utilize the Prometheus Agent, in addition to deploying the instance, it's critical to set up the scraping metrics from targets depending on the configuration file. With the management by the Prometheus Operator, the efforts of deploying, loading, and reloading the instance could be released, greatly easing the maintenance complexity. In this section, utilizing the Prometheus Operator to deploy and maintain the Prometheus Agent will be discussed in detail.

5.3.1 Configure the Prometheus Agent Configuration based on the PrometheusAgent CRD

The Prometheus Operator monitors the PrometheusAgent CRD, which is derived from the Prometheus configuration. The primary settings in this project are the "probeSelector" and "remoteWrite" fields, which are responsible for probe configurations and specifying the remote write endpoint, respectively.

The probeSelector functions as the Label Selector in this CRD, enabling users to define the labels of probes that will be selected by the Prometheus Agent. This feature allows the Prometheus Agent to distinguish the Probe used by the active monitoring platform from other monitoring systems, even though they might utilize the same official CRD.

The remoteWrite feature enables users to configure the remote endpoint to which the Prometheus Agent uploads metrics. By establishing the distributed Thanos Receive as the remote write endpoint in each local cluster, this architecture spreads some computational tasks to the edge, enhancing efficiency and availability.

5.3.2 Deploy the Prometheus Agent Configuration based on the PrometheusAgent CRD

Deploying a custom object from the PrometheusAgent CRD, which is monitored by the Prometheus Operator involves several steps to ensure successful integration and configuration. This process can be broken down into the following stages:

1. **Define a Custom Object from the PrometheusAgent CRD:** Create a custom object in YAML format with the necessary specifications of the PrometheusAgent CRD, including "probeSelector" and "remoteWrite" configurations. Here is an example:

```
apiVersion: monitoring.coreos.com/v1
kind: PrometheusAgent
metadata:
  name: example-prometheus-agent
  namespace: monitoring
spec:
  probeSelector:
    matchLabels:
    apps: active-monitoring
  remoteWrite:
    - url: http://thanos-receive.monitoring.svc.cluster.local:19291/api/v1/receive
```

2. **Apply a Custom Object:** Deploy the Prometheus Agent by applying the custom object in the cluster.

3. **Verify the Deployment:** Once the custom object is deployed, verify its status and ensure that the Prometheus Operator deploys the Prometheus Agent.

Following these steps, the Prometheus Agent is successfully deployed and configured in the cluster, leveraging the Prometheus Operator for efficient management.

5.4 Create the Monitoring Configuration

As of now, the preliminary settings have been completed, and users can utilize the Probe CRD to create an active monitoring configuration. This section will provide detailed instructions on configuring it, followed by the necessary steps.

5.4.1 Configure the Monitoring Configuration Based on the Probe CRD

With the Probe CRD, users can conveniently define monitoring for a set of targets, facilitating cloud-native configuration management within the cloud platform. The module, prober, and targets are the three most important fields for the configuration.

The "module" field in Prometheus configures how targets are probed, often using settings from the Blackbox exporter. The "prober" field specifies the prober details, with the essential "prober.url" parameter. "targets" defines a set of static or dynamic targets for the prober to monitor. Also, the "metadata.labels" needs to be set in coherent with the "probeSelector.matchLabels" in PrometheusAgent.

5.4.2 Deploy the Monitoring Configuration Based on the Probe CRD

Deploying the custom object from the Probe CRD is the last crucial step in active monitoring using Prometheus. It involves creating and applying the monitor configuration, ensuring that the specified targets are actively monitored based on the defined parameters. Here are the steps for the deployment:

1. **Define a Custom Object from the Probe CRD:** Create a custom object in YAML format to establish the desired monitoring configurations. Ensure that this object includes all essential fields, such as module, prober, and targets. Additionally, confirm that the metadata.labels within the object match the probeSelector.matchLabels specified in the previous section. For example, please refer to the provided code below.

```
apiVersion: monitoring.ruup.amadeus.net/v1
kind: Probe
metadata:
  labels:
    apps: active-monitoring
  name: probe-example
spec:
  interval: 15s
  jobName: probe-example
  module: existence_check
  prober:
    path: /probe
    scheme: http
    url: blackbox-exporter-service:9115
  targets:
    staticConfig:
      static:
        - 'http://example.com/'
```

2. **Apply a Custom Object:** Once the custom object is ready, apply it to the targeted cluster.

