

TESK - A GA4GH-TES based Kubernetes batch execution service

Personal Information

<Redacted>

Motivation & Technical Skills

<Redacted>

Project Details

Brief background

The Global Alliance for Genomics and Health

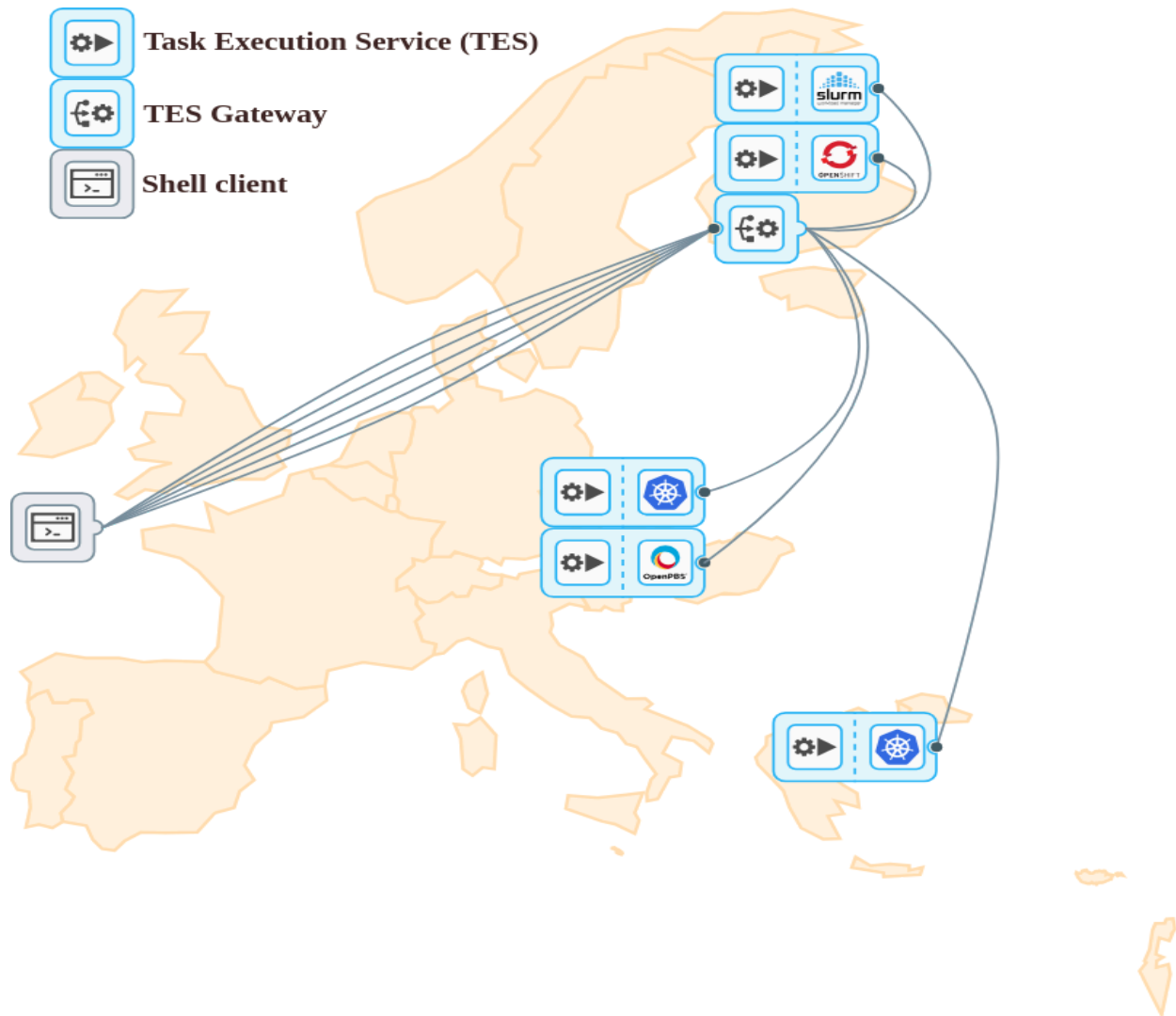
The mission of the [Global Alliance for Genomics and Health \(GA4GH\)](#) is to establish frameworks and standards for working with health-related and genomic data. Healthcare practitioners and researchers worldwide rely on this data to make informed decisions and improve treatments. Since the data, tools, and workflows involved are valuable resources, it's crucial that they can be located, accessed, and reused in an interoperable way. The GA4GH standards promote and maintain data principles across biomedical data exchange and analysis.

