Detailed Functional Overview of an API and Workflow Engine for Scientific Research Computing

Nathan Freeman
Texas Advanced Computing Center
Austin, Texas, USA
nfreeman@tacc.utexas.edu

Joe Stubbs
Texas Advanced Computing Center
Austin, Texas, USA
jstubbs@tacc.utexas.edu

Richard Cardone
Texas Advanced Computing Center
Austin, Texas, USA
rcardone@tacc.utexas.edu

Christian Garcia
Texas Advanced Computing Center
Austin, Texas, USA
cgarcia@tacc.utexas.edu

ABSTRACT

Constructing and executing reproducible workflows is fundamental to performing research in a variety of scientific domains. Many of the current commercial and open source solutions for workflow engineering impose constraints—either technical or budgetary—upon researchers, requiring them to use their limited funding on expensive cloud platforms or spend valuable time acquiring knowledge of software systems and processes outside of their domain expertise. Even though many commercial solutions offer free-tier services, they often do not meet the resource and architectural requirements (memory, data storage, compute time, networking, etc) for researchers to run their workflows effectively at scale. Tapis Workflows abstracts away the complexities of workflow creation and execution behind a web-based API with a simplified workflow model comprised of only pipelines and tasks. This paper will detail how Tapis Workflows approaches workflow management by exploring its domain model, the technologies used, application architecture, design patterns, how organizations are leveraging Tapis Workflows to solve unique problems in their scientific workflows, and this projects's vision for a simple, open source, extensible, and easily deployable workflow engine.

CCS CONCEPTS

• **Software and its engineering**;

KEYWORDS

workflows, containers, API, HPC

ACM Reference Format:
Nathan Freeman, Joe Stubbs, Richard Cardone, and Christian Garcia. 2023. Detailed Functional Overview of an API and Workflow Engine for Scientific Research Computing. In *Practice and Experience in Advanced Research Computing (PEARC '23), July 23–27, 2023, Portland, OR, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3569951.3593609

PEARC '23, July 23–27, 2023, Portland, OR, USA

ACM ISBN 978-1-4503-9985-2/23/07.
https://doi.org/10.1145/3569951.3593609

1 INTRODUCTION

Tapis Workflows is an API, workflow engine, and set of supporting services deployed in the Tapis ecosystem [26] designed to enable the construction and execution of research computing workflows. It is capable of building, testing, and persisting container images for scientific applications, running small containerized applications, executing arbitrary code, and communicating with external entities via HTTP requests. It is also fully integrated with the suite of Tapis services, allowing users to run research computing jobs on HPC infrastructure via the Tapis Jobs API [23], as well as trigger functions via Abaco [13], a distributed computing, Actor Model-based functions-as-a-service platform. This paper is organized as follows: in section 2, we provide an overview of the use-cases that Tapis Workflows is currently serving, and others that are motivating the development of new functionality; in section 3, we cover the tech-stack of Tapis Workflows and the decisions behind choosing those technologies; in section 4, we discuss source code structure and style in addition how CI/CD for the project is managed; Section 5 covers important aspects of the project's data model; In section 6 we discuss the application architecture of the workflow engine and role of each component in workflow execution; In section 7, we discuss the methods by which users can trigger workflows; In section 8, we discuss future features and features currently in the development pipeline for extending workflow functionality to serve more general use-cases and improve extinsibility; Finally, in section 9 we give a general overview of the development team's assessment on Tapis Workflows' ability to satisfy the current use-cases followed by a brief overview of the content covered in this paper.

2 USE CASES

This sections introduces some common use cases for which Tapis Workflows is currently employed and the details of how organizations in a wide variety of scientific and engineering domains are leveraging it to build and run their workflows.

2.1 Continuous Integration/Continuous Deployment

CI/CD is a common use case served by Tapis Workflows. Generally, these workflows include building images for containerized scientific applications, running any built-in programmatic tests to ensure they function as expected, and pushing them to remote image registries.

2.1.1 Tuitus. Tuitus [24], an NSF-funded project at the University of Texas at Austin, maintains a suite of scientific tools for supporting research in Natural Hazard Engineering. One of the aims of the Tuitus project is to develop a set of CI/CD best practices for applications designed to run on HPC infrastructure. Tapis Workflows will be used by Tuitius to build, test, and persist their containerized scientific applications.

2.1.2 HETDEX. The Hobby-Eberly Telescope Dark Energy Experiment (HETDEX) [17] is an international collaboration between universities including the University of Texas at Austin, Ludwig-Maximilians-Universität München, Georg-August-Universität Göttinge, and Pennsylvania State University, in which the clustering of galaxies is measured using McDonald Observatory's Hobby-Eberly Telescope in an effort to look for potential evolution in dark energy.