3. **Verify the Deployment:** After applying the object, verify that it has been successfully deployed.

4. **Check the Monitoring Results:** Once the object is deployed, the Prometheus Agent will begin monitoring the specified targets according to the intervals and parameters set in the configuration. The monitoring data and metrics can be viewed on the dashboards of the Prometheus Agent or the Blackbox Exporter.

After these steps, the custom object based on the Probe CRD is deployed, enabling active monitoring of specified targets using Prometheus Agent. This setup facilitates efficient and dynamic monitoring in cloud-native platforms.

5.5 GitOps for managing CRDs

GitOps can automate and manage infrastructure configurations using Git as a single source. In this section, we realize GitOps principles to manage CRDs using, the Argo CD, a declarative continuous delivery tool for GitOps.

5.5.1 Create Git Repository for CRDs

1. **Initialize a Git Repository:** Create a new Git repository, which will host all the CRD manifests and related configurations. Users could use GitHub, GitLab, or any other Git hosting service for this purpose.

2. **Add CRD Manifests:** Upload CRD manifests into the repository with a well-set structure and directory. Ensure each CRD manifest is written in YAML format and correctly structured.

3. **Version Control:** Adopt the branching strategy to maintain different versions or environments. Then, commit and push changes to the remote repository.

5.5.2 Register the Repository in Argo CD

1. **Add the Repository with Argo CD UI:** In the Argo CD dashboard, navigate to the settings and add your Git repository. Provide the necessary credentials if your repository is private.

2. **Configure Repository Settings:** Configure the synchronization settings, such as automated sync policies, sync frequency, and other repository-specific settings as required by your deployment strategy.

5.5.3 Deploy CRDs with Argo CD

1. **Create an Application in Argo CD:** In Argo CD, "Application" represents a set of resources to be deployed. Create a new Application and link it to the directory in your Git repository where the CRD manifests are stored.

2. **Define Sync Policy:** Choose an appropriate synchronization policy for your CRDs. You can opt for manual sync, which requires manual intervention to apply changes, or automatic sync, where changes in the Git repository are automatically applied to the cluster.

3. **Version Management:** For any updates or changes in CRDs, update the manifests in the Git repository. Argo CD will synchronize these changes based on the configured sync policy, ensuring that your cluster state matches the desired state defined in Git.

By integrating GitOps practices for CRD management, organizations can achieve automated, reliable, and version-controlled deployment processes, aligning infrastructure and application states with complex configurations.

6 Evaluation

This evaluation study seeks to determine if the design leads to a tolerable overhead compared to the original in-house monitoring application that Amadeus developed. Both applications operate on a private OpenShift Cluster managed by the Amadeus Site Reliability Engineering (SRE) team. The primary metrics collected include Openshift Metrics for CPU, memory, and network, gathered using cAdvisor or the Network Metrics Daemon [Reda] [Redb]. Subsequent sections will outline the evaluation methodology and present the findings from the system analysis.

6.1 Methodology

The evaluation focuses on analyzing system behaviors concerning CPU utilization, memory utilization, and network traffic. These three entities are critical for developers to gain insights into system overhead and performance intuitively. Understanding how a system allocates and utilizes its resources, along with how it communicates internally and externally, provides a comprehensive view of its efficiency and scalability.

6.1.1 CPU Utilization

For CPU utilization analysis within a pod, the metric "container_cpu_usage_seconds_total" will be utilized. This metric measures the cumulative CPU time consumed by a container in seconds. The methodology involves:

- Collecting CPU usage data over time to understand the baseline and peak CPU utilization patterns.
- Analyzing the CPU consumption in correlation with different applications to identify any potential inefficiencies.
- Comparing CPU utilization across different applications to assess resource allocation effectiveness.

This analysis will help in understanding the CPU demands of the applications running in OpenShift Pods and how effectively CPU resources are utilized.

6.1.2 Memory Utilization

Memory utilization will be assessed using the "container_memory_working_set_bytes" metric, which provides the amount of memory actively used by a container, excluding unused pages. The evaluation methodology includes:

- Monitoring the working set memory to identify memory usage under different operational conditions.
- Investigating memory patterns to ensure that applications operate appropriately and efficiently use memory resources.
- Evaluating memory usage trends over time to evaluate resource requirements.

