

## System for Task Management and Processing

### Technical Field

The present invention pertains generally to a task management and processing systems within computing environments, and in particular a modular task orchestrator that leverages repository sources for implementing a range of configuration management tools.

### Background

Configuration management tools, which are pivotal in ensuring consistent control over system resources, have historically involved significant human interaction. Prominent tools in this domain, such as Ansible, Terraform, Puppet, SaltStack, and Chef, have specific operational nuances that necessitate a meticulous and human-centric approach to defining and managing configurations. For example, most of these tools, including Ansible, Chef, SaltStack, and Puppet, require a preliminary setup often termed a "golden image" to ensure optimal functioning. Chef and SaltStack, in particular, typically demand an agent-based architecture, primarily to manage workloads in systems that do not have direct internet access. In contrast, Ansible employs a push-based operational model which demands direct connectivity. While managing configurations for devices such as Cisco switches or F5 load balancers might benefit from this approach, the reliance on direct connections can be constraining and may impede configurations in settings where consistent, direct connectivity is not always assured or practical.

In addition to the above tools, GitOps mechanisms, exemplified by tools like Flux and ArgoCD, have emerged with functionalities tailored for Kubernetes environments. Such tools are adept at sourcing files from Git repositories and subsequently deploying them as artifacts on Kubernetes. This deployment is typically predicated on pre-defined schemas, often represented as YAML manifest files. While this streamlined approach offers simplicity, it is not without its limitations. One of the primary constraints is the rigid nature of these predefined schemas, binding the tool's functionalities to what is explicitly defined within these schemas.

Moreover, despite the different operational approaches and specificities of these tools, they all share the need for an initial foundational setup. This setup could manifest as configurations on the server-side, the client-side, or even procedural tasks like "joining a cluster." Thus, the realm of configuration management and deployment tools, while extensive and versatile, still leaves room for innovations that could streamline processes, minimize manual interventions, and offer greater adaptability across different computing environments.

Consequently, it is desirable to provide a system for task management and processing that address the disadvantages or limitations of the existing art or, at the very least, provide the public with a useful alternative.

### Summary

- 5     The present disclosure aims to provide new and useful systems for task management and processing in computing environments, and in particular a modular software system that leverages repository sources for implementing a range of configuration tools.

In broad terms, the present disclosure proposes a task management and processing system including a task orchestrator configured to manage and coordinate a plurality of tasks including  
10   initiating, terminating, or pausing the tasks in response to detected changes to a remote directory, wherein the tasks relate to configuration management operations. One way of employing this system is further configuring the task orchestrator to initiate a self-update function for fetching updates from a plurality of data repositories and to initiate a fetch task function for retrieving tasks for execution from the repositories. As can be appreciated from  
15   the described embodiment, the task orchestrator may be further configured to validate software components through one or more cryptographic signatures or hash checks. A policy compliance module may be configured to ensure tasks executed by the task orchestrator correspond to predefined settings or security protocols. A task runner module may be configured to interface with the task orchestrator and execute tasks that are authorized by the  
20   task orchestrator.

In particular embodiments, the task orchestrator may interface with Ansible playbooks upon detecting changes in a GitOps directory.

In implementations, at least one of the plurality of tasks may include an ad hoc task for facilitating immediate, single-use operations within the system.

- 25   In implementations, at least one of the plurality of tasks may include a scheduled task programmed to run at predetermined time intervals or specific times.

In some embodiments, at least one of the plurality of tasks may include a backend task representing long-running activities within the system.

- In implementations, the self-update function may access a source Git repository and may fetch  
30   a list organized based on at least one of binary paths, hashes, and GPG keys.

In particular embodiments, the task orchestrator may integrate with a Kubernetes cluster, augmenting Kubernetes functionalities without duplicating existing Kubernetes components.

An API Server may act as a primary interface for interaction with the task orchestrator, offering at least one of authentication, authorization, and push model capabilities aligned with container orchestration resources.

- 5 An access control module may manage the authorization of the task orchestrator by using at least one of public and private key pairs, tokens, and external authentication protocols.

In embodiments, the task management and processing system may operate based on a push model, actively transmitting tasks or configurations to associated components that are connected to or interact with the system, thereby minimizing operational latency.

- 10 In implementations, the task runner module may be configured to validate binary files using both Pretty Good Privacy (PGP) and Message Digest Algorithm 5 (MD5) hashes prior to execution of at least one of the tasks or after at least one of the tasks have processed.

In some embodiments, a controller manager may collaborate with the task orchestrator for efficient task orchestration and state reconciliation via dispatching tasks or configurations

- 15 when the system operates in a push model.