The project idea and its specifics

TES

The Task Execution Service (TES) API is an effort to define a [standardized schema](#) and API for describing [batch execution tasks](#). A task represents a set of input files, a set of (Docker) containers and commands to run, output files, and some other logging and metadata. It provides a standard mechanism for managing tasks' deployment, scheduling, running, and clean-up across different computing environments, including high-performance computing (HPC) systems and cloud environments. This standardization allows researchers to move their

computational workloads between different environments without needing to rewrite their code for each specific infrastructure.



The above figure shows “Task distribution via the proTES gateway.” More about proTES is here. [source](#).

TESK

An implementation of a task execution engine based on the [TES standard](#) running on [kubernetes](#). [TESK](#) can be deployed using helm charts given in the [repo](#).

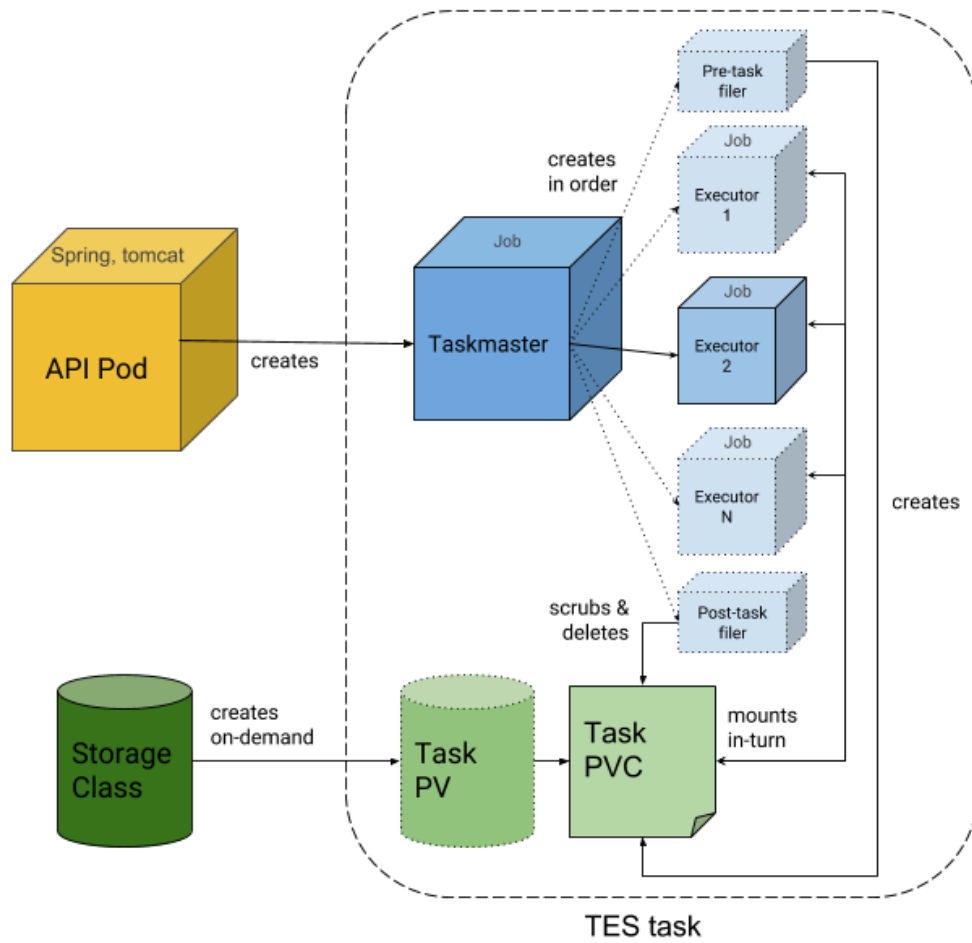
TESK - API

TESK API exposes TES-compliant endpoints. These endpoints allow users to interact with [Taskmaster](#) (k8s job that schedules TES tasks).