This evaluation aims to highlight memory utilization efficiency and potential areas for optimization within the OpenShift environment.

6.1.3 Network I/O

Network I/O will be analyzed through "container_network_receive_bytes_total" and "container_network_transmit_bytes_total" metrics, representing the total bytes received and transmitted by the container network interface, respectively. The methodology involves:

- Monitoring inbound and outbound network traffic to identify communication patterns.
- Assessing network traffic volume in relation to application activity.
- Analyzing network traffic trends to assess data flow efficiency.

By examining network I/O, we aim to understand the network performance and efficiency of applications running in OpenShift Pods, highlighting areas for network optimization.

6.2 System Analysis

Prometheus' and Amadeus's solutions feature a similar architecture: the scheduler triggers the prober to carry out monitoring tasks while the prober probes targets. The forthcoming analysis will span 12 hours, focusing on the average hourly values of each metric. Given that the target count is nearing 100, the following analysis will distinctly address both the scheduler and the prober, specifically within the scenario of probing 100 targets.

6.2.1 Scheduler Analysis

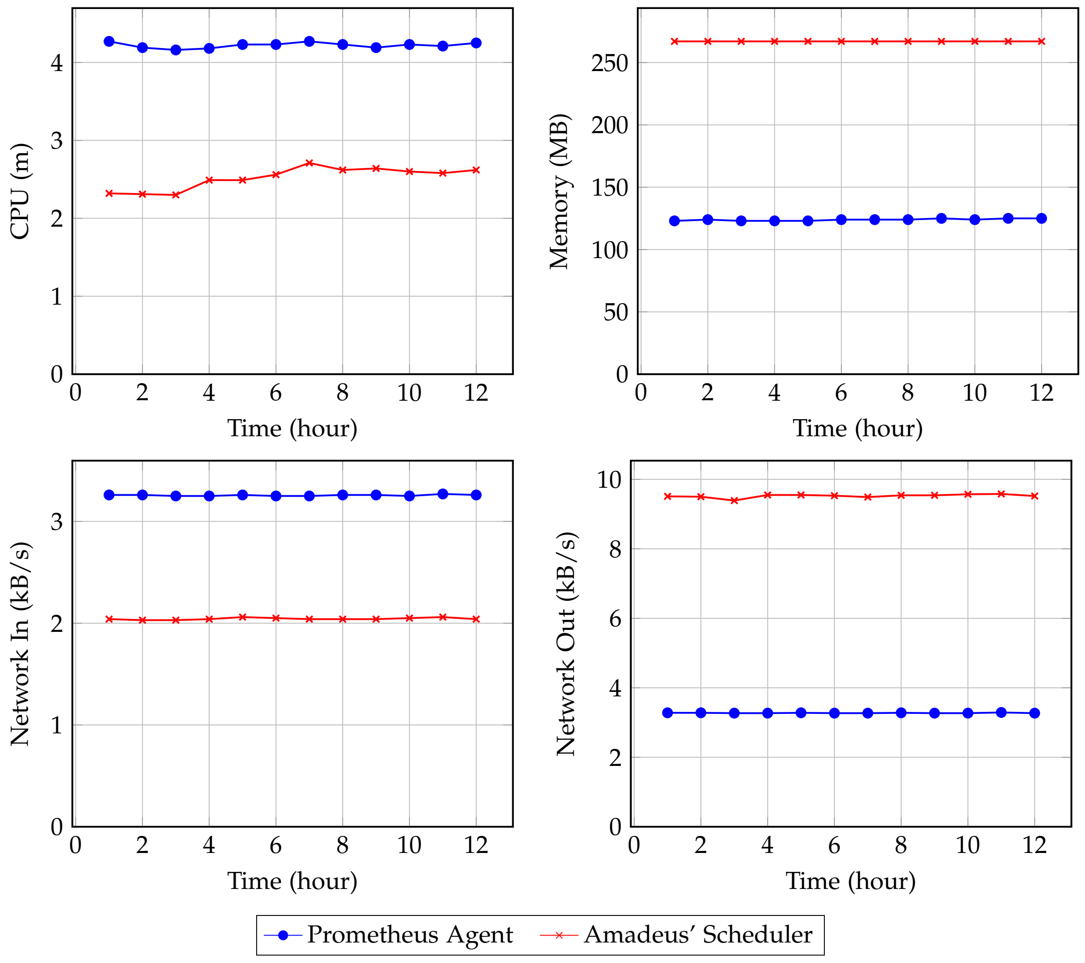

Figure 6.1: Scheduler analysis.