The HETDEX group employs Jupyter Notebooks via Jupyter SCINCO [25] for analysis and visualization of data produced by this experiment. The underlying image for their Jupyter Notebook is built using the image building functionality available with Tapis Workflows and subsequently persisted to a remote image registry.

2.2 ETL Pipelines

ETL Pipelines (extract, transform, load) are applications in which data is ingested, transformed, then transferred to a final destination.

2.2.1 JPL NEID project. NEID [5] is an astronomical spectrograph constructed at Penn State for NASA's Jet Propulsion Laboratory. Its purpose is to analyze the spectra of nearby stars for perturbations in order to discover and classify extra solar planets.

As data files are generated by JPL's NEID spectrograph, they are transferred to the local inbox—a Tapis System [12]—via a Globus [3] transfer, and a manifest containing the file paths is generated. Once manifest files are detected, a request is sent to the Tapis Workflows API to run the data transformation. The data transformation consist of running an HPC job via the Tapis Jobs API [23], archiving the data, and transferring the newly transformed data back to JPL via Globus.

3 TECHNOLOGIES

In this section, we discuss the technologies upon which Tapis Workflows is constructed and the decisions behind choosing those technologies.

3.1 API

The Tapis Workflows API makes use of Django [22], an open source Python-based framework for building secure and scalable web applications. Django was chosen for its ability to integrate with a variety of database management systems (DBMS) used in other Tapis projects, its use of object-relational mappers (ORMs) [6] to facilitate rapid development by enabling access to the data layer through high-level abstractions in contrast to direct use of the querying language in code, and built-in security measures for countering common exploits faced by web-based services (SQL injection, cross-site request forgery, etc.).

3.1.1 API Performance. The Tapis Workflows API can perform under loads approximately five times the average expected number of concurrent users without degradation of service. Load tests were performed using Locust [18] at a maximum of 500 concurrent users at a hatch rate 100 users per second for 32,520 requests over 5 minutes. This test resulted in a request failure rate of less than 0.1 percent.

3.2 Message Queue (RabbitMQ)

Tapis Workflows utilizes RabbitMQ [20] for inter-service communication — namely, requests between the Tapis Workflows API and the Workflow Engine. The message queue creates a loose coupling between the Workflows API and Workflow Engine, allowing other APIs and tools to be developed independently around the Workflow Engine. Exchanges on RabbitMQ are also used internally by the Workflow Engine for scheduling workflows and managing the workflow execution life cycle.

3.3 Persistence

Due to the relational nature of the Workflow API's data model, Tapis Workflows employs MySQL for workflow persistence.

4 PROJECT STRUCTURE, CODE STYLE, AND CI/CD

In order to promote and simplify open source contribution to Tapis Workflows, this project has employed a project structure, design patterns, and architectural paradigms found in common use amongst large and complex open source projects.

4.1 Monorepo

All source code for the services that comprise Tapis Workflows are contained within a single code repository, called a monorepo [19]. This has multiple advantages compared to the more common polyrepo [19] pattern. First, code bases that are housed in a single repository benefit from the ability to develop and run unit, functional, and integration test suites across services. For example, the Tapis Workflows API and the Workflow Engine share DAG validation utilities and various API Gateways used for sending requests to other Tapis Services. Additionally, it simplifies CI/CD as container images for each services can be built concurrently as part of a single deployment.

4.2 Code Style

The Tapis Workflows API and Workflow Engine are written in Python with a clear, modular design to encourage and facilitate open-source contributions to the project. This project makes extensive use of advanced object-oriented design commonly used in applications developed in languages such as Python, Java, and C#. Such patterns include Strategy, Builder, Observer, Decorator, Factory, Singleton/Multiton, and Object Pool.

4.3 Github Actions

CI/CD for Tapis Workflows is managed using GitHub Actions [7] and configured to build both the API and Workflow Engine images on push or merge of major branches, run unit and functional tests for both images, and push them to Dockerhub once all programmatic tests pass.