TESK - Core

[TESK](#) core contains two types of agents that reside in [kubernetes](#):

- The **taskmaster**, which spins up the containers needed to complete tasks as defined by TESK
- The **filer**, which populates volumes and input files and uploads output files



[Source](#)

Technical details

API

There are five endpoints of interaction:

- **GET /service-info**: Info about the services and jobs that are running.
- **GET /tasks**: Lists all the tasks ever scheduled.
- **GET /tasks/{id}**: List all info about the task whose id is provided.
- **POST /tasks**: Creates a new task with the specifications.
- **POST /tasks/{id}**: Cancel the task with the given ID.

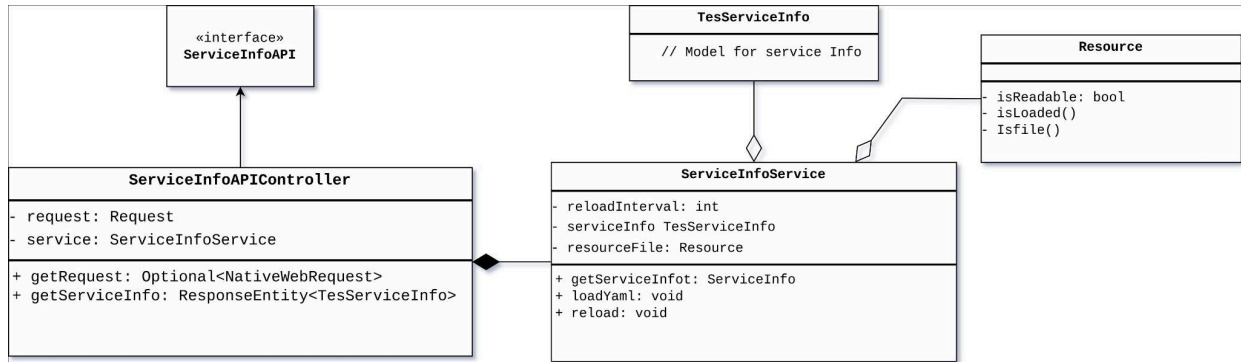
The current Java implementation has broken down the API into the following structure.

- **api**: This directory contains classes responsible for handling API requests and responses.
 - **ApiUtil.java**: Utility class for API-related tasks.
 - **HomeController.java**: Controller class for handling home-related (/) API requests.
 - **ServiceInfoApiController.java**: Controller class for handling service information API requests.
 - **TasksApiController.java**: Controller class for handling task-related API requests.
 - **ServiceInfoApi.java**: Interface defining service information API methods.
 - **TasksApi.java**: Interface defining task-related API methods.
- **exception**: This directory contains custom exception classes.
 - **CancelNotRunningTask.java**: Exception class for tasks that cannot be canceled because they are not running.
 - **TaskNotFoundException.java**: Exception class for when a task is not found.
- **model**: This directory contains model classes representing data structures used in the API.
 - **Service.java**: Model class representing a service.
 - **ServiceOrganization.java**: Model class representing the organization of a service.
 - **ServiceType.java**: Model class representing the type of a service.
 - **TaskView.java**: Model class representing the view of a task.
 - Other model classes represent various aspects of tasks, services, and their attributes.
- **service**: This directory contains service classes responsible for implementing business logic related to the API.
 - **ServiceInfoService.java**: Service class for retrieving service information.
 - **TesServiceImpl.java**: Implementation of the TES service interface.
 - **TesService.java**: Interface defining methods for TES-related services.

The current implementation delegates the API's endpoint into sections to abide by the principle of **single point of responsibility**, namely **service** and **task**.