The Prometheus Agent and Amadeus' Scheduler perform similar roles but differ in their implementations, with the Prometheus Agent providing additional features. As depicted in the Figure 6.1, the CPU utilization of both schedulers is nearly identical, a result of the minor units in the measurement. Surprisingly, memory utilization by the Prometheus Agent was significantly reduced by approximately 140 MB. Regarding network I/O, given that the measurements are in kB, the differences are relatively inconsequential. In summary, adopting the Prometheus Agent results in an acceptable overhead while substantially decreasing memory consumption.

6.2.2 Prober Analysis

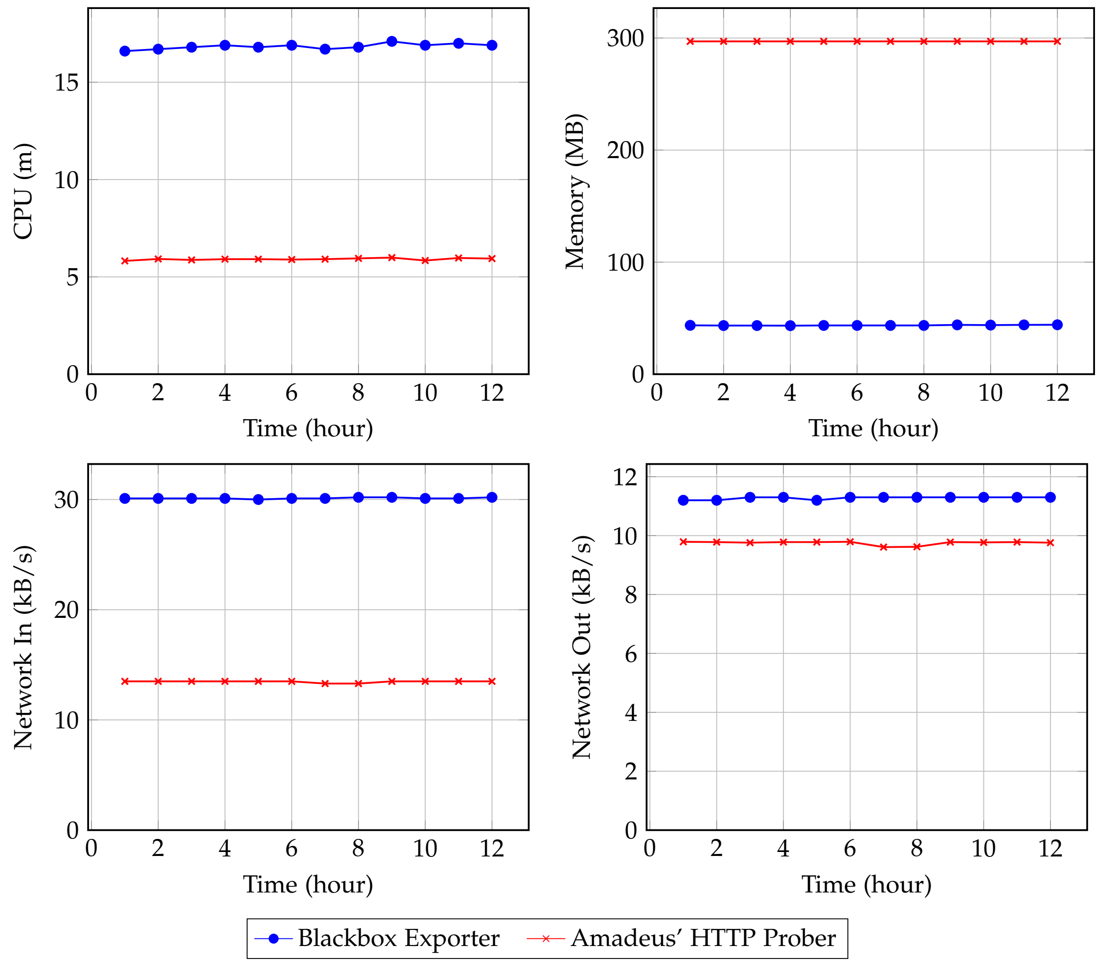

Figure 6.2: Prober analysis.

