COLONIES - COMPUTE CONTINUUMS ACROSS PLATFORMS

TECHNICAL REPORT

Johan Kristiansson
Department of Computer Science
RISE Research Institutes of Sweden
Luleå, Sweden
`johan.kristiansson@ri.se`

Thomas Ohlson Timoudas
Department of Computer Science
RISE Research Institutes of Sweden
Luleå, Sweden
`thomas.ohlson.timoudas@ri.se`

Henrik Forsgren
Department of Computer Science
RISE Research Institutes of Sweden
Luleå, Sweden
`thomas.ohlson.timoudas@ri.se`

Erik Källman
Department of Computer Science
RISE Research Institutes of Sweden
Luleå, Sweden
`erik.kallman@ri.se`

April 7, 2023

ABSTRACT

Artificial intelligence and machine learning has gained significant traction in recent years. At the same time, development and operation of AI workloads has become increasingly challenging. One difficulty is the lack of portability, making it cumbersome to move from one platform to another. Creating and operating fully automated end-to-end workflows across devices, edge, and cloud platforms is even more challenging.

To address the aforementioned challenges, the paper introduces an open-source framework called Colonies[1], which is designed to facilitate execution of computational workloads across a diverse range of platforms. Colonies enables the development of a *grid-like distributed virtual computer*, called a *Colony*, which can effortlessly integrate with any third-party application or other workflow management systems.

Built upon a modular microservice architecture, Colonies enables complex workflows to be broken down into composable functions. With the use of an HTTP protocol, these composable functions can be implemented in any programming language and be executed by independent executors deployed across platforms, anywhere on the Internet, including cloud, edge, high-performance computing (HPC) environments, devices, or even web browsers. A zero-trust security protocol enables a collection of distributed executors to operate as a unified entity, thus establishing seamless computational continuums across multiple platforms.

In addition to a technical description of the Colonies framework, the paper also describes some potential use cases. The paper describe how Colonies can be leveraged to build a remote sensing platform on Kubernetes, serve as a building block for edge computing, implement a serverless FaaS (Function-as-a-Service), and how it can be integrated with the Slurm workload manager. Finally, the paper presents a performance investigation, as well as scalability and robustness evaluation.

In summary, Colonies is a highly versatile and scalable framework that can streamline development and deployment of computational workloads across heterogeneous platforms while ensuring full traceability and zero-trust security.

Keywords Distributed computing, serverless computing · parallel computing · Workflow orchestration

[1]https://github.com/colonyos/colonies

1 Introduction

Developing robust and scalable AI systems is a challenging task that requires deep understanding in several fields. To begin with, an AI model must be trained which requires knowledge in advanced statistics or machine learning. Typically, training and validation data must be pre-processed through various stages before it can be utilized. Although it may be practical for small-scale projects to run the entire training processes on local development computers, larger AI models typically require access to powerful compute clusters or even high-performance computing (HPC) systems. Manual use of such infrastructure can be laborious and time-consuming. Automating the training process enables faster iterations and quicker discovery of useful models.

Taking an AI model into production requires substantial software engineering expertise and collaboration across teams. In contrast to traditional IT workloads, both the data and the model must be managed in addition to the software itself. As most models require regular re-training or re-calibration, it must be possible to update deployed models and software seamlessly without losing information or breaking the system. In many cases, there is a constant flow of data ingested into the system which must be managed even in case of failures. This becomes even more challenging when nodes or parts of the underlying infrastructure become unavailable due to maintenance such as software updates, hardware replacements and sometimes missconfiguration problems.

In some cases, it may be necessary to scale the system to increase or reduce the capacity. This is especially critical when using expensive cloud resources. Scaling the system means that the underlying infrastructure may change at any time, causing instability issues for running services or workflows. Therefore, it must be possible to detect failed computations and reprocess failed tasks part of a larger workflow. Workflows must hence be designed to handle an ever-changing infrastructure, and if a failed computation cannot be restored gracefully, engineers must be able to quickly perform root cause analysis to manually recover the system.

In reality, AI system requires integration of multiple systems. For instance, data need to be captured from an IoT system or pulled from third-party database running on different domains than the compute cluster itself. With the emergence of edge computing, parts of a data pipeline may also run on edge servers to bring computations closer to data sources. Configuring and setting up such pipelines add even more complexity.