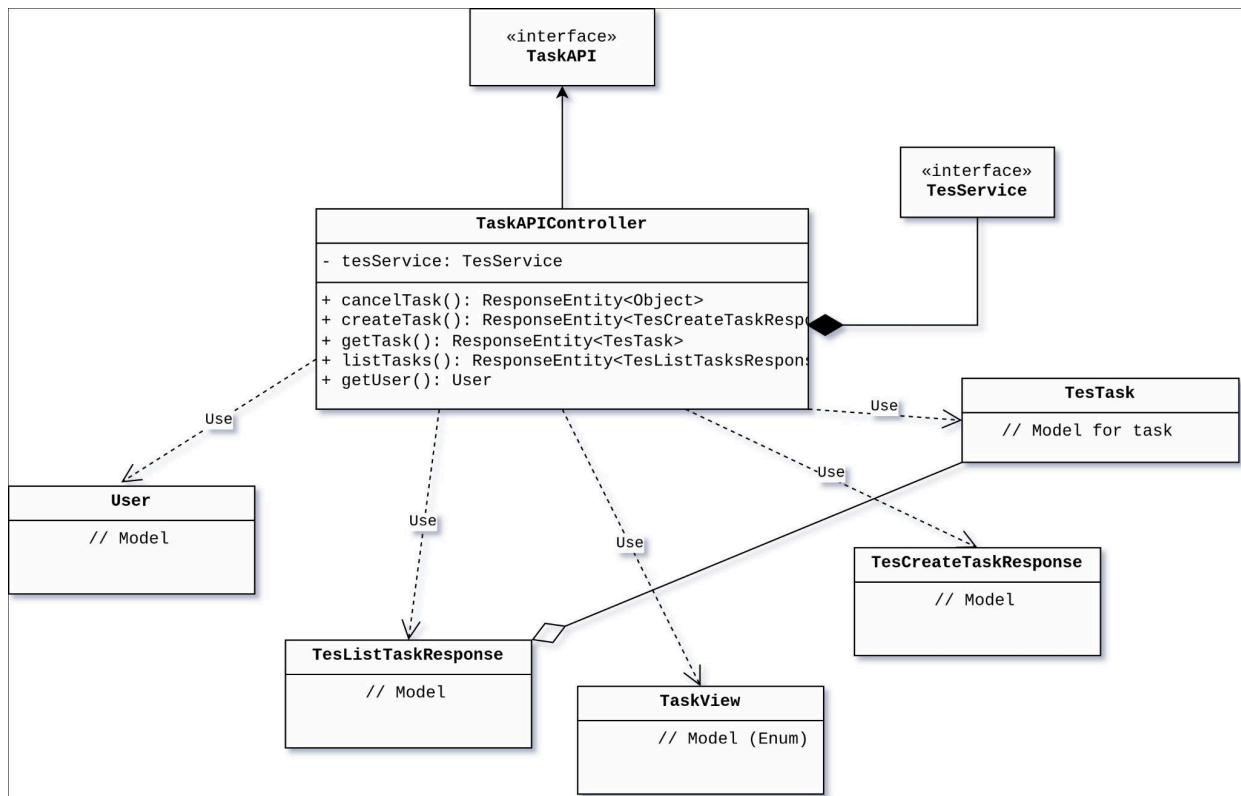
Service

The **ServiceInfoAPI** interface is implemented by **ServiceInfoApiController**, which is responsible for getting request and service information. This uses **command pattern** which uses **ServiceInfoService** as a command to instantiate **ServiceInfoApiController**, which helps in **reloading**.



