Colonies - Compute Continuums across Platforms

A Preprint

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 10, 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 these challenges, the paper presents an open-source framework called Colonies[1], which facilitates execution of computational workloads across a diverse range of platforms. Colonies enables 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.

Based on a stateless microservice architecture, complex workflows to be broken down into composable functions. These composable functions can be implemented in any programming language and be executed by distributed executors deployed across platforms, anywhere on the Internet such as IoT, cloud, edge, and supercomputers. A zero-trust security protocol enables a collection of distributed executors to operate as a unified entity, thus establishing seamless compute continuums across platforms.

In addition to a technical description of the Colonies framework, the paper also describes some potential use cases. The paper describes how Colonies can be leveraged to build a remote sensing platform on Kubernetes, serve as a building block for edge and serverless computing, and be integrated with the Slurm workload manager.

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

Keywords Distributed computing · HPC · 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 retraining 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 misconfiguration 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 platforms.

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 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 deliver higher throughput as the costs of starting new jobs are minimal. This is particularly useful when launching large container images, or workloads (e.g. Julia scripts) that can take a 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. Attempts 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 5.1. However, in contrast to previous work, Colonies is platform independent and does not require Kubernetes. By using a zero-trust security protocol, functions can be securely executed by distributed executors 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 executors.

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 are 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 architecture 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.

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 remote computer programs, 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

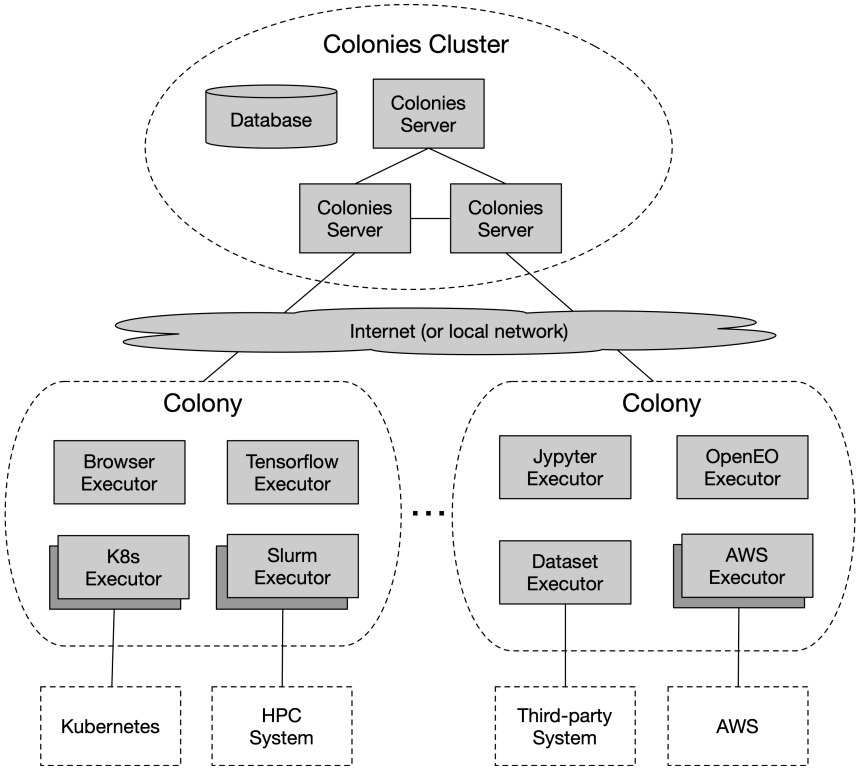


Figure 1: Overview of the Colonies framework. Executors may be deployed anywhere on the Internet.

twin of a real computing process. It is essential to note that a process does not necessarily have to be an operating system process, but 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 types of processes. One could view a Colony as an organization or a society of distributed computer softwares acting as 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 platforms. Zero-trust security is fundamental to the Colonies framework and will be further discussed in Section 3.4.5. The following section outlines the underlying design principles and describe the Colonies framework in more detail.