Additionally, many compute clusters operate on-premises installations. Sometimes it is necessary to temporarily increase the capacity of on-prem clusters by combining resources from multiple providers, for example, adding cloud compute resources to handle peak loads or utilize HPC resources to quickly reprocess historical data. Developing hybrid workflows where some jobs run in the cloud and others run on HPC systems requires even more software development efforts [1] and is beyond the scope of many users, preventing them from utilizing powerful hardware. Clearly, there is a need for a framework that can consolidate various workflow management platforms to simplify development and enable seamless execution across platforms.

This paper presents a framework called Colonies, specifically designed to implement a *distributed virtual computer system* that can easily integrate with any third-party system. Colonies is built around a loosely-coupled microservices architecture that separates workflow definitions from implementation and deployment. The primary objective is to establish a platform where monolithic workflows can be decomposed into independent microservices that can easily be integrated with other systems or applications running anywhere on the Internet. The remainder of the paper describes the Colonies framework and how it can be used to create robust and scalable AI systems across plattforms.

2 Related work

Workflow management has been extensively studied in both academic and industrial settings with numerous approaches [2, 3, 4, 5, 6, 7] proposed to address the challenges in this field. Recently, Apache Airflow [8] has become a popular open-source workflow management system for handling data engineering pipelines. Like Colonies, Apache Airflow enables developers to create custom operators and executors that can be integrated with various systems. Additionally, Apache Airflow offers an HTTP API that makes it possible to develop software development kits (SDKs) in various programming languages. However, Apache Airflow does not rely on a queuing system. Instead, it must be integrated with a message broker, such as RabbitMQ [9] or Kafka [10], to implement task queues, resulting in a more complex architecture than Colonies. Furthermore, Colonies is based on a distributed microservice architecture that makes it more suitable for DevOps software development. As Colonies is loosely coupled, executors can be implemented and dynamically deployed on the Internet without reconfiguring the workflow engine.

Argo [11] is an open source container-native workflow engine for orchestrating parallel jobs on Kubernetes. It is can be used for running CI/CD pipelines or compute intensive machine learning or data processing tasks where each job runs as a container. In contrast, Colonies offers a more versatile approach, allowing jobs to be launched within an already started container. As launching new containers on Kubernetes can occasionally be time-consuming, Colonies can delivers higher throughput as the costs of starting new jobs are minimal. This is particular useful when launching large container images, or workloads (e.g. Julia scripts) that can take long time to start.

Today, utilization of serverless computing is experiencing a significant growth [12], primarily due to its potential to liberate developers from the burden of managing underlying cloud infrastructures. Attempt are currently being made to implement serverless workflow management systems. For example, OpenWolf [13] is a serverless workflow engine designed to utilize the Function-as-a-Service (FaaS) paradigm for composing complex scientific workflows. It is based on OpenFaaS [14], which allows functions to run on Kubernetes clusters. The serverless workflow project [15] aims at providing a vendor-neutral workflow DSL. Synapse [16] is a workflow management system similar to Argo, but that implements the serverless workflow specification. A similar engine is proposed in [17]. Colonies can also be used to implement serverless workflow management systems. This will be further discussed in Section 4.1. However, in contrast to previous work, Colonies is plattform independent and does not require Kubernetes. By using a zero-trust security protocol, functions can be securely executed by distributed executor deployed anywhere on the Internet. It is important to point out that Colonies does not provide an infrastructure for function execution. Instead, Colonies primary role is to serve as a platform for coordinating function executions which are carried out by distributed executor.

Currently, microservices are primarily used to implement large-scale web applications or Internet applications requiring high-availability. It has not yet become a prevalent design principle for workload management or implementation of HPC applications. Instead, simple job scripts are commonly used. J. Represa et al. [4, 18] explore various challenges associated with developing microservice-based workflow management for industrial automation within the context of the Arrowhead project [19]. The authors conclude that microservice-based workflow technologies is viable for industrial applications, particularly due to their inherent flexibility. The primary contribution of this paper is a comprehensive technical description of how to implement a distributed workflow engine based on microservice principles, extending beyond orchestrating microservices to execute automation tasks within one platform. The vision is to create a distributed architecuture where microservices can reside anywhere on the Internet and still function as a cohesive unit to execute cross-platform workflows.