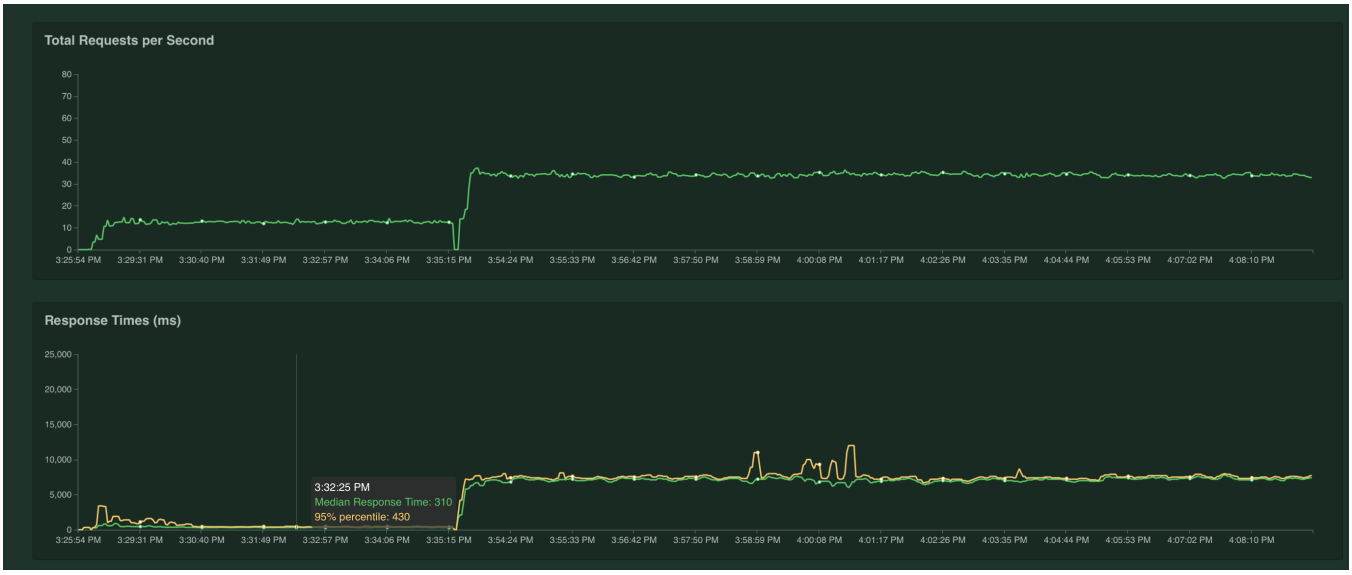

Figure 1: Load test with Locust. 32,520 requests over 5 minutes.

4.4 Branching and Image Versioning

This project roughly follows the GitFlow [8] branching model, maintaining four major branches that correspond to the four environments to which Tapis Workflows will be deployed; dev, staging, test, and release-*. For all non-release branches, the images built for a deployment are tagged with that branch name. For production releases, the branch is named after the release target version prefixed with "release-". When a release branch is pushed, the version is parsed from the branch name and the image is tagged with the number that follows "release-"; ex. A branch named "release-1.2.5" will result in images tagged as the following: tapis/workflows-api:1.2.5 and tapis/workflows-pipelines:1.2.5.

5 DATA MODEL

This section covers the fundamental entities of the Tapis Workflows API and how they relate to each other.

5.1 Tasks

Tasks are the units of work performed in a workflow. There are six different types of tasks, each exposing unique functionality that performs work commonly found in advanced research computing workflows. Each of these tasks' capabilities will be discussed in section 6.5. Tasks in a pipeline are modeled as nodes in a directed acyclic graph (DAG) in which their relationships, i.e. their dependencies, determine the order of their execution, and where tasks without dependencies are the first to be executed.

5.2 Pipelines

Pipelines are collections of tasks and a set of rules for governing the behavior and life cycle of a workflow. On this object, users can modify aspects of the execution profile. These include the max execution time or TTL of a pipeline, the task invocation mode which controls whether tasks are executed concurrently or serially, max retries which specifies the number of times a pipeline can be rerun after failure, and duplicate submission policy which determines whether the Workflow Engine should terminate the current duplicate workflow, cancel the incoming workflow, or run the duplicate workflows concurrently.

5.3 Groups

A Group defines a set of Tapis users that own workflow objects such as pipelines and tasks. All members of a group are capable of creating and running workflows owned by that group. Groups must have a unique id within the Tapis tenant to which they belong.

5.4 Identities

Identities are mappings of Tapis users to identities which are external to the Tapis framework. These identities can be used as a replacement for providing raw credentials in workflow definitions — specifically those required by the image build task (discussed in section 6.5.1) — when building images from source code in private repositories and pushing the resultant image to private registries. Additionally, identities are used to validate requests from external entities that trigger workflows such as Github Actions or Gitlab CI/CD.

The Tapis Workflows API leverages the Tapis Security Kernel (SK) [11] — backed by HashiCorp Vault [4] — to encrypt and store the credentials. Once the credentials for an identity are persisted, they will only ever be shared between the Tapis Workflows API and the Workflow Engine.