The Blackbox Exporter, capable of conducting customized probes across different protocols, contrasts with Amadeus' HTTP Prober's limitation to only HTTP probes. For a fair comparison, only the HTTP module is used in the Blackbox Exporter. Figure 6.2 shows the Blackbox Exporter's CPU usage is slightly higher than Amadeus' by about 10m, a minor increase justified by its extended features and considered acceptable. Notably, it also offers a substantial memory usage improvement of approximately 250 MB. Although its network input is double that of its counterpart, investigations reveal this is due to the Prometheus Agent's more detailed HTTP GET requests, which are not expected to cause issues, even when scaling up to 10 or 100 times the number of targets due to the lightweight nature of these requests. In summary, the minor increases in CPU and network input by the Blackbox Exporter are more than compensated for by considerable memory savings, making it a fully acceptable solution.

6.2.3 Scraping Analysis

This analysis evaluates the effectiveness of the Prometheus Agent's scraping strategy. Unlike Amadeus' Scheduler, which does not evenly distribute requests over time, the Prometheus Agent spreads scrapings uniformly across the time interval. Theoretically, this approach results in peak traffic at the start of each interval for Amadeus' Scheduler, while the Prometheus Agent achieves a more balanced load distribution. The subsequent figure presents the outcome of monitoring the Network In of the Prober—primarily influenced by the Scheduler's requests—over a 30-minute period:

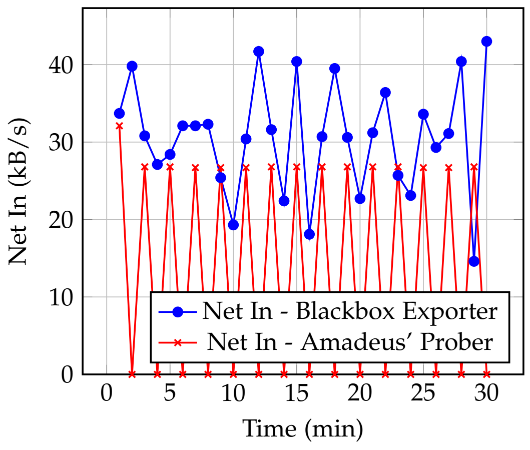

Figure 6.3: Scrape Analysis.

As demonstrated in the Figure 6.3, it is evident that the Blackbox Exporter experiences a more evenly distributed load of requests, in contrast to Amadeus' Prober, which exhibits saw-tooth traffic patterns. This observation suggests that the Prometheus Agent's scraping strategy results in more optimized and evenly balanced network traffic.

7 Related Work

Modern IT infrastructure management, monitoring, and analytics platforms ensure system reliability and performance. Some industries design and customize their solution, like Amadeus, while some employ mature existing commercial solutions. Among these solutions, as a representative player, Datadog is known for its robust functionalities and capabilities. Therefore, the upcoming section will introduce two solutions for active monitoring: the solution developed and used internally by Amadeus since 2021 and another solution driven by Datadog's synthetic monitoring [Datd].

7.1 Amadeus' Active Monitoring

Inspired by the Blackbox Monitoring principles, Amadeus developed the active monitoring system by integrating the Operator pattern, aiming for a simple cloud monitoring solution. As shown in Figure 7.1, this system is architecturally composed of three pivotal elements: the operator, the scheduler, and the prober. This design facilitates a lightweight yet practical approach to cloud-native monitoring.

Users initially tailor their monitoring setup using CR. The operator is key in interpreting these CRs to establish and maintain the scheduler's monitoring configuration. This approach highlights the system's flexibility, allowing users to define monitoring configurations that meet their specific needs.

The scheduler is central to the system's functionality, orchestrating the monitoring process. It is responsible for initiating monitoring requests, per the configurations detailed in the CRs, and coordinating with the probers to gather the requisite data. The probers, designed as workers, are tasked with executing the monitoring by probing the targeted services and directing the collected data back to the scheduler. This component-driven approach ensures a comprehensive monitoring cycle.