Grid computing [20] is a distributed computing model that allows multiple computers, which may be geographically dispersed, to collaborate in addressing large-scale computational challenges. The Colonies framework is founded upon a grid computing model with the primary purpose of assigning tasks to distributed executors. To the best of the authors' knowledge, no previous work has integrated a microservice-oriented architecture with a grid computing model to serve as an integration point for coordinating artificial intelligence workloads across a diverse range of platforms. Additionally, Colonies is designed to function as a ledger, offering complete transparency and execution history, which is essential for implementing zero-trust security.

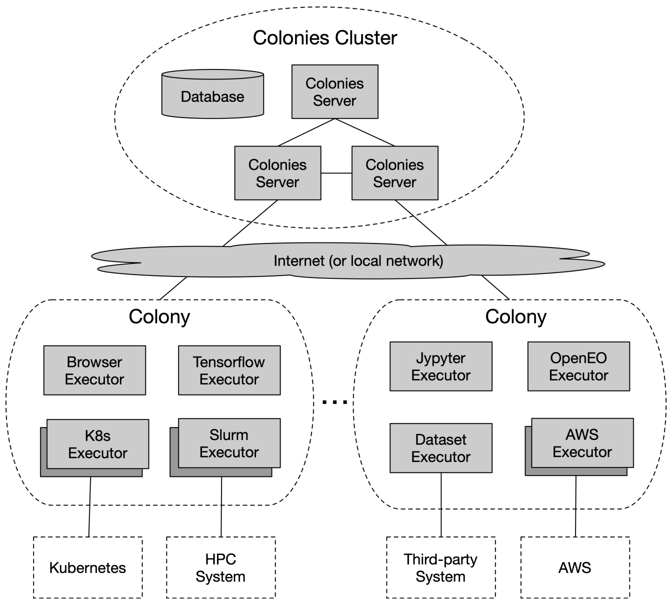


Figure 1: Overview of the Colonies framework. Executors may be deployed anywhere on the Internet.

3 The Colonies framework

A core concept of the Colonies framework is the notion of *processes*. A process contains meta-information about computations executed by a remote computer programs, which are referred to as *Distributed executors*. Specifically, a process contains a definition of a function to be executed, as well as contextual information such as execution status, including the result of the computation. One can think of a process as a digital twin of a real computing operation process. It is essential to note that a process does not necessarily have to be an operating system process; it can represent any type of computation, e.g. a remote procedure call executed by any kind of software.

Figure 1 depicts an overview of the components part of the Colonies framework. The *Colonies servers* form the backbone of the framework, functioning as a centralized control system for task submission and assignment. The Colonies server acts as a job broker for executors, almost like an employment agency for people. All execution information and history are stored in a database maintained by the Colonies servers. Upon submission, a process is stored in the database, serving as a queueing system. When a process is assigned to an executor its execution status is changed from *waiting* to *running*. Consequently, it is possible to submit a process that should run in the future even if no executors can run the process at the moment. In this way, Colonies supports both batch and real-time processing.

Distributed executors are responsible for executing processes assigned by the Colonies Servers. Several distributed executors form a *Colony*, which is a collection of executors operating as a cohesive unit where each executor is responsible for executing specific type of processes. One could view a Colony as an organization or a society of distributed computer softwares acting a single virtual computer system. To interact with other executors, executors must prove their Colony membership using a cryptographic protocol that follows the zero-trust security principle *never trust, always verify*. This security protocol ensures that users can keep control even when executors are distributed across plattforms. Zero-trust security is fundamental to the Colonies framework and will be further discussed in Section 3.4.5. The following sections outlines the underlying design principles and describes the Colonies framework in more details.