A database may serve as a centralized repository with the API server facilitating Create, Read, Update, and Delete (CRUD) operations and a controller manager reading from and writing to the database for task management.

In particular embodiments, the task management and processing system may include a configuration agent or entity initialized using predefined libraries and dependencies, said

- 20 agent or entity dedicated to managing configurations for tools.

The configuration agent or entity may utilize standard libraries from Golang code for routine operations.

The attributes of the configuration agent or entity may include at least one of cmd (operation to be executed), playbookFile (configuration file), md5Checksum (integrity verification), and

- 25 gpgSignature (source authentication).

The system may create an instance or object of the configuration agent or entity, equipped with specific values or attributes for managing configurations.

The system may verifies integrity or authenticity of configuration files or playbooks using

- 30 security tools embedded within one or more of the predefined libraries or the dependencies.

In implementations, the task management and processing system may provide capabilities to temporarily pause configuration processes without termination.

In embodiments, the task management and processing system may determine an operation for an instantiated configuration agent or entity, selecting from operations including but not limited to the initiating, terminating, or pausing of the tasks.

The implementations may be expressed as a method, or alternatively as a system including one or more computers and one or more storage devices storing instructions that when executed by the one or more computers cause the one or more computers to perform the method. It may also be expressed as a computer program product, such as downloadable program instructions (software) or one or more non-transitory computer storage media storing instructions. When executed by one or more computers, the program instructions cause the one or more computers to perform the method.

Embodiments described herein may thereby solve the technical problem of effectively streamline configuration tasks, reduce manual interventions, and enhance adaptability to diverse computing environments in the realm of configuration management and deployment tools.

The above description is provided as an overview of some implementations of the present disclosure. Further description of those implementations, and other implementations, are described in more detail below.

#### Brief Description of the Drawings:

Embodiments of the invention will now be explained for the sake of example only, with reference to the following figures in which:

Figure 1 illustrates a block diagram of an example task management and processing system for orchestrating configuration management tools, according to an embodiment of the invention.

Figure 2 illustrates a flowchart illustrating an example process of using the system of Figure 1 for configuration management, according to an embodiment of the invention.

Figure 3 illustrates a block diagram of an example task management and processing system with a control manager for orchestrating configuration management tools, according to an embodiment of the invention.

Figure 4 illustrates an example directory structure of a modular architecture for a task orchestrator, where each configuration management tool is integrated as an individual plugin, according to an embodiment of the invention.

Figure 5 illustrates a block diagram illustrating an example computer system which may be configured to implement the systems and methods as disclosed herein.

#### Detailed Description

Embodiments will now be discussed with reference to the accompanying FIG.s, which depict one or more exemplary embodiments. These embodiments are described in sufficient detail to enable those skilled in the art to practice the embodiments and it is to be understood that mechanical, logical, and other changes may be made without departing from the scope of the embodiments. Therefore, embodiments may be implemented in many different forms and should not be construed as limited to the embodiments set forth herein, shown in the FIG.s, and/or described below.

Unless otherwise defined, all terms (including technical and scientific terms) used herein are to be interpreted as is customary in the art. It will be further understood that terms in common usage should also be interpreted as is customary in the relevant art.

Figure 1 illustrates a block diagram of an example task management and processing system 100 for orchestrating configuration management tools, according to an embodiment of the invention.

A task orchestrator 102 serves as the primary control unit for the operation of system 100, orchestrating task management, execution, and enforcing policy compliance. The task orchestrator 102 may manage, control, and oversees the lifecycle of tasks, allowing for initiation, pausing, or halting of tasks as needed. For example, the task orchestrator 102 may monitor a remote file folder directory, and upon detecting changes, fetches the new content to initiate the execution of associated tasks. The task orchestrator 102 may also perform self-updating or “self-healing” 104 by periodically updating itself using a remote GitOps folder, ensuring its operation is aligned with the most recent configurations and policies.

The task orchestrator 102 may be built for minimal resource consumption, such that it can run on diverse platforms, from personal computers to embedded systems, without hindering performance. The orchestrator 102 may integrate a dynamic inventory by conducting regular system scans, relaying the collected metrics and system information to a centralized Dynamic Inventory. The orchestrator 102 may further prioritize security by validating binaries through PGP signatures and MD5 hash checks before execution. This not only ensures the authenticity of tasks but also allows for the integration of new tools via plugins, emphasizing the system's scalability and adaptability. The orchestrator 102 may perform clean-up operations by

maintaining system integrity by cleaning up residuals post-task execution, optimizing resource allocation and ensuring operational hygiene.