Despite its sound design, the system's limitations restrain its potency, particularly in large-scale applications. Notably, the architecture presents a single point of failure at the scheduler, posing risks of data collection interruptions and potential traffic bottlenecks. The lack of scalability poses these challenges, making it difficult to efficiently monitor a vast array of distributed services across multiple clusters.

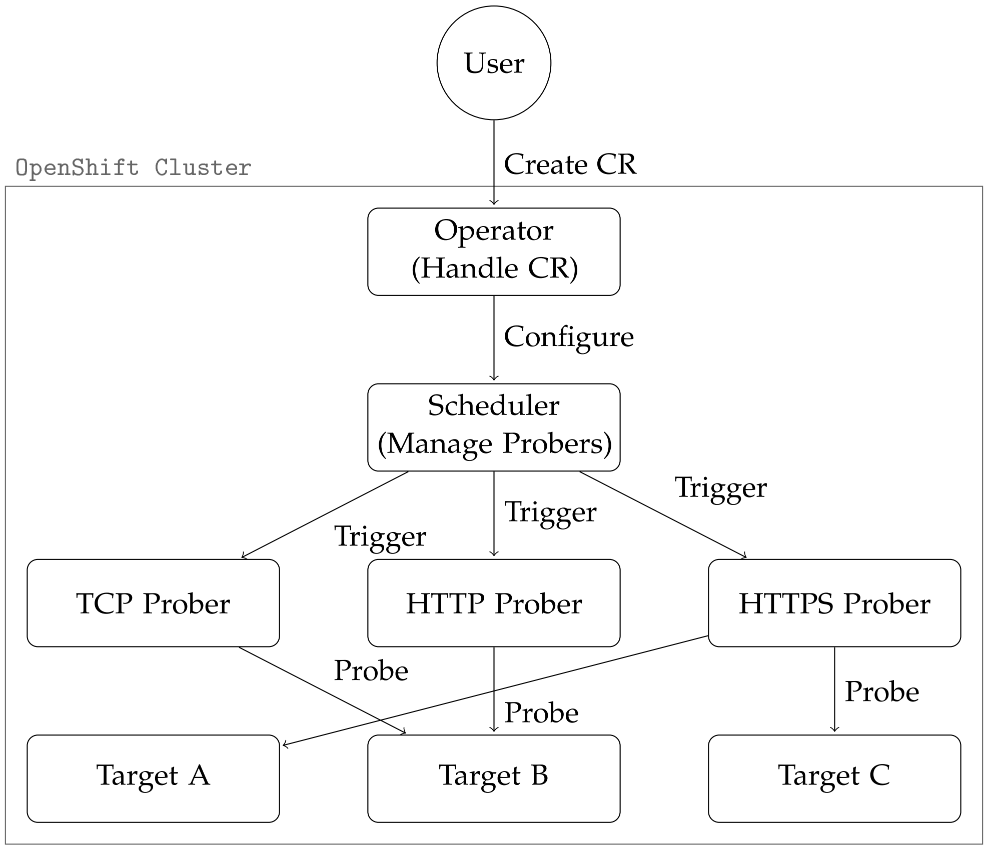

Figure 7.1: Amadeus' Active Monitoring.