3.1 Microservices

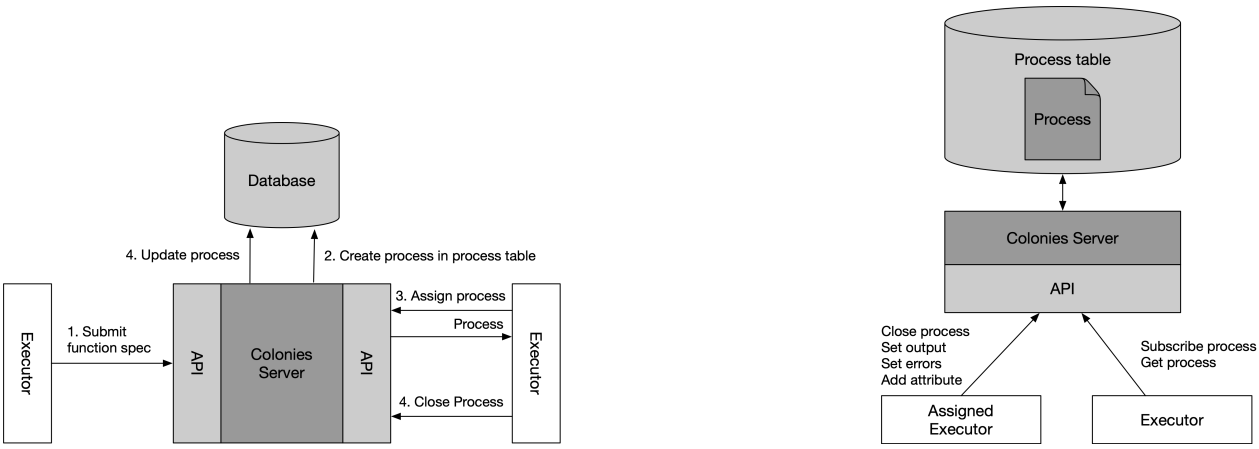
Microservices [21] is an architectural design pattern in which an application is structured as a collection of small, independently deployable, and loosely coupled services that communicate with other microservices through a well-defined API. By dividing the application into smaller, focused microservices,

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 have attractive properties and simplify 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 monitoring 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 platforms. Colonies must define the sequence in which processes should be executed, manage 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.



(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.

The Colonies framework uses a combination of technologies to address the aforementioned challenges. A fundamental design principle is statelessness, 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 how queues can be used 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 fail or the underlying infrastructure 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 make 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 enable 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 importantly, 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 adopt an alternative strategy and leverage a standard database and query 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 queues, or provide the ability to pull specific messages out of the queue. Generally, 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.

Table 1: Process Table

Process Id	Function Spec	Wait for Parents	Executor	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

Table 2: 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

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 servers to receive process execution assignments. A new process entry is added to the process table database when a function specification is submitted to the Colonies server.

When an executor connects to the Colonies server, the server hangs the incoming 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 the executors. 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.

Listing 1: Example of a function specification.

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

Figure 2 depicts 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 then makes sure that the conditions (i.e requirements) part of the function specification matches the capability of an executor.

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

Table 1 shows an example of a process table. The process table also contains a reference to a function specification depicted in Table 2. To assign a process to an executor, the Colonies server searches in the process table to find a process matching a waiting executor. By using the *priority time* column to sort the processes, the process table serves as a queue. This can be done in SQL by utilizing the *order by* to sort

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

processes according to their submission time. To make it possible to handle priorities, the submission time is adjusted so that higher priority processes are processed before lower priority processes. When a process is submitted, a *priority time* value is calculated and stored in the process table. Equation 1 shows how the priority time is calculated for a nanosecond timestamp.

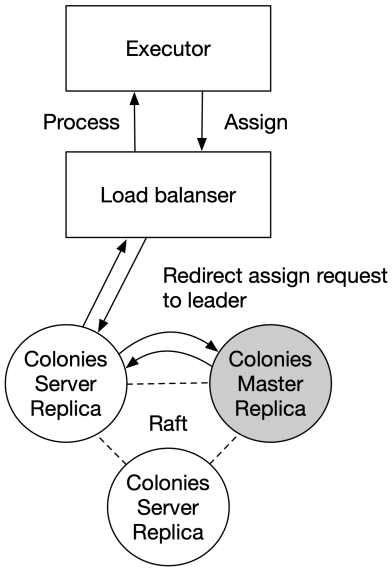


Figure 3: High-availability deployment of Colonies.