In some implementations, the orchestrator 102 may activate Ansible playbooks or scripts/directives from other comparable configuration tools when changes are identified in the  
5 GitOps directory, thereby automating tasks such as configuration and application deployment. For example, orchestrator 102 may interface with Ansible for dynamic inventory sourcing, determining targets for playbook runs. Orchestrator 102 may further integrate principles of GitOps including, for example, Version Control, Desired State Management, and Automatic Reconciliation.

10 A self-update mechanism 104 may ensure that the task orchestrator 102 is up-to-date by fetching updates from various sources, such as Git repositories, databases, or object stores, and upon request from the orchestrator 102, the mechanism 104 may directly update an orchestrator binary or its submodules to the latest version. The orchestrator 102 updates may be deployed using cloud-init and user-data, standard mechanisms for initializing cloud  
15 instances, ensuring seamless deployment in cloud environments. In embodiments of the present invention, the self-update procedure may be configured to access a source Git repository and retrieve a list organized in descending order based on binary paths, MD5 hashes, and GPG keys. Notably, the most recent version of the orchestrator is positioned at the list's outset. Upon retrieval, the orchestrator 102 procures the associated configuration file,  
20 executing instructions from the initial to the final element within the list. Following successful initiation, the orchestrator 102 enters a dormant state, reactivating only upon the arrival of a subsequent update pull. This methodology ensures that the orchestrator 102 consistently operates utilizing the latest, verified, and secure version available.

The orchestrator 102 may request a fetch task mechanism 106 to fetch tasks for further  
25 execution. It may retrieve tasks from various sources, including databases, object stores, or other designated repositories. In the context of a GitOps paradigm, the mechanism 106 may synchronize with the Git repository. The fetch task mechanism 106 corresponding to a source may detect repository modifications, promptly notifying the orchestrator 102 of such alterations for initiating corresponding responsive actions.

30 An ad hoc task 108 may facilitate immediate, single-use operations primarily aimed at setting up specific configurations or parameters within the system 100. These specific tasks 108, which are usually unique and non-repetitive, can be initiated by the task orchestrator 102 either manually or in response to particular system or user events.

A scheduled task 110 may represents task programmed to run at predetermined intervals or  
35 specific times, akin to cron jobs. The task orchestrator 102 may be initiate these tasks 110,

which might include actions like nightly backups, hourly data pulls, or daily report generations, based on a predefined schedule.

A backend task 112 may represent long-running or persistent activities within the system 100. The orchestrator 102 may initiate these tasks 112 as services or daemons designed to run indefinitely, such as monitoring services or data stream processors.

#### System Config/Policy module

A system config/policy module 114 of system 100 may serve as the gatekeeper or policy enforcer for tasks executed by the task orchestrator 102. In example embodiments, it may check to ensure that the task meets established policies before the task orchestrator executes any task. This could be security guidelines, compliance mandates, resource constraints, or other criteria. The module 114 might also manage or dictate how tasks are executed based on preset configurations. For example, if a task needs specific system settings or environments to run, the System Config/Policy module 114 ensures those settings are met. If any task handles sensitive data, this module 114 may ensure the data is securely managed, possibly decrypting encrypted data using tools like Ansible Vault.

#### Task runner module

A task runner module 116 may serve as an engine or executor of system 100. Once the task orchestrator 102 has decided on a task to run and the System Config/Policy module 114 has validated it, the task runner may take over to execute the task. Whether it's a one-off ad hoc task, a scheduled task, or a long-running backend task, the task runner may execute the actual operation. The module 116 may interact closely with the task orchestrator 102, receiving tasks and possibly sending back execution statuses or results. This feedback loop ensures the orchestrator 102 knows the outcome of each task, whether successful, failed, or in-progress. The start/stop/pause labels correspond to direct management of tasks by the task runner 116, granting flexibility to pause, resume, or terminate tasks as circumstances change within system 100.

Figure 2 illustrates a flowchart illustrating an example process 200 of using the system of Figure 1 for configuration management, according to an embodiment of the invention.

In step 202, a preliminary phase of initializing libraries and dependencies related to a configuration entity or agent is performed. This step corresponds to setting up essential resources and libraries for a configuration agent or entity to operate. Here, a configuration agent or entity can be understood as a dedicated agent or entity responsible for managing configurations for any tool, such as Ansible, Terraform, Kubernetes, etc. For example, the

configuration agent or entity may be a “configuration worker” represented as a ‘Worker’ struct in Golang code which is a struct defining a blueprint for creating instances of workers, each with its properties and methods to manage configurations.

In example embodiments, libraries may provide pre-compiled routines such as standard Golang libraries ‘fmt’ and ‘os/exec.’ Dependencies may include software modules, specific runtime environments, necessary configuration files, system resources like open ports or services, and security tools for verifying integrity. Such security tools may include cryptographic libraries or tools to verify the integrity and authenticity of the provided playbook files.