5.5 Pipeline Runs & Task Executions

Pipeline Runs and Task Executions are objects that represent the status of Pipelines and Tasks as they are being processed by the Workflow Engine. These objects can only be created by the API during the runPipeline operation, or the Workflow Engine as a

workflow runs through the various stages of the execution life cycle.

6 WORKFLOW ENGINE

The Workflow Engine is composed of seven core components. The Workflow Server, Worker Pool, Workflow Executor, Task Executors, Event Exchange, Middlewares, and Reactive State. This section covers the roles of each component in controlling workflow execution.

6.1 Workflow Server

The Workflow Server is the entry point for the Workflow Engine. It is responsible for establishing and maintaining connection with the message queue (RabbitMQ) and its exchanges (The Inbound Exchange, Retry Exchange, and Dead-letter Exchange), managing the Workflow Executors via the Worker Pool, and handling request idempotency.

6.2 Worker Pool

The Worker Pool is an in-memory collection of Workflow Executor instances organized as a double-ended queue (deque; pronounced "deck"). The number of workers, and thus, number of Workflow Executors instantiated are determined by configurations specified in the deployment files of the Workflow Engine. The Worker Pool is elastic, meaning users can specify a minimum and maximum number of Workers; as requests come in, additional Workers can be added to the pool at runtime up to the limit specified in order to accommodate increased request load.

Worker concurrency is implemented via threads. This was chosen over multi-processing due to the convenience of shared memory and constraints imposed by request idempotency (discussed in section 6.7.2).

6.3 Workflow Executor

The Workflow Executor is the primary workhorse of the Workflow Engine. Each Workflow Executor is capable of processing a single workflow submission at a time. It is responsible for Task Executor dispatching and threading, task dependency management, processing and validating their inputs and outputs, and maintaining the Event Exchange to which Events can be published and subscribed to by various middlewares that handle requests to remote backends and task result archivers.

As tasks are executed by the Workflow Executor, their results are stored on an NFS-server and operated upon by subsequent tasks. Once a workflow has reached a terminal state, all of the results are deleted.

6.3.1 Hooks. The life cycle of a workflow execution is managed through a series of five hooks; on_start, on_change_ready_task, on_change_state, on_task_terminal_state, and on_pipeline_terminal_state.

6.3.1.1 _on_start. The on_start hook is called at the beginning of workflow execution. This hook is responsible for task DAG validation, preparing the file system structure to organize task results and logs, determining which tasks to execute initially, and populating the ready_tasks array with those initial tasks.

6.3.1.2 on_change_ready_task This hook runs when new tasks are appended to the ready_tasks array. For all tasks in the ready_tasks array, Task Executors are dispatched according to their type and the Workflow Executor awaits their terminal state.

6.3.1.3 _on_change_state This intermediate hook is called when Task Executors mutate state in the Workflow Executor. It is registered with a state locking mechanism (Reactive State, covered in section 6.4) that enables a Task Executor to perform operations over shared state between the Workflow Executor and other Task Executors in a thread-safe manner.

6.3.1.4 _on_task_terminal_state When a task reaches a terminal state (succeeded, failed, or terminated), the _on_task_terminal_state hook will clean up the temporary resources created during the task's execution and populate the ready_tasks array with new tasks, thereby triggering the _on_change_ready_tasks hook and processing the next tasks. This occurs recursively until all tasks have entered a terminal state or the workflow reaches its max exec time.

6.3.1.5 _on_pipeline_terminal_state This is the final hook which is called once all tasks in a workflow have either completed, or a task fails causing the workflow to fail. This hook is responsible for cleaning up temporary resources created by the Workflow Executor, and triggering middleware responsible for archiving and reporting the status of workflows and their tasks.

6.4 Reactive State

A single Reactive State object is instantiated per Workflow Executor for storing and mutating state subject to potential race conditions–i.e. state accessed by the Workflow Executor and Task Executors running in multiple threads. This object has two responsibilities. The first is to lock access to the state from other threads when values are set or fetched. The second is to dispatch hooks registered with the Reactive State when values specified in the hook's dependency list are accessed by Task Executors. These hooks allow methods on the Workflow Executor to be called during the state lock, ensuring there are no race conditions for that particular dependency. Once all hooks for that value are called, the state is then released and other threads can access it once again.

6.5 Task Executors

Task Executors instances are spawned by the Workflow Executor to handle the execution of individual tasks based on their type. There are six different types of tasks and a corresponding Task Executor for each. This section covers the unique functionality that they bring to Tapis Workflows.