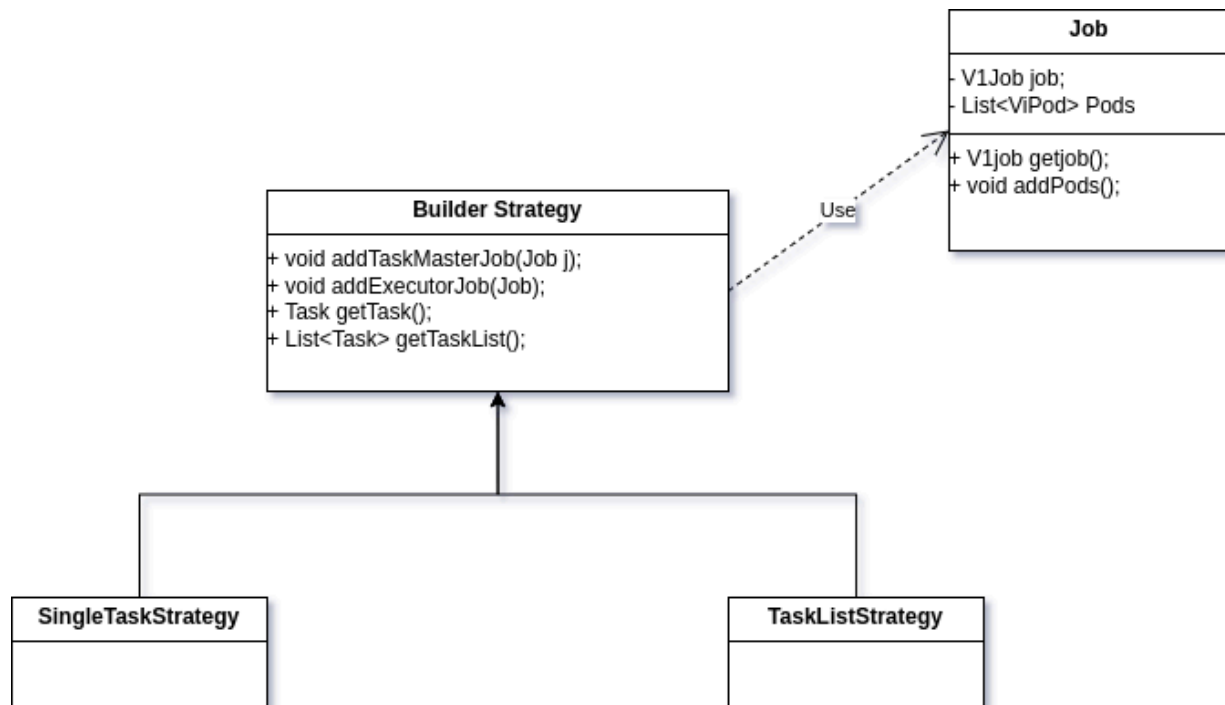
Task

TesService is the main interface that defines **GET** and **POST** endpoints and is responsible for talking to the **K8s** wrapper classes (defined below).



Kubernetes wrapper

To schedule **K8s Jobs** the Tomcat server needs to talk to the K8s client, which internally would create new tasks. In this code base, it clearly uses **builder pattern** to instantiate the classes that are required to collect **k8s pods** and **jobs**, and then based on if API needs for a single task or list of tasks to operate on, **strategy pattern** is used.



Security

Security and admin control are currently implemented using spring boots authentication, such as `@PreAuthorize` annotation, which is used for method-level security. It allows you to enforce access control on individual methods based on expressions evaluated at runtime.

K8s and core

This API server (**TomCat**) is the first pod that exposes the endpoint and talks to **task-core** which implements **Taskmaster** and **filer**.

The **taskmaster** consumes the **executor jobs**, **inputs** and **outputs**. It first creates **filer pod**, which creates a persistent volume claim (PVC) to mount as scratch space. All mounts are initialized, and all files are downloaded to the locations specified in the TES request; the populated PVC can then be used by each executor pod one after the other. After the filer is finished, the taskmaster goes through the executors and executes them individually as pods.

FOCA

FOCA (Flask-OpenAPI-Connexion-Archetype) is an opinionated archetype that enables fast development of OpenAPI-based HTTP API microservices in **Flask**, leveraging the Connexion framework.

FOCA reduces the required boilerplate code and will allow us to focus on your application logic. It also **avoids code repetition** and **introduces cross-service consistency** when developing multiple applications.

The main functionality we can leverage is the abstraction that FOCA can provide for configuring **security**, **logging**, **exception**, and **server**. This would make the code more maintainable and easier to debug since the current implementation is Java. As all the configuration can be done using **YAML**, this would also reduce testing significantly.

proTES

proTES, a proxy or middleware layer for **TES** requests, is already written using **FOCA**. It acts both as a **TES server** and **client**, and relays incoming tasks to the TES instances it knows about. This repository can be taken as an example to develop, and improving upon its code base maintenance strategy (discussed below).