7.2 Datadog's Synthetic Monitoring

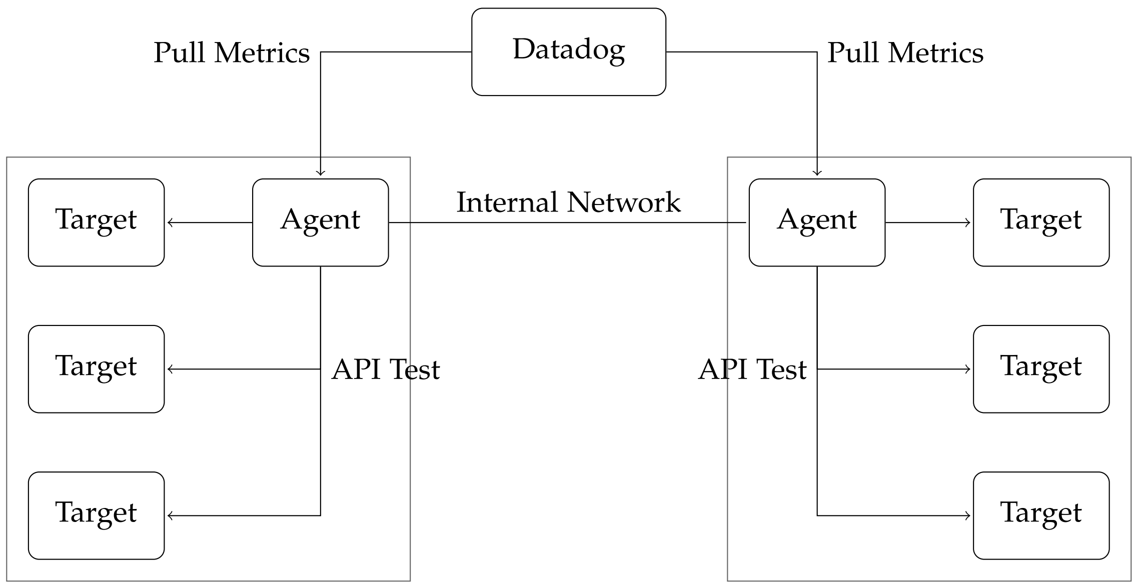

Figure 7.2: Datadog's Synthetic Monitoring.

As an integrated platform for monitoring, Datadog collects and analyzes data from multiple sources, offering a unified view of an organization's IT infrastructure. A standout feature of Datadog is the Synthetic Monitoring (see Figure 7.2), which is particularly applicable for active monitoring [Datd] [Data]. This feature simulates requests and actions, thereby offering insights into the performance of APIs across various network layers, from backend to frontend [Data] [Datb].

There are two distinct elements of Datadog's Synthetic Monitoring that are of particular interest: API Tests and Browser Tests [Data] [Datb]. API Tests actively probe target services, gathering their statuses and metrics. These tests offer a snapshot of the availability of the monitored services, a crucial factor in any active monitoring system. In Datadog's implementation, API Tests form the bulk of the monitoring process, thereby highlighting their importance. Browser Tests, on the other hand, execute defined scenarios on target services using a chosen browser. This essentially reproduces the actions of a user, providing experiential feedback about the services. This user-centric approach complements the more technical data gathered by the API Tests, contributing to a more holistic view of the system's health.

When it comes to monitoring availability, it is crucial to decide the position of source endpoints that proceed with monitoring [Datc]. Datadog provides the feature to customize private locations to execute Synthetic Monitoring. By installing Docker containers as private endpoints in desired locations, Synthetic Monitoring, including both API Tests and Browser Tests, can employ these private endpoints to compose its line of departure.

While Datadog's synthetic monitoring approach presents a well-rounded solution, it operates primarily within a centralized model in terms of the agents' side. To elaborate, monitoring targets from a private location requires an installed agent in the same cluster, and the agent could be a single point of failure.

8 Summary and Conclusion

8.1 Section

8.1.1 Subsection

9 Future Work

9.1 Section

9.1.1 Subsection

cross-cluster/namespace probe management
blackbox exporter CRD
Interface of custom body/module probe

Abbreviations

ACS Amadeus Cloud Service

API Application Programming Interface

CLI Command-line Interface

CNCF Cloud Native Computing Foundation

CPU Central Processing Unit

CR Custom Resource

CRD Custom Resource Definition

DNS Domain Name System

EDIFACT Electronic Data Interchange for Administration, Commerce and Transport

HPA Horizontal Pod Autoscaler

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IaC Infrastructure as Code

ICMP Internet Control Message Protocol

IT Information Technology

PaaS Platform as a Service

QoS Quality of Service

SNMP Simple Network Management Protocol

SOAP Simple Object Access Protocol

SRE Site Reliability Engineering

REST Representational State Transfer

TCIL Transport Cluster-Interconnect-Logic

TCP Transmission Control Protocol

TLS Transport Layer Security

URL Uniform Resource Locator

List of Figures

List of Tables

Bibliography

[Bey+16] B. Beyer, C. Jones, N. R. Murphy, and J. Petoff. *Site Reliability Engineering*. O'Reilly Media, Inc., Apr. 2016. ISBN: 978-1-4919-2911-7.