6.5.1 Image build. Containerization is an important paradigm for packaging and running scientific applications in a reproducible manner. Tapis Workflows leverages existing image building technologies such as Singularity [10] and Kaniko [15] to build both Singularity and Docker images from source code and persisting those images to container registries or in local workflow storage. The Image Build Task also has the capability to convert Docker images into SIF files.

The Image Build task is defined in two steps. First is the "context", i.e. the source of the image to be built. This can be a public or private repository on a source control platform, or a registry

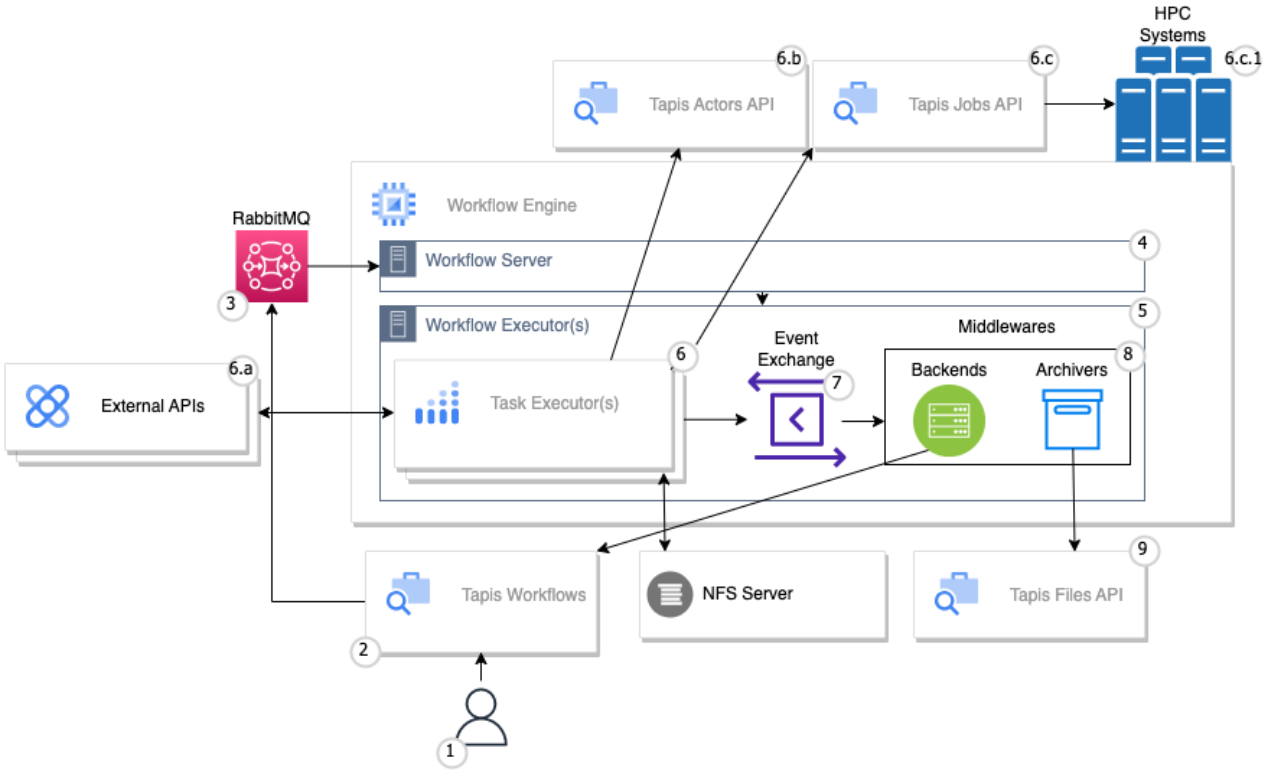

Figure 2: Workflow Engine Architectural Overview

on Dockerhub. For private repositories, the user must provide the credentials necessary to access it when defining the context. The credentials (username and access token) can be furnished directly on the context definition itself or by providing the UUID of a Workflow Identity (mentioned in section 5.4).

The second step is defining the "destination", i.e. where the resultant image is to be persisted. Like the context, credentials or a Workflow Identity will be required to push images to a private registry. Users can also specify a local destination. This useful if users desire to test their images before pushing them to a final remote repository.

6.5.2 Request. Requests enable users to make HTTP requests to applications external to the Workflow Engine. Outputs from completed tasks can be sent from the Workflow Engine to trigger additional workflows that run on external resources. Alternatively, a Request task can fetch data from an external resource to be processed as a part of a workflow execution.

6.5.3 Container run. Container Run task offers a way for users to run small containerized applications (less than 4CPUs and 16GB of memory) run as a Job [14] on the Tapis Kubernetes [16] cluster. This task is intended for applications that do not need to make use of high performance computing infrastructure. This task can be executed in one of two modes which is specified by a boolean value in the "poll" property of a workflow definition.