The initialization of step 202 may correspond to setting up or preparing the libraries and dependencies for us. Such initialization may include loading necessary libraries into memory, checking and verifying the presence of required tools or services, reading and parsing necessary configuration files, and setting up initial state or variables that the configuration agent or entity requires.

In step 204, a configuration entity or agent responsible for managing configurations is defined and initialized. Here, a data structure for managing and handling configuration operations may be established. For example, a “configuration worker” such as a Worker struct in Go code may be specified. This configuration worker may act as a configuration agent managing configurations for tools like Ansible, Terraform, or Kubernetes. The worker may possess certain properties or attributes such as cmd (the operation to be executed), playbookFile (the configuration file), md5Checksum (for integrity verification), and gpgSignature (for source authentication). In embodiments, step 204 may be understood as a process of initializing a configuration worker with foundational attributes, akin to declaring and defining a class or struct in programming, where a framework/blueprint is established but an object or instance is not created yet.

In step 206, a new instance or object of the configuration entity or agent (such as the Worker struct) is instantiated with specific values. For example, in Golang code, this may be achieved using a “New Worker” function which sets attributes like a playbook file and associated security checks (checksum and its signature) and returns a new stance of the Worker struct with these values set. Step 206 equips the system 100 with a specialized agent, primed to manage configurations based on a specific set of parameters.

In step 208, a specific operation—such as initiating, terminating, or pausing the configuration—is chosen for the configuration entity to execute. This decision aligns with methods like Start(), Stop(), and Pause() in Golang code. Step 208 may correspond to a

crucial decision point for determining the next action of the worker struct and guiding the subsequent steps in the process 200.

In step 210, the configuration entity (e.g., Worker struct) begins the execution of a task, such as deploying configurations or running scripts. In Go code, this step may align with the Start() method of the Worker struct, which may trigger tasks such as executing an Ansible playbook. Step 210 may represent the worker entering its active phase of operation.

In step 212, the system 200 terminates an ongoing configuration process. In the Go code, this may correspond to the Stop() method of the Worker struct, designed to cease an active configuration task. This step 212 may immediately interrupt the worker's current operation and may often be used in cases of errors or changes in the system.

In step 214, the system 200 temporarily pauses the configuration process without ending it. In the Go code, this may align with the Pause() method of the Worker struct, which halts the process without termination. This step 214 is often employed for reasons like resource management or awaiting other processes, allowing potential resumption from the paused point.

Figure 3 illustrates a block diagram of an example task management and processing system 300 with a control manager 324 for orchestrating configuration management tools, according to an embodiment of the invention.

A task orchestrator 302 of system 300 is a central component designed for the efficient handling, scheduling, and management of tasks. A self-update mechanism 304 may ensure that the task orchestrator 302 is up-to-date by fetching updates from various sources, such as Git repositories, databases, or object stores, and upon request from the orchestrator 302, the mechanism 304 may directly update an orchestrator binary of the orchestrator 302 or its submodules to the latest version.

A functionality of the task orchestrator 302 may be to ensure the proper initiation of binaries. For example, the orchestrator 302 may be equipped with feedback mechanisms, reporting on the status and health of these binaries in real-time.

System 300 may be configured to function as a push model, facilitating a proactive approach in task assignment and configuration dispatch. Contrary to conventional pull models wherein systems predominantly solicit information, system 300 would instead actively transmit tasks or configurations to designated entities. The architecture and operational framework of this push model may be structured to correspond to the functionalities of a Kubernetes object system, leveraging inherent advantages in task management and efficiency. An aspect of the task orchestrator 302 is its capability to activate a Kubernetes cluster, enhancing the

orchestrator's functionality and capabilities without duplicating Kubernetes functionalities or components.

The orchestrator 302 may monitor a remote self-update source, which could be a Git repository, database, or object store. Any changes detected in the self-update source may prompt the orchestrator 302 to fetch the updated tasks or configurations via fetch task mechanism 306. This ensures that the orchestrator 302 is always updated with the latest tasks and can function with self-healing and self-updating capabilities.

The orchestrator 302 may request a fetch task mechanism 306 to fetch tasks for further execution. It may retrieve tasks from various sources, including databases, object stores, or other designated repositories. The mechanism 306 may synchronize with the Git repository and may detect repository modifications, promptly notifying the orchestrator 102 of such alterations for initiating corresponding responsive actions. These response actions in the context of detecting modifications in a Git repository may include fetching the latest changes, triggering automated tests, deployment, rollback, adjusting resources, updating dependencies, and the like.