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 in memory between requests. Each request is handled independently without relying on information from previous requests.

Figure 1 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 a 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 starve them out by denying them new process assignments. Another, simpler solution, is to immediately destroy the executors and use the *maxexectime* failsafe mechanism to 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 the 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 [22]. 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 flux of executor replacements ensures that the system is capable of gracefully tolerating failures.

3.4.1 Data consistency and distributed consensus

Synchronization is essential to prevent data inconsistency and race conditions when accessing shared resources concurrently with multiple threads. However, synchronization can also slow 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 is 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 synchronized. It is worth noting that synchronization is not necessary for other requests. For example, as the *submit* request only adds new entries to the process table, there are no race conditions. The *close* request sets 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 are 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 resend 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 coordinate which replica server incoming assign requests so that precisely one executor is assigned to a process.

Raft [23] 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.

Adding 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 3 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 processes that need to be completed in a specific order. In Colonies, workflows are represented as directed acyclic graphs (DAGs) where nodes represent processes and edges represent dependencies and data flow between processes. Similar to processes, workflows are managed completely stateless. When a workflow is submitted, all processes part of the workflow are submitted and added to the process table similar to how ordinary processes are handled. To control the order processes can be assigned, Colonies sets a flag, *wait for parents* to prevent processes to be assigned to an executor before their dependencies have been satisfied. Note that processes may run in parallel if a sufficient number of executors are available.

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	$Task_4$	$Task_2$, $Task_3$

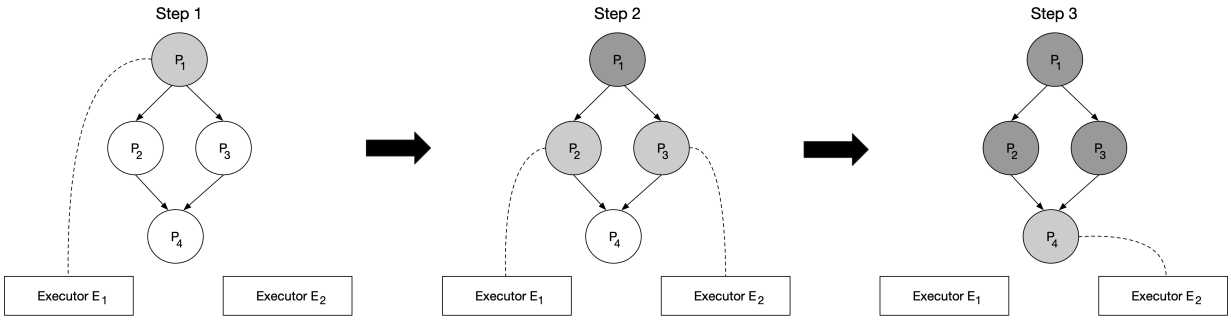


Figure 4: Workflow execution timeline.

When a process is closed, the Colonies server checks its child processes to see if all their parent processes have also completed. If all parents of a child process have finished, the server updates the process table in the database by setting the flag *wait for parents* to false, which allows child processes to be assigned to executors. This operation is also done statelessly when processing the close request.

Figure 4 portrays an execution timeline for a workflow. Upon submission, process P_1 is assigned to Executor E_1. After being closed, processes P_2 and P_3 are simultaneously assigned to E_1 and E_2, allowing for concurrent execution. Lastly, process P_4 is assigned to E_2.

To be able to generate a DAG, an additional database table is required. Table 3 shows an example of a dependency table storing relationships between processes. The table is updated when a workflow is submitted to a Colonies server. As mentioned before, when an executor is assigned a process it obtains exclusive rights to that process within the process table. This exclusivity also means that it is possible for an exector to dynamically add new children processesto the workflow DAG on the fly. As a result, it becomes possible to implement MapReduce [24] data processing patterns similar to Hadoop [25].