3.1 Microservices

Microservices is an architectural design pattern in which an application is structured as a collection of small, independently deployable, and loosely coupled services that communicate with other microservices through a well-defined API. By dividing the application into smaller, focused microservices, applications become easier to understand, maintain, and develop. In the Colonies framework, executors are microservices having the following characteristics:

- **Single responsibility:** Each executor is only accountable for executing specific functions. This makes the system easier to understand, develop, test and maintain.
- **Loosely coupled:** Executors are designed with minimal dependencies on other executors, enabling various software development teams to work independently. For instance, a data engineering team may handle the implementation of pre-processing functions, while a data science team oversees machine learning functions, and another team manages visualization or customer integration etc.
- **Scalability:** Executors can be deployed independently, which enables horizontal scaling. This allows for better resource utilization, parallelism, and improved performance.
- **Resilience:** The failure of a single executor does not compromise the entire application or workflow. Executors' isolation from one another contributes to a more resilient and fault-tolerant system. If an executor crashes during execution, the process is automatically reassigned to another executor.
- **DevOps and Continuous Integration:** Executors' inherent fault-tolerant design permits changes to individual executors without impacting the entire system. This makes it possible to seamlessly update the system.
- **Technology agnostic:** Executors can be implemented in any programming language, facilitating seamless integration with other applications and systems.
- **Decentralized governance:** As the Colonies framework is technology-agnostic, different software development teams can make independent technology and design decisions when developing executors, promoting greater flexibility and adaptability. For example, some executors may be implemented in Rust, while others may use Python to leverage state-of-the-art machine learning frameworks.

Although microservices has attractive properties and simplifes both development and deployment of executors, they also introduce complexity to the Colonies servers. To implement a workflow management framework supporting distributed microservices, the following challenges must be addressed:

- **Process management:** Colonies must be able to distribute processes among available executors based on their capabilities and current workload. This involves assigning processes to the most appropriate executor and then monitor process progress.
- **Fault tolerance and recovery:** In the event of executor failures, the Colonies framework must be able to re-assign processes to other executors, manage executor restarts, or trigger recovery mechanisms to maintain system reliability and resilience. In addition, the Colonies framework must be able to handle restarts or crashes of Colonies servers, thereby preventing any internal states in Colonies from becoming corrupted.
- **Load balancing:** The Colonies framework must manage load balancing among executors to optimize performance and avoid overloading any single executor.
- **Service discovery:** The Colonies framework must enable executors to dynamically register and deregister to Colonies servers. It must be possible to deploy executors anywhere on the Internet, even behind firewalls.
- **Workflow orchestration:** The Colonies framework must coordinate and orchestrate complex multi-step workflows executed by several executors, sometimes in parallel where executors run on different plattforms. Colonies must define the sequence in which processes should be executed, manages dependencies among processes, and ensure that workflows execute successfully to completion.
- **Monitoring and debugging:** The Colonies framework must monitor the overall system, allowing administrators to track the health, performance, and resource utilization of the executors and the system as a whole.

The Colonies framework uses a combination of technologies to address the aforementioned challenge. A fundamental design principle is stateless web serices, which makes Colonies simpler, more reliable and scalable, and easier to implement. To ensure reliability and data consistency, distributed consensus algorithms are needed to offer high-availability and handle server crashes. The remainder of this section describes how the Colonies framework is implemented. The next discusses the role of queues to support batch jobs and realtime processing, but also as a mechanism to enable loose coupling of system components.

3.2 The role of queues as seperation of concerns

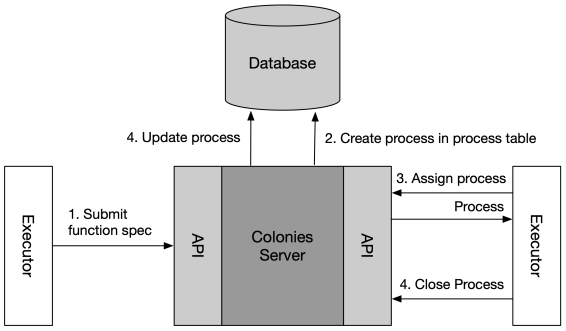
Separation of concerns (SoC) is a design principle to break down a complex software system into smaller, more manageable parts. For example, HTTP APIs can be used to abstract away implementation detail and provide a clear and simple interface for interacting with a particular service. However, HTTP protocols alone are insufficient for handling dynamic environments where components frequently fails or the underlying infrastructures is constantly changing. To address such environments, an alternative mechanism is necessary.