Project Goals and Implementation Strategy

The **TESK** codebase is spread across multiple repositories, and this project aims to make it more maintainable and more widely used with the below-mentioned tasks:

- Replace **Java Springboot** API implementation with **FOCA-based API** implementation (similar to **proTES**).
Rationale: Rewrite the codebase in Python using the up-to-date version of its dependencies in Python to make it future-proof, sanitize deprecated code, and make it more maintainable, given **tesk-core** is written in Python. This move will facilitate interoperability, code reusability, easier maintenance and reduced overhead.
- Merge all **TESK** repositories into a single one.
Rationale: Reduce redundancy, and make management easier for **tesk-api** and **tesk-core**. It will simplify dependency management, ensuring compatibility and synchronized updates between components. Unified documentation can be maintained, covering the entire system comprehensively. Developer collaboration will be enhanced, facilitating seamless code sharing and simultaneous contributions to both components. The build and deployment process will be streamlined, with CI pipelines configured to handle both pieces. Testing will become more efficient, enabling effective integration testing to prevent cross-component issues.
- Add support for the latest GA4GH TES features by implementing **TES v1.1.0**.
Rationale: The latest version of **TES** specification has some essential features for client-side GUI components to work correctly, such as filtering. It makes sense for **Tesk** to use the latest specs to be future-proof and support all the features.

Thrust 0: Pre Project considerations and discussions

We can use [Google's Python style guide](#) to follow the best practices, and along with migration, there are certain issues that I think could be easily solved and incorporated in the migration itself.

Current Issues

- [Implementing the ListTasks filter by name prefix](#)
Specs define listing the task using the name prefix. Currently, the Web Component for TES also supports the filtering, but since the backend end doesn't have the filtering yet, there is no server-side way to do it. This issue can be solved via migration.
- [Cancel Task returning blank response](#)
The [specs](#) define the API to return the task_id that has been canceled, that is currently not the case, this can be easily solved with minor modifications to models, and especially using the [pydantic](#) models, where we can explicitly validate the data.
- [Implement logs.outputs in TesTask object](#)
Add output logs to the [TesTask](#) for better debugging.
- [Allow decimal numbers for 'cpuCores'](#)
Change the type of cpuCore to float than int that it is currently, but as the [specs](#) define it be [int64](#), this needs to be discussed.
- [TESTask logs.logs.exit_code not present in some cases](#)
Add exit code logs to the [TesTask](#) for better debugging.

Note: Some of the issues open are also a part of TES v1.1.0, e.g. number of cpu cores and filtering.

Project-wide configurations

There are some new changes that will need to be revised and configured for the entirety of the repository, which will be done in pieces while migrating from [Java](#) to [Python](#). Then, after the merge, there will be changes and additions.

Package management

We can use [uv](#), which is multiple folds faster than [poetry](#). But since [uv](#) is relatively new, there are issues in it, and it's not as stable as some of its alternatives. But instead of using [pip](#) or [pipx](#), [poetry](#) would be beneficial as it would improve DX and CI performance. We can track and manage dependencies better using [pyproject.toml](#).

Type checking

As Python is not a statically typed language and we would need to type annotations to avoid problems, I suggest using [mypy](#), to type annotate, which would make the code less error-prone and help catch bugs faster. However there can be other alternatives as well, such as [Google's pytype](#), [Microsoft's pylance](#) or [Pyre](#).

Formatting

Instead of having multiple formatting and linting tools as [proTES](#) does right now such as [black](#), [pylint](#), [flake](#) etc we can use [ruff](#) which is magnitudes faster than those as mentioned earlier. Having a single dependency will also reduce the number of configs.

Testing

[Pytest](#) can be used to test python code. This will help us generate coverage reports which can be then used to track coverage for unit tests using [Codecov](#) and try to achieve near 70% coverage to ensure that most of the logic is covered.

Continuous Integration

For continuous integration (CI), we can use [GitHub Actions](#) as it's easier to set up and native to Github. Looking at [other OSS workflows](#), we can see that below are the most common and essential checks that we can/should include.

- [Smoke test](#): To check that all the endpoints are healthy after code change.
- [Unit test](#): To check if all unit tests pass.
- [Integration test](#): To check if all the integration tests pass.
- [Lint and format check](#): To check the code, adhere to the code standard defined.

Thrust 1: Migration

API Strategy

Models

There won't be changes in the models and exceptions, so these components can be easily migrated. [Pydantic](#) models can be created for each of them.