Table 4: Input/Output

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

To manage data flow between processes that are part of a workflow, it is necessary to store the input and output values resulting from function invocations in a database. An example of such a database is presented in Table 4. An executor can subsequently query the output from a parent process and use that as arguments while invoking a function.

3.4.3 Cron

The Colonies framework is based on a stateless architecture where each request is treated as an independent transaction and is processed without any information from prior requests. Since there is no session data stored in memory, Colonies can easily switch between servers without disruptions or data losses. In a stateful architecture on the other hand, the server must maintain a record of connected clients' state and session data in memory, which can result in complications if a server fails or crashes.

Cron is a time-based job scheduler designed to trigger workflows at predefined intervals. It can for example be used to automate tasks such as routinely fetching data from external systems. To make Colonies robust and easy to implement it is essential to make Cron workflows stateless. This can be achieved by introducing an additional database table to store information about Cron workflow, and let the elected Colonies server leader assume the responsibility of managing Cron workflows through a two-step process.

Firstly, the leading Colonies server calculates a future deadline, indicating when a specific Cron workflow is scheduled to be executed. The leader server then conducts periodic scans of the Cron table. When the current time surpasses the calculated deadline for a particular workflow, a new workflow is automatically submitted. Note that this protocol is completely stateless as no session information is stored in memory between cron scans. In the event that the leader Colonies server experiences a crash, a new leader is simply selected and assumes the responsibility of managing Cron workflows according to the two-step protocol.

3.4.4 Generators

Another feature of the Colonies framework are so-called generators. Generators are workflow templates that are automatically triggered when specific conditions are met. Generators can be used to accumulate input data into bulks to improve performance or parallelism. For example, an external service downloads satellite images and can use a generator to automatically triggered a workflow when 10 images are collected. This is achieved by sending a special *pack request* containing input data to a Colonies server, targeting a certain generator.

Generators facilities integration with third-party systems as it allows for stateless implementation where interactions with Colonies can be done in a fire-and-forget style. Colonies guarantees that all pack requests submitted to a generator are processed correctly, even if a Colonies server is restarted or fails. Without generators, third-party integration software would have to use an alternative method such as a message broker or a database to enqueue input data and periodically empty the queues to generate workflows themselves. This adds complexity to the integration process and may require additional infrastructure components. Generators are a powerful feature of the Colonies framework that can help simplify integration and implement robust data processing pipelines.

Generators can be implemented using a similar method to how Cron workflows are implemented. The elected leader Colonies server takes on the responsibility for scanning a special generator table to decide if a workflow should be triggered. This approach ensures that there is a single point of control for generating workflows and prevents generation of duplicate workflows. If the leader server crashes, a new leader Colonies server is automatically elected, which takes over the responsibility for triggering generators. This ensures that the generation of workflows is not impacted by server failures, and the system can continue to operate smoothly even in case of failures. It can be worth pointing out that since the pack request only adds information to the generator table and does not manipulate any states, it can be processed by any Colonies server without synchronization precautious. This means that any Colonies server in the cluster can handle incoming pack requests, which increases the scalability and fault-tolerance of the system as a whole.

3.4.5 Zero-trust security

As executors may be deployed across multiple platforms, there is a need for a unified security model that can be applied on top of any platform. Zero-trust security [26] is a security model that assumes that any device or user is a potential threat, even if they are located within a secure network perimeter. Therefore, zero-trust security requires that every interaction between clients and servers is verified and authenticated before being processed, making it ideal for heterogeneous platform deployments. However, there must be a way to verify the identities of the clients to determine whether they should be granted access.

As previously stated, a colony is a collection of distributed executors. Executors part of the same colony trust each other and can thus submit function specifications and get process execution assignments. To implement such a scheme, the Colonies server must be able to verify the identity of the executors and then check their colony membership. This can be accomplished in various ways. One solution is to use public key encryption and assign each executor a pair of keys - a public key and a private key. The public key is openly available and uploaded to the Colonies server, whereas the private key is kept secret and is only used by the executor to sign API request messages. The Colony server can then verify the signature of incoming request messages and use the public keys to determine whether an executor is a member of the colony it is attempting to interact with.