Queues enable software services to communicate indirectly by acting as a buffer between them. Queues makes it possible to decouple each executor and make them operate independently, e.g. an executor can be updated or replaced without affecting other executors. Queues also allow for asynchronous communication between executors, enabling them to process tasks at their own pace. This ensures that slower executors do not bottleneck faster ones, leading to a more efficient and scalable system. Most importantely, queues enable load balancing by distributing tasks among multiple executors, thus making it possible to parallelize workflow execution.

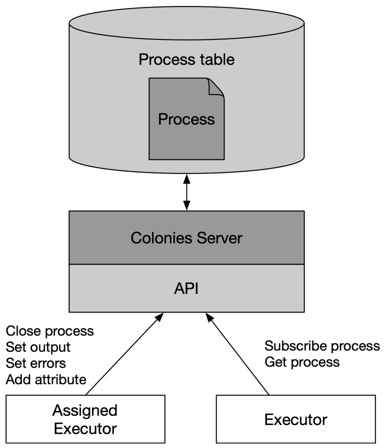
Queues can be implemented in different ways, and while message brokers are a common solution, Colonies adopts an alternative strategy and leverage a standard database and querying it for tasks to assign to different executors. One key advantage of this approach is that it enables fine-grained process assignments, making it possible to assign specific processes to particular executors. For instance, an executor of the browser type can be limited to only executing processes in web applications. This level of granularity cannot easily be implemented using message brokers which generally do not offer introspection of queue, or provide the ability to pull specific messages out of the queue. Generarally, the only way to retrieve a specific message is to pull all messages from the queue, obtain the message, and then place all remaining messages back into the queue in the same order. In contrast, a database can function as a queue and a query can match any columns thus making it possible to assign specific executors to specific processes.

$$priority_{time} = submission_{time} - priority \cdot 10^9 \cdot 60 \cdot 60 \cdot 24 \tag{1}$$

Implementing a queue using a database can be achieved in SQL by utilizing the *order by timestamp* and *limit 1* clauses to select tasks according to their submission times. To make it possible to handle priorites, adjust the submission time so that higher priority processes are processed before lower priority processes. When a process is submitted, a *priority time* value is computed and stored in the database. The primary is a modified submission timestamp calculated by subtracting a delta time calculated as a function of a priority value. This enables prioritization of more critical tasks. Equation 1 shows how the priority time is calculated for a nanosecond timestamp.



(a) Process assignment steps.



(b) Manipulation of a process.

Figure 2: Process management via Colonies HTTP API. Note that only the assigned executor has write access to the process database entry.

3.3 Process tables

Colonies enables executors to interact with each other by submitting function specifications to the Colonies servers. Once submitted, other executors can connect to the server to receive process execution assignments. When a function specification is submitted to the Colonies server, it is store as a process entry in a process table database.

When an executor connects to the Colonies server, the server hang the incomming HTTP connection[2] until the executor is assigned a process, or until a connection timer expires. Note that the Colonies server does not connect to any executor. Rather, it is the responsibility of the executors to connect to the Colonies server. This enables executors to be deployed anywhere on the Internet, behind firewalls, commercial telco networks, or even in web browser-based applications.

Figure 2 illustrates an executor submitting a function specification that is later assigned to another executor. When registering, executors have to specify to the Colonies server which functions they are capable of executing. The Colonies server makes sure that the conditions (i.e requirmenets) of a function specification matches the capability of an executor.

```
{
  "conditions": {
    "colonyid": "0c1168fe986ffe39fad14f17e0bd9e5896f6d968405ac0fb3380154109ee4022"
    "executortype": "test_executor"
  },
  "funcname": "say",
  "args": ["hello world"]
  "maxwaittime": 10,
  "maxexectime": 100
  "maxretries": 3
  "priority": 1
}
```

Figure 3: Example of a function spec.

3.4 A stateless failsafe mechanism

The primary objective of all Colonies API requests is to alter some state stored in the database or retrieve information from the database. The Colonies framework is designed to be stateless, meaning that the Colonies server does not keep any information between requests. In other words, each request is handled independently, without relying on any information from previous requests.

[2]An alternative protocol is to use WebSockets or gPRC to communicate with the Colonies server.