- [Service](#): GA4GH service.
- [ServiceOrganization](#): Organization providing the service.
- [ServiceType](#): Type of a GA4GH service.
- [TaskView](#): View type, minimal, basic etc.
- [TesCreateTaskResponse](#): Describes the response from the CreateTask endpoint.
- [TesExecutor](#): Executor describes a command to be executed, and its environment.
- [TesExecutorLog](#): ExecutorLog describes logging information related to an Executor.

- **TesFileType**: Gets or Sets tesFileType.
- **TesInput**: Input describes Task input files.
- **TesListTasksResponse**: Describes a response from the ListTasks endpoint.
- **TesOutputFileLog**: OutputFileLog describes a single output file.
- **TesOutput**: Task output files.
- **TesResources**: Resources describes the resources requested by a task.
- **TesServiceInfo**: Info for the TES instance.
- **TesState**: Task state as defined by the server.
- **TesTask**: Task describes an instance of a task.
- **TesTaskLog**: TaskLog describes logging information.

View

Foca configuration can bootstrap the API with swagger documentation, security, logging and a server. This will give us a definite structure for the controllers as they will be defined in the specs (swagger).

Controller

With models and a boilerplate ready, controllers can be migrated.

- **ListTasks**: List all the tasks.
- **ListTask**: List the task with the given ID, with minimal, basic etc view.
- **CreateTask**: Create a task.
- **CancelTask**: Cancel the task.
- **ListService**: List service info.

Using **FOCA** we can implement all the endpoints mentioned above of the API server. Foca's YAML configuration can be leveraged to implement standardized org-wise best practices for api, exceptions, jobs, log, security, and server.

Tests

Unit tests: Write unit tests to ensure that individual components of your Flask application work as expected.

Integration tests: Test the integration of your Flask API with other components such as databases or external services.

Load/Performance Tests: Use tools like **Locust** with **pytest-benchmark** to simulate load and measure the performance under different conditions.

Contract tests: Tests to ensure that the API contracts remain consistent.

Challenge: K8s client upgrade

Moving from Java to Python the client used for K8s will change, and since current implementation uses an older version of the client various API's have been changed and are [no longer backwards compatible](#). This will require changes in the codebase to accommodate for those changes, making the migration more difficult than a one-to-one language translation of the project.

Solution

Since these clients have a wrapper class and implement an interface, the code changes won't break the entire application just the business logic of the class, this will help isolate the changes to be made and outline the deprecated features and APIs, find solutions to it in the new version and implement the changes.

Thrust 2: Merging repos

All the different components can be merged and developed in the same repository such that it is more maintainable, and each different component is not released separately which makes **TESK** hard to maintain. All **core** and **api** can be bundled together as module and helm chart and manifest files can be kept in a deployment directory like it's been done in [other projects](#).

Preserving history

We can merge repositories with some restructuring to avoid conflicts, **TESK** repositories can be restructured by putting all the files in a new directory called packages (say). This directory will hold all other components. The other two components, namely **tesk-api** and **tesk-core** can be bundled into a similar directory structure.

Once all the repos are configured, we can merge their **main** branches with the flag **--allow-unrelated-histories** to preserve histories.

```
git merge app-b/merge-into-a --allow-unrelated-histories
```

Note: Even though this is demonstrated for *Monorepo*, a similar approach can be adopted for our use case.

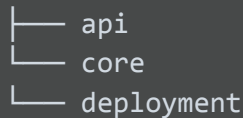
Credit: [Source](#)

Previous structure

```
├── TESC
├── tesk-api
├── tesk-core
```

New structure

```
├── TESC
```



```
├── api
├── core
└── deployment
```

Changes in pipeline

The merge will introduce the need to change CI pipelines as well. Since all the projects had their own CI, after merging common parts of their CI/github-workflow would need to be merged and new workflows to be created for each component's individual need, i.e. `tesk-core`, `tesk-api` and `deployment`.

Maintainability

Having one repository as different packages will offer several advantages. It will simplify dependency management, ensuring compatibility and synchronized updates between components. Overhead associated with managing multiple repositories will be reduced, leading to simpler project management. This will promote code sharing and reusability, fostering a modular and efficient codebase. Issue tracking and bug fixing will be simplified, ensuring consistent improvements across both components.