Ad hoc tasks 308, schedule tasks 310, and backend tasks 312 are various task types that the orchestrator 302 can process. Ad hoc tasks 308 may refer to tasks that are set up as needed or on-the-fly. Schedule tasks 310 may refer to tasks that run based on a specific schedule, e.g., daily or weekly reports or checks. Backend tasks 312 may refer to tasks that run continuously or for extended periods.

System Config/Policy 314 provides guiding policies and configurations that guide the task orchestrator's operations. It may ensure that the orchestrator 302 and other components of system 300 align with the defined policies or configurations. System Config/policy 314 may establish whether a task should be executed based on these policies.

A task runner 316 may be configured to execute the tasks fetched by the schedule/task source function 306. The task runner 316 may verify binary files for PGP signatures and MD5 hashes before executing the files. Post-execution, the task runner 316 may ensure clean-up of processes and binaries, especially after the processing of ad hoc tasks 308.

The API Server 318 may serve as the primary interface for interaction with the task orchestrator 302. While sharing similarities to the Kubernetes' API server, API server 318 incorporates its own authentication and authorization mechanisms, which encompass methods such as CA/public/private key pairs, token keys, and external authentication processes. API server 318 may also employ its own Role-Based Access Control (RBAC) for precise authorization. This server 318 not only enables a push model, enabling the creation

of Kubernetes Objects but also processes requests related to the orchestrator 302, including tasks, configurations, and other entities.

The Access Control Module 320 is designed to oversee and administer authentication, leveraging methods such as CA/public/private key pairs, tokens, or external authentication protocols. Concurrently, it may employ a distinct Role-Based Access Control (RBAC) mechanism to manage authorization, thereby ensuring that interactions with the orchestrator 302 and its integral components are strictly limited to authorized entities.

The Logs/Metrics Module 322 is configured to document the operational activities and performance indices of the orchestrator 302, as well as its executed tasks. These documented logs and metrics are designed to be transmitted to a centralized repository, referred to as a Dynamic Inventory, facilitating comprehensive reporting and analytical evaluations.

Within the architecture of System 300, the task orchestrator 302 and Controller Manager 324 collaborate to ensure efficient task orchestration and state reconciliation. The task orchestrator 302 may be structured to function as a central system, especially when operating in a push model. This push model, distinct from traditional pull paradigms, permits system 300 to actively dispatch tasks or configurations to its associated entities. By design, this proactive approach minimizes latency, expediting system responses to ever-changing requirements.

A collaborative operation between the task orchestrator 302 and control manager 324 ensures streamlined task orchestration and state reconciliation. The task orchestrator 320, when operating in the push model, actively dispatches tasks or configurations. Simultaneously, the control manager 324 oversees the state reconciliation of API objects, ensuring alignment between their current states and their desired outcomes. The control manager 324 may operate in tandem with the API Server 318, which serves as the primary interface for interactions and updates related to the API objects. This collaboration allows for integrated collaboration: as the API Server 318 processes changes and requests, the control manager 324 is positioned to reconcile these changes promptly and ensure they align with the intended system configurations. This approach offers an adaptable and efficient environment for task orchestration within System 300.

Database 326 may be a centralized storage solution, such as a repository. The API server 318 may facilitate CRUD (Create, Read, Update, Delete) operations based on extra requests and then store or retrieve relevant data from database 326. Controller Manager 324 may both read from and write to database 326, including fetching current states of tasks or configurations to compare and reconcile them and storing new states or updates after processing or reconciliation is complete. Updated task configurations or task states corresponding to CRUD operations may be stored in database 326.

Figure 4 illustrates an example directory structure 400 of a modular architecture for the task orchestrator 102, where each configuration management tool is integrated as an individual plugin, according to an embodiment of the invention.

In example implementations, the directory structure 400 is designed to encapsulate a compact binary optimized for minimal size, rapid execution, and reduced dependencies. This design not only streamlines deployment and management of the task orchestrator 102 but also ensures efficient performance with limited resource overhead.

#### Main Application Directory ('app/')

In preferred embodiments, 'app/' may represent a root directory or main directory of the task orchestrator system 102. The root directory may serve as the top-level container for all components, configurations, plugins, and other relevant files pertaining to the task orchestrator 102. In practical implementations, the designation "app/" may be substituted with the specific given name of the orchestrator system 102.

#### Plugins Directory and Architecture ('plugins/')

In the directory structure 400, the 'plugins/' directory is specifically designed to enable the extensibility and modularity of the task orchestrator 102. This structure facilitates the integration of various configuration tools without requiring changes to the core application code of any individual tool or service. Within this directory, each tool or service is encapsulated as its own distinct plugin, denoted as 'plugin1/', 'plugin2/', and the like.