Figure 3 shows an example of a function specification. The *maxexectime* attribute specifies the maximum time an executor may run a process (in this case, 100 seconds). Before a process is assigned to an executor, the Colonies server updates the process entry in the process table database and calculates a deadline when the process must be finished. The server then regularly checks for any running processes that have exceeded their deadlines. If such process is detected, it is reset, allowing it to be re-assigned to other executors.

Making it possible to specify maximum execution time is a simple but powerful mechanism. To scale up a system, more executors can simply be deployed. Scaling down, however, can be more challenging. One solution is to select a set of executors to be removed and then starv them out by denying them new process assignments. Another, simpler solution, is to immediately destroy the executors and use the *maxexectime* failsafe mechanism move back processes from defunct executors to the queue. The *maxexectime* failsafe mechanism ensures that processes will eventually be executed even in the case of failures. This mechanism also relieves the burden of user to check if a process has been executed or not, as they can simply look up the process in the database to get its current status.

Utilizing the *maxexectime* failsafe mechanism not only enhances system reliability, but also provides an opportunity to apply Chaos engineering [21]. For example, a Chaos monkey can be used to deliberately terminate executors. If executors are deployed on Kubernetes, Kubernetes will then automatically redeploy terminated executors. The constant replacement of executor instances ensures that the system is capable of gracefully tolerating failures.

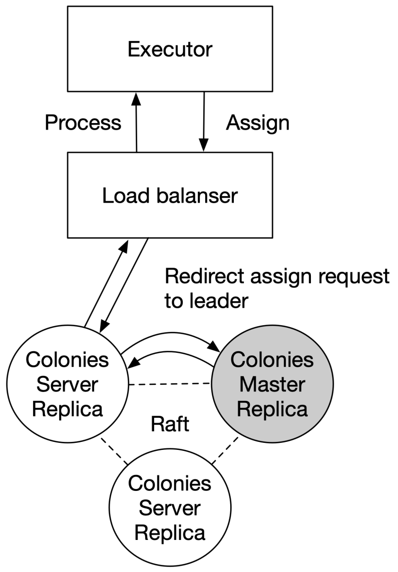


Figure 4: High-availability deployment of Colonies servcer.

3.4.1 Data consistency and distributed consensus

Synchronization is esstential to prevent data inconsistency and race conditions when accessing shared resources concurrently with multiple threads. However, synchronization can also slows down execution as only one thread can access critical sections at a time. By carefully designing multithreaded applications and employing the right synchronization techniques, it possible to minimize the performance impact while still ensuring data consistency and correctness.

The assign API request binds a process from the database to an executor. Given the multi-threaded nature of the Colonies server, it is essential that the assign request is synchronized to ensure that only one thread at a time can modify the database and update the process table, thus preventing multiple executors from being assigned to the same process. To ensure that only one executor can be assigned to a process, the assign request must be synchronization. It is worth noting that synchronization is not necessary for other requests. For example, as the submit request only add new entries to the process table, there is no race conditions. The close request set the output of the function innovation and updates the process state to either successful or failed in the process table. Since there can only be one executor assigned to a process there is no race conditions and consequently no need for synchronization.

To minimize downtime, Colonies supports high-availability deployments. If one Coloniser server crashes, an executor simply need to re-send the failed request, which will then be served by another Colonies server replica. However, by introducing multiple Colonies servers, there is again a risk of race conditions when assigning processes to executors.

This means that the Colonies server replicas must coordintate which replica server incomming assign requests so that precisely one executor is assigned to a process.

Raft [22] is a consensus algorithm specifically designed to manage a replicated log within a distributed system. It functions within a cluster of servers, where a single server takes on the role of leader while the remaining servers act as followers. The leader is responsible for managing the replicated log, processing client requests, and replicating entries to the followers. Followers passively replicate the leader's log and participate in leader elections. The leadership can change over time due to elections triggered by timeouts or other factors.

Incorporting Raft with the Colonies framework allows incoming assign requests to be directed towards the leading Colonies server, thereby ensuring that only one Colonies server replica handles such requests. Figure 4 shows an overview of a high-availability Colonies deployment. A new leader is elected in the event that a Colonies server replica fails. The Raft protocol also enables seamless updates to the Colonies server software by making it possible to upgrade each replica individually. Consequently, Colonies is well-suited for Kubernetes deployments, ensuring high levels of availability and fault tolerance.