When set to "true", the task executor queries the Kubernetes API for the job status until some terminal state (Failed or Completed) is reached. If the container exits with a "Completed" status, the stdout of the container is validated to ensure that it conforms to the output specified in the workflow definition (if defined). If an output produced by the container fails validation, the task execution is marked as failed. If the job concludes with status "Failed", the task execution will also be marked as failed and all tasks that are dependent on it will not be executed. When set to "false" the container will run in the background and the Kubernetes Job's status will be ignored.

6.5.4 Function. Function tasks offer a way for users to run arbitrary code in the language and runtime environment of their choice. These environments are furnished with the workflow context—a pre-loaded object that contains the current state of the Workflow Executor—which includes the outputs of previous tasks and can be accessed via the "ctx" constant inside the user-defined code. The user-defined code is provided via the "code" property of the task definition and is expected to be a base64-encoded string.

Functions are a specialized set of the container run type with restrictions. The pods spawned by these tasks are network isolated

as well as restricted from accessing the Kubernetes API via Cluster Roles (CR) and Cluster Role Bindings (CRBs). Additionally, the Kubernetes service token mounted into pods by default is deleted to ensure access to the Kubernetes API is impossible. The runtimes available for use are node19, python3.9, and python2.7. Support for more runtimes will be developed on an as-needed basis.

6.5.5 Tapis-actor. The Tapis Actors API, also known as Abaco [13], is a distributed function-as-a-service platform deployed as a part of Tapis. The Workflow Engine's integration with Abaco allows users to extend their existing actor pipelines with additional tasks in Tapis Workflows.

Like the Function task, actors can be polled until they reach a terminal state, or triggered and run in the background. Since actors can be linked, the workflow engine will poll each actor recursively until all linked actors have finished running. If a single actor ends in a failed terminal state, the task will also fail.

6.5.6 Tapis-job. The Tapis Jobs API is a service with which users can run containerized scientific applications on high performance computing systems. Job submissions are defined as JSON objects on the tapis_job_def property of the task object and submitted directly to the Jobs API during workflow execution. Like other tasks, the submitted Tapis Job can be polled until it reaches a terminal state.

6.6 Middleware

Each Workflow Executor is furnished with a set of middlewares that respond to events generated throughout the life cycle of a workflow. These middlewares come in two types: Backends and Archivers. Backends are used to send data to remote entities regarding status of workflow and task executions and Archivers are used to persist the results generated by the tasks to external entities such as a Tapis System or an S3 bucket.

6.6.1 Event Exchange. The Event Exchange is the mechanism by which a Workflow Executor publishes updates about its current stage of execution. These include pipeline statuses (ACTIVE, COMPLETED, FAILED, TERMINATED, etc), and task statuses (ACTIVE, COMPLETED, FAILED, TERMINATED, SKIPPED, etc). When published, each event contains the state of the Workflow Executor at the moment it was published. This enables independent components, such as the aforementioned middlewares, to perform operations over that state and communicate that data with external entities.

6.7 Workflow termination

There are a number of challenges to managing state in applications that share memory between a variable number of dynamically-generated nested threads. The Workflow Executor components in the Workflow Engine employ a combination thread locking and method interception (aka action filters) via decorators to ensure that state accessed in the various stages of workflow execution remains consistent in order to avoid race conditions.

6.7.1 Termination decorator. The termination decorator [1] is a function that is called before and after life-cycle methods in a Workflow Executor. This is implemented as a decorator factory that can be passed an optional clean-up function. When a life-cycle method is called, the Workflow Executor is checked for a termination status. If it is "terminated" or "terminating", that life-cycle method will be skipped and the optional clean-up function will be invoked to roll-back the state of the executor in preparation for the next workflow execution.

6.7.2 Request Idempotency. In order to track and handle duplicate requests to the Workflow Engine, an idempotency key is created and assigned to each workflow submission. This idempotency can either be directly supplied by the user in a workflow request via the "idempotency_key" property, or constructed according to the unique constraints specified by a property of the same name. These instructions are an ordered list of properties and selectors that, when combined, form the idempotency key.

When a duplicate request is detected, the newly submitted workflow will be governed in accordance with the current running workflow's duplicate submission policy. There are three policies available for handling duplicate requests. "ALLOW" permits the new workflow to run in parallel with the current workflow, "TERMINATE" terminates the current workflow, and "DENY" discards the new workflow submission. Support will also be added for a "DEFERRED" policy which the new request will be placed in a special queue in which it will wait for the current workflow to reach a terminal state before running.