In example embodiments, each plugins/ directory may serve as a repository for individual modules or extensions. These modules or extensions typically provide additional functionalities or features that can be added to the main task processing systems 100 and 300. For example, an individual plugin may be tailored for integrating with APIs to perform specific tasks like deploying apps, scaling, or managing workloads. A plugin may execute specific plans or modules based on the desired infrastructure state, helping automate cloud infrastructure. Further, plugin1/ or plugin2/ may correspond to running specific playbooks for tasks like configuration management, application deployment, or routine system tasks. Additionally, one or more plugins may correspond to GitOps and integrating with popular GitOps tools like ArgoCD or Flux, allowing the orchestrator 102 to trigger deployments or synchronize configurations based on changes in a Git repository.

Within each plugin's directory, such as plugin1/, a shared object file, like plugin1.so, might be present. This shared object file, in various implementations, could either contain the executable code related to the respective plugin or a compiled binary encapsulating specific

functionalities of that plugin. For instance, the shared object file for a plugin might act as a dynamically linked plugin library dedicated to specific operations for a certain kind of infrastructure management tool. Similarly, a shared object file might be a dynamically linked plugin library for a specific tool, empowering the task orchestrator to execute configuration tool specific playbooks or commands. In the case of GitOps, the object file may be configured to ensure the system's current state aligns with the state detailed in a Git repository.

Each plugin comes with an associated configuration directory, typically named `config/`, which houses configuration files such as `plugin1_config.json`. These files play a pivotal role in customizing the behavior of the plugins, allowing users to define settings, parameters, and other essential details tailored to their specific requirements. For example, the configuration files might detail connection specifics, cluster settings, or any other parameters crucial for seamless interaction with a configuration tool specific cluster such as a Kubernetes cluster. The configuration file may encompass default inventory files, user credentials, or other prerequisites for executing configuration tool or service specific operations. The configuration files could list repository details, access tokens, and other configuration nuances with respect to specific operations.

#### Signatures Directory (signatures/):

The `signatures/` directory emphasizes security by facilitating the verification of plugins or configurations before they are loaded or executed. It contains `.asc` files, such as `plugin1.asc`, which are ASCII-armored PGP signatures associated with individual plugins. These signatures may be cross-checked against the `publickey.gpg`—a public key file used for PGP encryption and digital signatures—to confirm the authenticity and integrity of the plugins. This verification process ensures that the plugins are genuine, untampered, and free from potential malicious content, aligning with the security measures commonly found in systems like Kubernetes, Terraform, Ansible, and GitOps.

#### Version Tracking (current\_version.txt):

The `current_version.txt` file indicates the current version of the task orchestrator 102. It plays a vital role in tracking versions of the orchestrator 102, which is crucial for ensuring updates, maintaining compatibility, and facilitating debugging processes. This file can also assist the orchestrator 102 in recognizing the availability of newer versions, potentially supporting self-updating mechanisms.

#### Primary configuration file (config.json):

The config.json file serves as the main configuration for the task orchestrator 102, consolidating global settings, default values, and key system-wide parameters. This file establishes fundamental behaviors and configurations across the orchestrator system 102.

Go programming language (main.go):

- 5 The task orchestrator 102 may be developed in Go (Golang) as indicated by the 'main.go' file, a programming language renowned for its efficiency, especially in the cloud-native domain. Notably, many cloud-native tools, including Kubernetes and Terraform, are written in Go, facilitating integration and extension. Although Ansible primarily utilizes Python, Go can augment performance-intensive components. The file acts as the pivotal entry point for the  
10 application, containing the orchestrator's core logic.

FIG. 5 is a block diagram of a computing device 500 configured to implement functionalities of the task orchestrator 102, in accordance with the embodiments of the systems and methods disclosed herein. The computing device 500 includes a processor 522 (which may be referred to as a central processor unit or CPU) that is in communication with memory devices including  
15 secondary storage 524 (such as disk drives), read only memory (ROM) 526, random access memory (RAM) 528. The processor 522 may be implemented as one or more CPU chips. The computing device 500 may further comprise input/output (I/O) devices 530, and network connectivity devices 532. The secondary storage 524 is typically comprised of one or more disk drives or tape drives and is used for non-volatile storage of data and as an over-flow data  
20 storage device if RAM 528 is not large enough to hold all working data. Secondary storage 524 may be used to store programs which are loaded into RAM 528 when such programs are selected for execution.