3.4.2 Workflows

A workflow is a series of tasks that need to be completed in a specific order. Workflows are often represented as directed acyclic graphs (DAGs), where nodes represent tasks and edges represent dependencies or data flow between tasks.

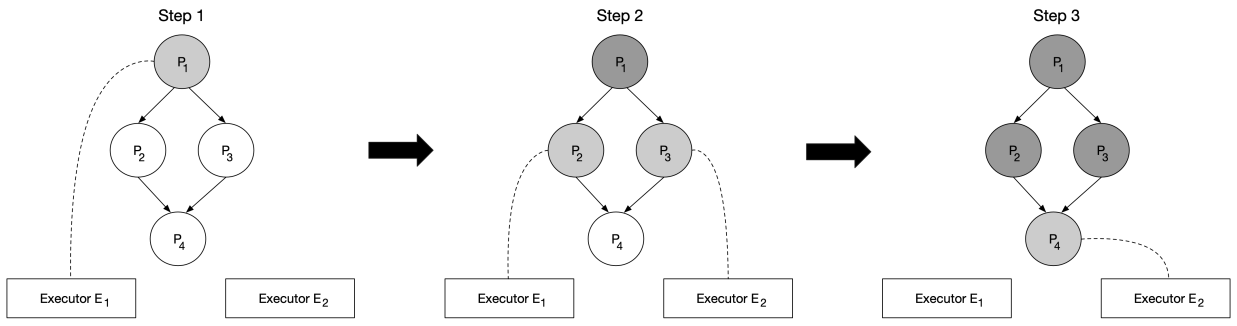


Figure 5: cron management

Table 1: Function Specifications

Function Spec	Function	Executor Type	Priority	Max Exec Time	Max Retries
F_1	gen_nums()	Edge	1	200 s	5
F_2	square()	Cloud	1	200 s	5
F_3	square()	Cloud	1	200 s	5
F_4	sum()	Browser	1	200 s	5

Table 2: Snapshot of Process Table as in Step 2

Process Id	Function Spec	Wait for Parents	Assigned Executor Id	State	Priority Time
P_1	F_1	$False$	E_1	Successful	1679906715352024000
P_2	F_2	$False$	E_1	Running	1679906715353453000
P_3	F_3	$False$	E_2	Running	1679906715354286000
P_4	F_4	$True$	-	Waiting	1679906715355188000

3.4.3 Cron

TODO

3.4.4 Generators

TODO

Table 3: Dependency Table

Process Id	Name	Dependencies
P_1	$Task_1$	-
P_2	$Task_2$	$Task_1$
P_3	$Task_3$	$Task_1$
P_4	$Task_4$	$Task_2, Task_3$

Table 4: Input/Output Table

Process Id	Input	Output
P_1		[2,3]
P_2	2	4
P_3	3	9
P_4	[4,9]	13

3.4.5 Zero-trust security

TODO

4 Use cases

4.1 Serverless computing

5 Evaluation

5.1 Implementation

References

[1] Rafael Ferreira da Silva, Henri Casanova, Kyle Chard, Tainã Coleman, Dan Laney, Dong Ahn, Shantenu Jha, Dorran Howell, Stian Soiland-Reys, Ilkay Altintas, Douglas Thain, Rosa Filgueira, Yadu Babuji, Rosa M. Badia, Bartosz Balis, Silvina Caino-Lores, Scott Callaghan, Frederik Coppens, Michael R. Crusoe, Kaushik De, Frank Di Natale, Tu M. A. Do, Bjoern Enders, Thomas Fahringer, Anne Fouilloux, Grigori Fursin, Alban Gaignard, Alex Ganose, Daniel Garijo, Sandra Gesing, Carole Goble, Adil Hasan, Sebastiaan Huber, Daniel S. Katz, Ulf Leser, Douglas Lowe, Bertram Ludaescher, Ketan Maheshwari, Maciej Malawski, Rajiv Mayani, Kshitij Mehta, Andre Merzky, Todd Munson, Jonathan Ozik, Loïc Pottier, Sashko Ristov, Mehdi Roozmeh, Renan Souza, Frédéric Suter, Benjamin Tovar, Matteo Turilli, Karan Vahi, Alvaro Vidal-Torreira, Wendy Whitcup, Michael Wilde, Alan Williams, Matthew Wolf, and Justin Wozniak. Workflows Community Summit: Advancing the State-of-the-art of Scientific Workflows Management Systems Research and Development. Technical report, 2021.