ECDSA (Elliptic Curve Digital Signature Algorithm) [27] is a digital public key encryption signature algorithm. It is widely used in blockchains such as Bitcoin and Ethereum to verify transactions. One of the advantages of ECDSA is its ability to recover public keys from received messages and signatures without explicitly transmitting the public keys. This is particularly useful as the identity of an executor can be calculated simply as the SHA-3 hash of the recovered signature, saving space and eliminating the need for storing public keys.

In Colonies, there exist three distinct roles, each with its own set of responsibilities and authority levels. The Colonies server owner holds the highest level of authority and has permission to add new colonies to the Colonies server. The colony owner is responsible for the management of a given colony and has permission to register or unregister executors within that colony. Executors, on the other hand, have the lowest level of authority and are only authorized to manage processes within their appointed colony. This functionality can be implemented by introducing a new table to the Colonies database, the colony table.

Table 5: Colony Id: $4787a5071856a4acf702b2ffcea422e3237a679c681314113d86139461290cf4$

Executor Id	Executor Name
$8a491bcac0be623a54411dd0934fdcdc9c844de5700527a7dbc9da08a6a8310d$	E_1
$6a65f40343999415f47e739f09b625ab437b069b5f591f9844c234533f505bee$	E_2
$41e4c1b10f92a53b7a5a86620ed366635901a1a96c48312a0ef566a13217fe03$	E_3

Table 5 shows an example of a colony table consisting of three executors. In this case, to register a new executor, the SHA-3 hash of the recovered signature needs to match the identity $4787a5071856a4acf702b2ffcea422e3237a679c681314113d86139461290cf4$ stored in the colony table. This operation can only be performed by the colony owner possessing the corresponding private key. Similarly, when an executor connects to the Colonies server to either submit a function specification or get a process assignment, it needs to sign the API message with its private key. The Colonies server will then recover the identity of the executor and look up the colony table to check if the executor is a rightful colony member. See Appendix A for additional details on how the various roles apply to different Colonies operations.

An advantage of using a stateless architecture for workflow management as proposed in the paper is the ability to monitor executor performance to detect any unusual or suspicious behaviors. Because all transactions are recorded in a database, it is possible to track the activity of individual executors and identify any unusual patterns or behaviors. For example, if an executor suddenly starts to consume a lot of resources, it could indicate a performance issue, or it could be a sign of malicious activity, such as a denial-of-service attack. Automatic monitoring can for example be implemented using anomaly detection algorithms and machine learning to grade the performance of each executor. Administrators can then be automatically notified if the grade of an executor exceeds a certain threshold, enabling them to take appropriate action.

4 Implementation

Colonies is implemented in Golang and is publicly available on GitHub[3] under the MIT license. It consists of a statically compiled binary that offers a CLI tool for deploying Colonies servers or managing the system. Furthermore, there are several software development kits (SDKs) available for various programming languages, including Golang[4], Rust[5], Julia[6], JavaScript[7], Python[8], and Haskell[9]. As an

[3] https://github.com/colonyos/colonies
[4] https://github.com/colonyos/colonies/tree/main/pkg/client
[5] https://github.com/colonyos/rust
[6] https://github.com/colonyos/Colonies.jl
[7] https://github.com/colonyos/colonies.js
[8] https://github.com/colonyos/pycolonies
[9] https://github.com/colonyos/haskell

introduction to Colonies, the following section gives a brief overview of the Colonies tools and how to use the Python SDK.

4.1 Python SDK

A colonies application typically consists of a network of executors written in different programming languages and deployed across multiple platforms. The executors are small microservices that interact with one another by submitting function specifications or workflows, which are then executed by other executors that are part of the same colony.

The first step in implementing an executor in Python is to create a private key and identity, followed by registering the executor with a colony. Registering an executor can only be carried out by the colony owner with access to the colony private key. Code in listening 2 illustrates how to register an executor and which functions a specific executor is capable of running. In the example, an executor of type *helloworld_executor* is registered to run a function named *helloworld*.