Challenges

Ideally, keeping git history preserved, there might will be huge and might pose difficulty, but we can create an extra commit for each package, that would refactor and rearrange the code base to accommodate the new structure, this will allow us to merge the repository into one without loss of history and numerous conflicts.

Thrust 3: Implementing TES v1.1.0

Certain changes are obvious and very easy to implement, such as adding a new task state `preempted` and a `canceled` state, changing `page_size` and `cpu_count` from `int64` to `int32` etc. These changes should not change business logic or introduce new ones.

While other changes, such as `filter by status` feature, would require the introduction of new business logic or at least a data sanitization middleware. Upgrades as such would require discussion and mentor advice.

Changes

feat: add `backend_parameters` to `tesResources` by in [#154](#)

This will extend `tesResources` to support an arbitrary dictionary of additional properties.

feat: add in new `tesState` for `preempted state` in [#184](#)

Implementation to return a pre-empted message to retry with a dedicated machine

feat: allow wildcards in output paths in [#185](#)

allow clients to specify pathname matching wildcards ("globs") when specifying task outputs.

imp: making file type an optional argument in [#155](#)

fix: changing int64 to int32 for page_size and cpu_count by in [#175](#)

Change the size of page_size and cpu_count to comply with specifications.

imp: add ignore_error flag in [#159](#)

Add an ignore_error field to the caller to define with executors to allow it to fail, but still continue the chain.

feat: add CANCELING state to delete cloud resources in [#189](#)

Add a new TES state: CANCELING. When a caller calls the CancelTask operation, TES shall set the task's state to CANCELED if and only if all dependent/downstream resources have been deleted.

feat: add streamable flag to tesInput in [#157](#)

Provide a flag that indicates that a file could be mounted via a streaming interface, if it is available, without causing performance degradation to the code being deployed.

feat: add filtering features to task listing in [#170](#)

Filter on tags and/or state

Note: The descriptions are meant as context, please look into the issues for more clarity.

Stretch Goals

Artifact Hub

There are Helm charts for deployment in the repo but it would make sense to publish them on a public repository like [artifacthub](#). Or maybe discuss a solution like hosting artifacts using [harbor](#) which would give more control over the artifacts and help us secure artifacts with policies and role-based access control, ensuring images are scanned and free from vulnerabilities

CI and release

With the new architecture, there will be a need to test for different packages, one test would be to lint and test manifest files and Helm charts. Using [kubeval](#) we can test if the charts are semantically correct. With tests in place and artifact registry configured, we can use [chart-releaser](#) to push the latest artifacts, keeping them up to date.

Milestones

1. Setup env
2. Setup Foca: server, security and logging
3. Migrate endpoints as POC
4. Merge `tesk-core`
5. Create `pydantic` data models and data validations
6. Migrate `K8s` wrapper
7. Migrate service and controllers
8. Upgrade to `TES V1.1.0`
9. Update helm charts and docker images
10. Merge deployment charts and add unit tests
11. Write documentations

Timeline

Bonding period: Discuss priorities and execution plan, as well as different approaches and ideas.

Week 1: Setting up the environment.

Create a repo or start with one of the earlier repositories such as `tesk-core` or `TESK`.

Setup package manager, linter and other repository management tools and github workflows.

Week 2: Setup FOCA, studying best practices FOCA examples.

Configure Server, openAPI, security and logging.

Week 3: Setup task and service endpoints.

Week 4: Merge repositories or add remaining components, i.e. if starting with a new repo, add both `tesk-core` and `TESK`, but if starting with one of them as base repositories, add the other component.

Week 5: Migrate data models and add validations. Many data models, such as `TesTask`, will be language agnostic, so all of those can be migrated simultaneously.

Week 5 – Week 6: Migrate K8s wrapper and logic using Python's Kubernetes client library.

Week 7 – Week 9: Set up controllers for business logic of endpoints.

Week 9 – Week 10: Upgrade to TES v1.1.0 and update deployment files, like Helm charts and dockerfiles.

Week 11: Add unit tests.

Week 12: (Left blank to account for any cleanup, modifications, and delays.)

The above timeline is subject to change as per discussion and mentor advice.