[2] W. Viriyasitavat, L. Da Xu, G. Dhiman, A. Sapsomboon, V. Pungpapong, and Z. Bi. Service Workflow: State-of-the-Art and Future Trends. *IEEE Transactions on Services Computing*, 16(01):757–772, jan 2023.

[3] Caspar Schmitt and Thomas Kuhr. A workflow management system guide, 2022.

[4] Jaime Garcia Represa Felix Larrinaga Pal Varga William Ochoa Alain Perez Dániel Kozma and Jerker Delsing. Investigation of microservice-based workflow management solutions for industrial automation. *Applied Sciences*, 13(3), 2023.

[5] Chun Ouyang, Michael Adams, Moe Thandar Wynn, and Arthur H. M. ter Hofstede. *Workflow Management*, pages 387–418. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[6] Nikolay Nikolov, Yared Dejene Dessalk, Akif Quddus Khan, Ahmet Soylu, Mihhail Matskin, Amir H. Payberah, and Dumitru Roman. Conceptualization and scalable execution of big data workflows using domain-specific languages and software containers. *Internet of Things*, 16:100440, 2021.

[7] L. Belcastro and Fabrizio Marozzo. *Workflow Systems for Big Data Analysis*. 01 2018.

[8] Apache Airflow. `https://airflow.apache.org`.

[9] RabbitMQ. `https://www.rabbitmq.com`.

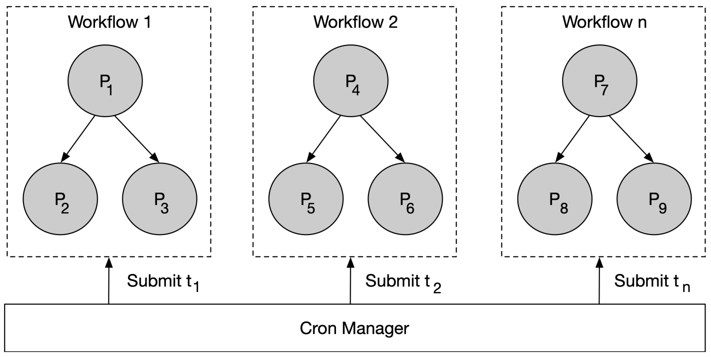


Figure 6: Sample figure caption.

[10] Apache Kafka. `https://kafka.apache.org`.

[11] Argo Workflows. `https://argoproj.github.io/argo-workflows`.

[12] COGNIT: Challenges and Vision for a Serverless and Multi-Provider Cognitive Cloud-Edge Continuum. Under review.

[13] Christian Sicari, Lorenzo Carnevale, Antonino Galletta, and Massimo Villari. Openwolf: A serverless workflow engine for native cloud-edge continuum. In *2022 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*, pages 1–8, 2022.

[14] OpenFaaS. `https://www.openfaas.com`.

[15] Serverless Workflows. `https://serverlessworkflow.io`.

[16] Synapse. `https://github.com/serverlessworkflow/synapse`.

[17] Zhijun Ding, Yuanyuan Zhou, Shuaijun Wang, and Changjun Jiang. SCAFE: A Service-Centered Cloud-Native Workflow Engine Architecture. *IEEE Transactions on Services Computing*, pages 1–14, 2023.

[18] Jaime Garcia Represa. Workflows in Microservice Based System of Systems, 2022.

[19] Jerker Delsing. *IoT automation: Arrowhead framework*. Crc Press, 2017.

[20] Armu Sungkar and Tena Kogoya. A review of grid computing. *Computer Science and IT Research Journal*, 1:1–6, 04 2020.

[21] Principles of Chaos Engineering - The Chaos Engineering Manifesto. `https://principlesofchaos.org`.

[22] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.