7 TRIGGERING WORKFLOWS

In this section, we will discuss the different methods available for triggering workflows and how to modify workflow execution behavior through the use of directives.

7.1 Tapis Workflows API

The most direct way to trigger a workflow is directly through the Tapis Workflows API. This is done via POST request to the "runPipeline" operation. In the request body, users can specify directives which enable users to override the default behaviors of the workflow. These directives will be discussed in a later section.

7.2 TapisUI

A recent addition to the Tapis framework, a front end web interface called TapisUI [21], enables users to make requests to core Tapis services–in addition to the Workflows API–in the browser.

7.3 Tapipy

Tapipy is the Python SDK used by both developers and users for making API calls to the full suite of Tapis services, including the Tapis Workflows API. It utilizes each service's OpenAPI specification [9] to dynamically generate methods on a Tapis client object that correspond to all possible operations exposed by each of the services APIs. With this tool, users can develop applications or scripts that integrate with the Tapis Workflows API and other services to programmatically submit and run workflows.

7.4 Source control platforms

Users may also trigger their workflows with webhook notifications from source control platforms such as Github Actions and Gitlab CI/CD [2]. Using the template provided in the Tapis Workflows

documentation, users can set up webhook notifications on push or merge.

7.5 Overriding workflow execution behavior

In some scenarios, some properties of workflows or tasks will need one-off modifications in order to accommodate a special circumstance or requirement under which they will be executed. For example, in an image build task, a user may need to tag an image with a different tag than the one defined in the workflow. This is made possible through the use of directives. Directives are commands sent along with a workflow submission that instruct the workflow engine to modify certain properties of a workflow definition before or during its execution. In the use-case mentioned previously, the user would specify a "custom_tag" directive in addition to the new value they would like to use; ex. "v1.2.3" instead of "latest".

Directives leveraged in two ways. The first is by specifying a "directives" property in a workflow submission request. This property is an array comprised of strings that conform to the required syntax. Using the example above in which a user wants to tag the image with the new value "v1.2.3", the string value of that directive would be "custom_tag:v1.2.3". The second way to specify directives is by adding a special string to the end of a commit message on a commit that will trigger a workflow. These directives must follow the same syntax as before with additional constraints. All directives must be inside of a single set of square brackets and separated by a pipe character.

8 FUTURE WORK

This project plans to expand in scope to satisfy the functional requirements of a broader spectrum of use cases in order to be more generally useful to the scientific and research computing communities. This section further elaborates on the vision of this project and how it plans realize that vision through a more flexible, extensible architecture and robust feature set.

8.1 Plugin architecture & Extensibility

One of the visions for this project is to provide a workflow engine that is implementation agnostic, extensible, and generally useful across many different domains and computing workloads. This can be accomplished through a plugin-based architecture against which software engineers can develop their specific implementations for task execution, archiving, and backend notification systems.

As explained previously, workflow and task executors operate on a publish-subscribe model in which events are published by core components of the workflow engine. These events are subscribed to, and consumed by, various middlewares such as backends and archivers. Currently, the workflow engine and certain middlewares–specifically, the Tapis backend and Tapis System archiver–are tightly coupled, forcing an implementation that will not be used in deployments that do not make use of the Tapis framework.

We plan to extract these middlewares and convert them into separate Python packages. These packages can then be enumerated in the workflow engine's deployment configurations and installed at runtime as part of the engine's start up procedure. The interface we design will serve as an example to other developers who need to develop their own implementations of core workflow components.

8.2 Imperative workflows

Thus far, this project has focused solely on supporting simple, declarative workflows, as conditionality of task execution is generally more appropriately handled in the application logic of a task. However, to say that taskB fails because taskA erroneously produced a result with a value that falls outside of expected range or taskB may not semantically accurate and complicates debugging failed workflows. Additionally, there are many use cases in which it would be useful for tasks to be dynamic, such as when a task performing work on a variable amounts of data produced by previous tasks. For such cases, we plan to support conditional expressions in task definitions as well as workflow-modifying functionality to be used inside of those expressions in addition to exposing the same functionality within the application logic of "function" type tasks.

8.2.1 Conditional Tasks. In some cases, a task may become rendered redundant or ineffectual by a previous task's output but not necessary for the continued successful execution of a workflow. In such cases it may be desirable to skip such a task. This use case will be supported by implementing an "if" property in conjunction with a set the special functions and operators (covered in section 8.2.2). If the conditional expression defined in this property evaluates to a boolean "true" or can be coerced into such, the task will run. If it evaluates to a false-y value, the task will be skipped. For cases in which a task is skipped and one or more tasks are dependent on it, the workflow will fail.