[CNC] CNCF. *CNCF Operator White Paper - Final Version*. GitHub. URL: `https://github.com/cncf/tag-app-delivery/blob/163962c4b1cd70d085107fc579e3e04c2e14d59c/operator-wg/whitepaper/Operator-WhitePaper_v1-0.md` (visited on 12/28/2023).

[Data] Datadog. *API Tests*. Datadog Infrastructure and Application Monitoring. URL: `https://docs.datadoghq.com/synthetics/api_tests/` (visited on 05/21/2023).

[Datb] Datadog. *Browser Tests*. Datadog Infrastructure and Application Monitoring. URL: `https://docs.datadoghq.com/synthetics/browser_tests/` (visited on 05/21/2023).

[Datc] Datadog. *Run Synthetic Tests from Private Locations*. Datadog Infrastructure and Application Monitoring. URL: `https://docs.datadoghq.com/synthetics/private_locations/` (visited on 05/21/2023).

[Datd] Datadog. *Synthetic Monitoring*. Datadog Infrastructure and Application Monitoring. URL: `https://docs.datadoghq.com/synthetics/` (visited on 05/21/2023).

[DW] J. Dobies and J. Wood. *Kubernetes Operators*. ISBN: 978-1-4920-4803-9.

[JD21] P. Jorgensen and B. DeVries. *Software Testing: A Craftsman's Approach*. May 31, 2021. 7-8. ISBN: 978-1-00-316844-7. DOI: `10.1201/9781003168447`.

[Kub] Kubernetes. *Operator Pattern*. Kubernetes. URL: `https://kubernetes.io/docs/concepts/extend-kubernetes/operator/` (visited on 12/28/2023).

[Mye04] G. J. Myers. *The Art of Software Testing*. Newark, UNITED STATES: Wiley, 2004. 9-11. ISBN: 978-1-280-34616-3.

[NVP21] F. Neves, R. Vilaça, and J. Pereira. "Detailed Black-Box Monitoring of Distributed Systems." In: *ACM SIGAPP Applied Computing Review* 21 (Mar. 1, 2021), pp. 24–36. DOI: `10.1145/3477133.3477135`.

[Opea] P. Operator. *API Reference*. Prometheus Operator. URL: https://prometheus-operator.dev/docs/operator/api/ (visited on 12/28/2023).

[Opeb] P. Operator. *Prometheus Agent Support*. GitHub. URL: https://github.com/prometheus-operator/prometheus-operator/blob/main/Documentation/designs/prometheus-agent.md (visited on 12/29/2023).

[Ope20] P. Operator. *Introduction*. Prometheus Operator. Oct. 6, 2020. URL: https://prometheus-operator.dev/docs/prologue/introduction/ (visited on 12/28/2023).

[Proa] Prometheus. *How Relabeling in Prometheus Works*. Grafana Labs. URL: https://grafana.com/blog/2022/03/21/how-relabeling-in-prometheus-works/ (visited on 12/27/2023).

[Prob] Prometheus. *Introducing Prometheus Agent Mode, an Efficient and Cloud-Native Way for Metric Forwarding | Prometheus*. URL: https://prometheus.io/blog/2021/11/16/agent/ (visited on 12/27/2023).

[Proc] Prometheus. *Understanding and Using the Multi-Target Exporter Pattern | Prometheus*. URL: https://prometheus.io/docs/guides/multi-target-exporter/ (visited on 12/27/2023).

[Pro23] Prometheus. *Blackbox Exporter*. Prometheus, May 26, 2023.

[Reda] RedHat. *Associating Secondary Interfaces Metrics to Network Attachments | Networking | OpenShift Container Platform 4.10*. URL: https://docs.openshift.com/container-platform/4.10/networking/associating-secondary-interfaces-metrics-to-network-attachments.html (visited on 02/19/2024).

[Redb] RedHat. *Prometheus Cluster Monitoring | Configuring Clusters | OpenShift Container Platform 3.11*. URL: https://docs.openshift.com/container-platform/3.11/install_config/prometheus_cluster_monitoring.html (visited on 02/19/2024).

[Spl23] Splunk. *Active vs. Passive Monitoring: What's The Difference?* Splunk-Blogs. Nov. 20, 2023. URL: https://www.splunk.com/en_us/blog/learn/active-vs-passive-monitoring.html (visited on 01/08/2024).