In this embodiment, the secondary storage 524 has an order processing component 524a comprising non-transitory instructions operative by the processor 522 to perform various  
25 operations of the method of the present disclosure. The ROM 526 is used to store instructions and perhaps data which are read during program execution. The secondary storage 524, the RAM 528, and/or the ROM 526 may be referred to in some contexts as computer readable storage media and/or non-transitory computer readable media.

I/O devices 530 may include printers, video monitors, liquid crystal displays (LCDs), plasma  
30 displays, touch screen displays, keyboards, keypads, switches, dials, mice, track balls, voice recognizers, card readers, paper tape readers, or other well-known input devices.

The processor 522 executes instructions, codes, computer programs, scripts which it accesses from hard disk, floppy disk, optical disk (these various disk-based systems may all be considered secondary storage 524), flash drive, ROM 526, RAM 528, or the network

connectivity devices 532. While only one processor 522 is shown, multiple processors may be present. Thus, while instructions may be discussed as executed by a processor, the instructions may be executed simultaneously, serially, or otherwise executed by one or multiple processors.

5 In the computing device 500, a graphics processing unit (GPU) 534 can be included to enhance processing capabilities. The GPU 534 is a specialized electronic circuit designed for rendering images, animations, and video, and is highly efficient at parallel processing tasks. This makes them well-suited for various machine learning and artificial intelligence related operations. The GPU 534 communicates with the processor 522, secondary storage 524,  
10 ROM 526, RAM 528, I/O devices 530, and network connectivity devices 532. It can be integrated within the processor 522 as an integrated GPU (iGPU) or as a separate, dedicated GPU connected through a high-speed interface. Inclusion of the GPU 534 allows for more efficient processing of tasks that benefit from parallelism, leading to increased system performance, especially in applications optimized for GPU-based processing. The GPU 534  
15 can also offload tasks from the CPU 522, improving overall system efficiency. By incorporating the GPU 534 into the computing device, the system can be transformed into a specific purpose machine with enhanced processing capabilities for parallel tasks, as taught by the present disclosure.

Although the computing device 500 is described with reference to a single computer, it should  
20 be appreciated that the computing device may be formed by two or more computers in communication with each other that collaborate to perform a task. For example, but not by way of limitation, an application may be partitioned in such a way as to permit concurrent and/or parallel processing of the instructions of the application. Alternatively, the data processed by the application may be partitioned in such a way as to permit concurrent and/or  
25 parallel processing of different portions of a data set by the two or more computers. In an embodiment, virtualization software may be employed by the computing device 800 to provide the functionality of a number of servers that is not directly bound to the number of computers in the computing device 500. In an embodiment, the functionality disclosed above may be provided by executing the application and/or applications in a cloud computing environment.  
30 Cloud computing may comprise providing computing services via a network connection using dynamically scalable computing resources. A cloud computing environment may be established by an enterprise and/or may be hired on an as-needed basis from a third-party provider.

Additional components, such as one or more application specific integrated circuits,  
35 neuromorphic computing units, field programmable gate arrays, or other electronic or photonic

processing components can also be included and used in conjunction with or in place of the processor 522 to perform processing operations. The processing operations can include machine learning operations, other operations supporting the machine learning operations, or a combination thereof.

5 The technical solution detailed in present disclosure may be embodied in the form of a computer program product. The computer program product may be stored in a non-volatile or non-transitory storage medium, which can be a compact disk read-only memory, USB flash disk, or a removable hard disk. The computer program product includes a number of instructions that enable a computer device (personal computer, server, or network device) to  
10 execute the methods provided in the embodiments described herein. For example, such an execution may correspond to a simulation of the logical operations, including the training and aggregation of model updates in the federated learning process, as described herein. The software product may additionally or alternatively include number of instructions that enable the computing device 800 to execute operations for configuring or programming a digital logic  
15 apparatus in accordance with embodiments of the present invention.

By programming and/or loading executable instructions onto the computing device, at least one of the CPU 522, the RAM 528, and the ROM 526 are changed, transforming the computing device in part into a specific purpose machine or apparatus having the novel functionality taught by the present disclosure. It is fundamental to the electrical engineering and software  
20 engineering arts that functionality that can be implemented by loading executable software into a computer can be converted to a hardware implementation by well-known design rules.

It will be appreciated to those skilled in the art that the preceding examples and embodiments are exemplary and not limiting to the scope of the present disclosure. Whilst the foregoing description has described exemplary embodiments, it will be understood by those skilled in  
25 the art that many variations of the embodiment can be made within the scope and spirit of the present invention. It shall be noted that elements of any claims may be arranged differently including have multiple dependencies, configurations and combinations.