8.2.2 Special functions. Task definitions will be further extended to include the use of macros and comparison operators inside of the conditional expressions mentioned above. These functions would enable sub-string checking, type validation, and environment variable inclusion.

9 CONCLUSION

Tapis Workflows has demonstrated that it is a viable workflow management platform that can support a wide variety of use cases for advanced research computing pipelines in the Tapis ecosystem. Development will continue on both software features and organizational processes in an effort make the platform more general purpose and open-source friendly. In this paper, we covered the real-world scientific implementations of Tapis Workflows, the API data model, tech stack, and Workflow Engine architecture.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation Office of Advanced CyberInfrastructure, the Tapis Framework:[1931439 and 1931575] and Tuitus [2229702]

REFERENCES

[1] [n. d.]. *Decorator Pattern.* https://www.dofactory.com/net/decorator-design-pattern
[2] [n. d.]. *Gitlab CI/CD.* https://docs.gitlab.com/ee/ci/introduction/
[3] [n. d.]. *Globus.* https://docs.globus.org/api/transfer/overview/#overview
[4] [n. d.]. *HashiCorp Vault.* https://www.vaultproject.io/
[5] [n. d.]. *NEID (pronounced NOO-id).* https://neid.psu.edu/what-is-neid/

[6] [n. d.]. *Object Relation Mappers (ORMs).* https://www.fullstackpython.com/object-relational-mappers-orms.html
[7] [n. d.]. *Understanding Github Actions.* https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions
[8] 2010. *A successful Git branching model.* https://nvie.com/posts/a-successful-git-branching-model/ Last access: 2023-02-13.
[9] 2017. *OpenAPI Specification.* https://swagger.io/specification/ Last access: 2023-02-13.
[10] 2019. *Introduction to Singularity.* https://docs.sylabs.io/guides/3.5/user-guide/introduction.html Last access: 2023-02-13.
[11] 2019. *Tapis Security Kernel.* https://tapis.readthedocs.io/en/latest/technical/security.html Last access: 2022-4-1.
[12] 2019. *Tapis System.* https://tapis.readthedocs.io/en/latest/technical/systems.html Last access: 2023-2-13.
[13] 2020. *Actors.* https://tapis.readthedocs.io/en/latest/technical/actors.html Last access: 2022-3-31.
[14] 2020. *Jobs.* https://kubernetes.io/docs/concepts/workloads/controllers/job/ Last access: 2022-3-31.
[15] 2020. *Kaniko.* https://github.com/GoogleContainerTools/kaniko Last access: 2022-03-31.
[16] 2020. *Kubernetes.* https://kubernetes.io/ Last access: 2022-3-31.
[17] 2021. *The Hobby-Eberly Telescope Dark Energy Experiment (HETDEX) Survey Design, Reductions, and Detections.* https://ui.adsabs.harvard.edu/abs/2021ApJ...923..217G/abstract Last access: 2022-04-05.
[18] 2021. *Locust Documentation.* https://docs.locust.io/en/stable/what-is-locust.html Last access: 2023-02-28.
[19] 2021. *Monorepo vs. Polyrepo.* https://github.com/joelparkerhenderson/monorepo-vs-polyrepo Last access: 2022-03-31.
[20] 2021. *RabbitMQ.* https://docs.informatica.com/integration-cloud/application-integration/current-version/rabbitmq-connector-guide/introduction-to-rabbitmq-connector/rabbitmq-overview.html Last access: 2023-02-13.
[21] 2021. *Tapis UI - A Rapid Deployment Serverless Science Gateway Built on the Tapis API.* https://zenodo.org/record/5570569 Last access: 2023-02-13.
[22] 2022. *Django Web Framework.* https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Introduction Last access: 2023-02-13.
[23] 2022. *Tapis Jobs API.* https://tapis.readthedocs.io/en/latest/technical/jobs.html Last access: 2022-03-31.
[24] 2022. *Tuitus: Award Abstract.* https://www.nsf.gov/awardsearch/showAward?AWD_ID=2229702&HistoricalAwards=false Last access: 2023-02-13.
[25] Joe Stubbs et al. 2020. Integrating Jupyter into Research Computing Ecosystems. Proceedings of the Practice and Experience on Advanced Research Computing, PEARC 2020.
[26] Joe Stubbs, Richard Cardone, Mike Packard, Anagha Jamthe, Smruti Padhy, Steve Terry, Julia Looney, Joseph Meiring, Steve Black, Maytal Dahan, Sean Cleveland, and Gwen Jacobs. 2020. Tapis: An API Platform for Reproducible, Distributed Computational Research. (2020). submitted.