Listing 2: Register an executor to a colony in Python.

```
colonies = Colonies("localhost", 50080)

crypto = Crypto()
colonyid = "4787a5071856a4acf702b2ffcea422e3237a679c681314113d86139461290cf4"
colony_prvkey = "ba949fa134981372d6da62b6a56f336ab4d843b22c02a4257dcf7d0d73097514"
executor_prvkey = crypto.prvkey()
executorid = crypto.id(executor_prvkey)

executor = {
    "executorname": "helloworld_executor",
    "executorid": executorid,
    "colonyid": colonyid,
    "executortype": "helloworld _executor"
}

colonies.add_executor(executor, colony_prvkey)
colonies.approve_executor(executorid, colony_prvkey)

# register capability run the helloworld function
colonies.add_function(executorid, colonyid, "helloworld",  executor_prvkey)
```

After registration, the executor must connect to the Colonies server and request process assignments of the type *helloworld_executor*. Note that executor properties, such as the executor type, are not explicitly specified, but rather derived implicitly by the Colonies server from the executor identity, which is derived from the signature generated by executor's the private key. This approach ensures that the executor can only be assigned processes that match its registered capabilities.

Listing 3: Assigning and executing a process in Python.

```
while True:
    process = colonies.assign(colonyid, 10, executor_prvkey)
    if process["spec"]["funcname"] == "helloworld":
        colonies.close(process["processid"], ["hello world"], executor_prvkey)
```

This *assign* request is always initiated by the executor using an HTTP-based protocol, allowing executors to be deployed behind firewalls. When receiving the request, the Colonies server hangs the request until a matching process is found or a timer expires. In the example in Listing 3, the timer is set for 10 seconds, after which an exception is raised, and the executor must reconnect to the server.

When assigned to a process, the executor interprets the metadata stored in the process data structure, including the function name and arguments, and performs some kind of computation such as preprocessing some data or training a neural network. Upon completion, the executor closes the process and sets the output. In the example, the output is set to the the string *helloworld*.

Listing 4: Submitting a function specification.

```
func_spec = create_func_spec(func="helloworld",
                             colonyid=colonyid,
                             executortype="helloworld_executor",
                             priority=0,
                             maxexectime=100,
                             maxretries=3)
process = colonies.submit(func_spec, executor_prvkey)
```

Listing 4 demonstrates how to submit a function specification using the Python SDK and call the *helloworld* function. Note that the executor must complete the process within 100 seconds. If the execution takes longer, the process is reassigned to another executor, and the previosuly executor receives an error upon closure. As discussed in Section 3.4, establishing boundaries on computations is critical to achieving robustness and supporting infrastructural changes that may result from DevOps or IT operations.