## Claims

1. A task management and processing system, comprising:

a task orchestrator configured to manage and coordinate a plurality of tasks, including at least one of initiating, terminating, or pausing the tasks in response to detected changes to a remote directory, wherein the tasks relate to configuration management operations;

wherein the task orchestrator is further configured to initiate a self-update function for fetching updates from a plurality of data repositories and to initiate a fetch task function for retrieving tasks for execution from the repositories;

wherein the task orchestrator is further configured to validate software components through one or more cryptographic signatures or hash checks;

a policy compliance module is configured to ensure tasks executed by the task orchestrator correspond to predefined settings or security protocols; and

a task runner module configured to interface with the task orchestrator and execute tasks that are authorized by the task orchestrator.

2. The task management and processing system of claim 1, wherein the task orchestrator interfaces with Ansible playbooks upon detecting changes in a GitOps directory.

3. The task management and processing system of claim 1 or 2, wherein at least one of the plurality of tasks comprises an ad hoc task for facilitating immediate, single-use operations within the system.

4. The task management and processing system of any one of claims 1-3, wherein at least one of the plurality of tasks comprises a scheduled task programmed to run at predetermined time intervals or specific times.

5. The task management and processing system of any one of claims 1-4, wherein at least one of the plurality of tasks comprises a backend task representing long-running activities within the system.

6. The task management and processing system of any one of claims 1-5, wherein the self-update function accesses a source Git repository and fetches a list organized based on at least one of binary paths, hashes, and GPG keys.

7. The task management and processing system of any one of claims 1-6, wherein the task orchestrator integrates with a Kubernetes cluster, augmenting Kubernetes functionalities without duplicating existing Kubernetes components.

8. The task management and processing system of any one of claims 1-7, wherein an API Server acts as a primary interface for interaction with the task orchestrator, offering at least one of authentication, authorization, and push model capabilities aligned with container orchestration resources.
- 5 9. The task management and processing system of any one of claims 1-8, wherein an access control module manages the authorization of the task orchestrator by using at least one of public and private key pairs, tokens, and external authentication protocols.
10. The task management and processing system of any one of claims 1-9, wherein the system operates based on a push model, actively transmitting tasks or configurations to  
10 associated components that are connected to or interact with the system, thereby minimizing operational latency.
11. The task management and processing system of any one of claims 1-10, wherein the task runner module is further configured to validate binary files using both Pretty Good Privacy (PGP) and Message Digest Algorithm 5 (MD5) hashes prior to execution of at least one of the  
15 tasks or after at least one of the tasks have processed.
12. The task management and processing system of any one of claims 1-11, wherein a controller manager collaborates with the task orchestrator for efficient task orchestration and state reconciliation via dispatching tasks or configurations when the system operates in a push model.
- 20 13. The task management and processing system of any one of claims 1-12, wherein a database serves as a centralized repository with the API server facilitating Create, Read, Update, and Delete (CRUD) operations and a controller manager reading from and writing to the database for task management.
14. The task management and processing system of any one of claims 1-13, wherein the  
25 system includes a configuration agent or entity initialized using predefined libraries and dependencies, said agent or entity dedicated to managing configurations for tools.
15. The task management and processing system of claim 14, wherein the configuration agent or entity utilizes standard libraries from Golang code for routine operations.
16. The task management and processing system of claim 14 or 15, wherein attributes of the  
30 configuration agent or entity comprise at least one of cmd, representing an operation to execute, playbookFile serving as a configuration file, md5Checksum for verifying integrity, and gpgSignature for authenticating a source.

17. The task management and processing system of any one of claims 14-16, wherein the system creates an instance or object of the configuration agent or entity, equipped with specific values or attributes for managing configurations.

5 18. The task management and processing system of any one of claims 14-17, wherein the system verifies integrity or authenticity of configuration files or playbooks using security tools embedded within one or more of the predefined libraries or the dependencies.

19. The task management and processing system of any one of claims 1-18, wherein the system provides capabilities to temporarily pause configuration processes without termination.

20. A task management and processing method comprising:

10 managing and coordinating a plurality of tasks using a task orchestrator, including at least one of initiating, terminating, or pausing the tasks in response to detected changes to a remote directory, wherein the tasks relate to configuration management operations;

using the task orchestrator to initiate a self-update function for fetching updates from a plurality of data repositories;

15 using the task orchestrator to initiate a fetch task function for retrieving tasks for execution from the repositories;

validating software components through the task orchestrator using one or more cryptographic signatures or hash checks;

20 ensuring, using a policy compliance module, that tasks executed by the task orchestrator correspond to predefined settings or security protocols;

interfacing a task runner module with the task orchestrator; and

executing tasks that are authorized by the task orchestrator using the task runner module.

25

30