The Colonies CLI tool can be used to

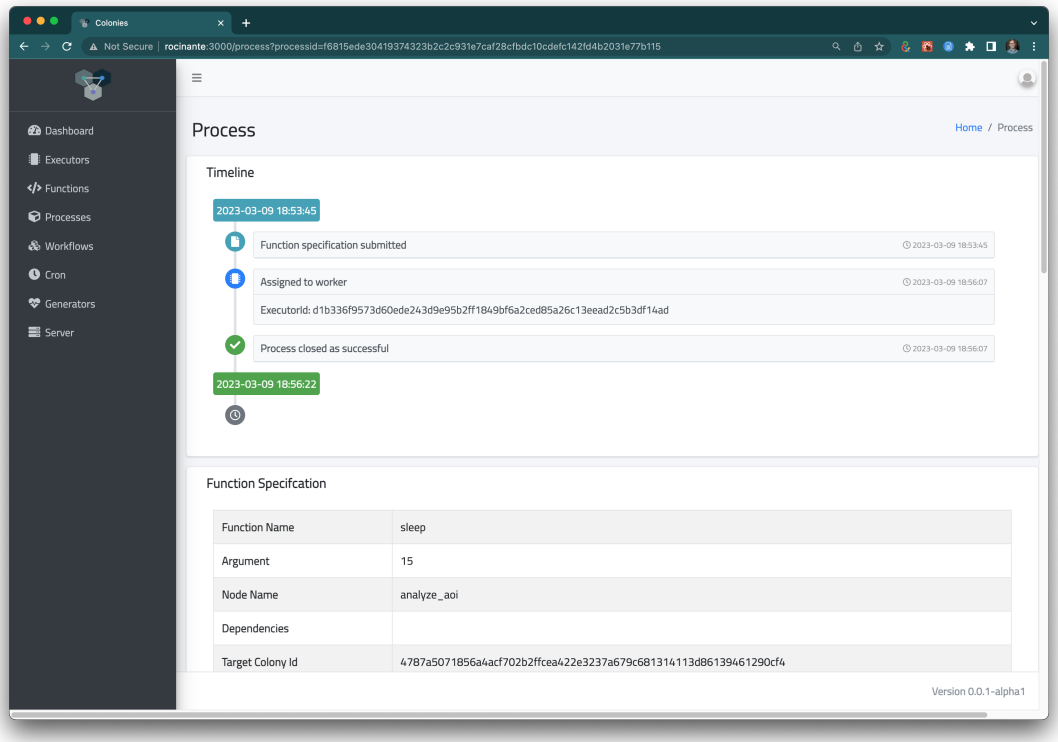


Figure 5: Process information. The timeline makes it possible to follow the execution of process in realtime.

5 Use cases

5.1 Serverless computing

5.2 Edge computing

5.3 Remote sensing

5.4 HPC integration

6 Discussion

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`.

[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] Microservices - A definition of this new architectural term. https://martinfowler.com/articles/microservices.html.

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

[23] 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.

[24] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[25] Apache Hadoop. https://hadoop.apache.org.

[26] Christoph Buck, Christian Olenberger, André Schweizer, Fabiane Völter, and Torsten Eymann. Never trust, always verify: A multivocal literature review on current knowledge and research gaps of zero-trust. *Computers & Security*, 110:102436, 2021.

[27] Panagiotis Kontogiannis and Theodora Varvarigou. ECDSA Private Keys Study of Security. *OALib*, 06:1–20, 01 2019.

A Appendix - Colonies operations

Operations	Role	Assignment	Note
add_colony	Server owner		Registers a new colony to the Colonies server.
delete_colony	Server owner		Unregisters a colony from the Colonies server.
add_executor	Colony owner		Register a new executor to a colony.
delete_executor	Colony owner		Unregister a executor from a colony.
approve_executor	Colony owner		Approve an executor to get process assignments.
reject_executor	Colony owner		Disaprove an executor from getting process assignments.
get_executors	Executor		List all executors member of a colony.
get_executor	Executor		Get info about an executor.
get_colonies	Executor		List all colonies on the Colonies server.
get_colony	Executor		Get info about a colony.
subscribe_process	Executor		Subscribe to a particular process.
subscribe_processes	Executor		Subscribe to processes status changes.
submit	Executor		Submit a function specification.
assign	Executor		Assign and binds a waiting process to an executor.
unassign	Executor	x	Unassign a process from a executor.
close	Executor	x	Close a process as successful.
fail	Executor	x	Close a process as failed.
get_processes	Executor		List all waiting, successful or failed processes.
get_process	Executor		Get info about a process.
delete_all_processes	Colony owner		Delete all processes.
delete_process	Colony owner		Delete a process.
add_attribute	Executor	x	Add attribuite to a process.
remove_attribute	Executor	x	Remove attribute from a process.
subscribe_attribute	Executor		Subscribe to a attribute.
submit_workflow	Executor		Submit a workflow.
get_workflow	Executor		Get info about a workflow.
get_dag	Executor		Get info about a DAG.
add_child	Executor	x	Add a child to a process part of a DAG.
add_generator	Colony owner		Add a generator.
get_generators	Executor		Get info about a generator.
pack	Executor		Add input to an a generator.
remove_generator	Colony owner		Remove a generator.
add_cron	Colony owner		Add a cron to a colony.
remove_cron	Colony owner		Remove a cron from a colony.
get_cron	Executor		Get info about a cron.
delete_cron	Colony owner		Delete a cron from a colony.
get_crons	Executor		List all crons part of a colony.
add_function	Executor		Register a function to an executor.
remove_function	Executor		Unregister a function from an executor.
get_functions	Executor		List all functions supported by the colony.