Felucca

MITS CMU

Architecture Document

Project Felucca

Team BugHunter

Authors

Di Mu (dimu@andrew.cmu.edu)
Sudi Lyu (sudil@andrew.cmu.edu)
Zihao Zhou (zihaozho@andrew.cmu.edu)
Guancheng Li (guanchel@andrew.cmu.edu)

Revision History

Version	Date	Change	Updated by	Reviewed by
0.1	Mar 21st, 2020	Construct overall architecture	Sudi Lyu	
0.2	Mar 27th, 2020	Reconstruct overall architecture based on pharos toolset requirement	Sudi Lyu	
0.3	Mar 29th, 2020	Add glossary	Sudi Lyu	
0.4	Apr 12nd, 2020	Add time sequence diagram	Guancheng Li	Jeffrey Gennari
0.5	Apr 24th, 2020	Update detailed flow diagram and time sequence diagram to unify the glossary	Sudi Lyu, Guancheng Li	Hasan Yasar
0.6	Apr 30th, 2020	Redraw all flow diagram using Lucidchart	Sudi Lyu	
1.0	May 1st, 2020	Reformat, add more description and technical stack	Di Mu, Sudi Lyu, Guancheng Li, Zihao Zhou	

Contents

1 Introduction

1.1 Document Purpose

This is an architecture design document for a **control center** that manages the execution of Pharos tools and its output result. This document illustrates the objectives of this architecture and detailed introductions of every module inside, elaborating how this architecture fulfills functional and non-functional requirements with modularity. Moreover, this document also illustrates the interactions and control/data flow between modules and layers to provide a clearer view.

1.2 Architecture Objectives

Felucca is a control center that manages the execution of Pharos tools and its output result. It abstracts the executions of Pharos toolset into the **task** concept and uses the **job** concept to represent the collections of tasks. By using this abstraction, it provides functions of job submission that enable users to submit a job to Felucca and Felucca execute it with users' specification in a background way. Users could check the status of this job or manage it during the execution and would be able to export the execution result from Felucca after it's done. Users could also upload their customized tools to Felucca and let Felucca manage the jobs using these customized tools, while developers could also update the pharos tools in Felucca to let all the jobs in the future could use the latest version of Pharos toolset.

1.3 System View

The whole architecture contains 3 components: **Felucca control center**, **Pharos toolset**, and **data storage**. The users only interact with the Felucca control center to submit and manage their jobs, while the pharos toolset is managed by Felucca. The data storage is responsible for persisting data we need including job metadata, job output file, and also tools uploaded. By interacting with Felucca, the users could use the Pharos toolset with a better UI by submitting jobs to Felucca and check the output from it.

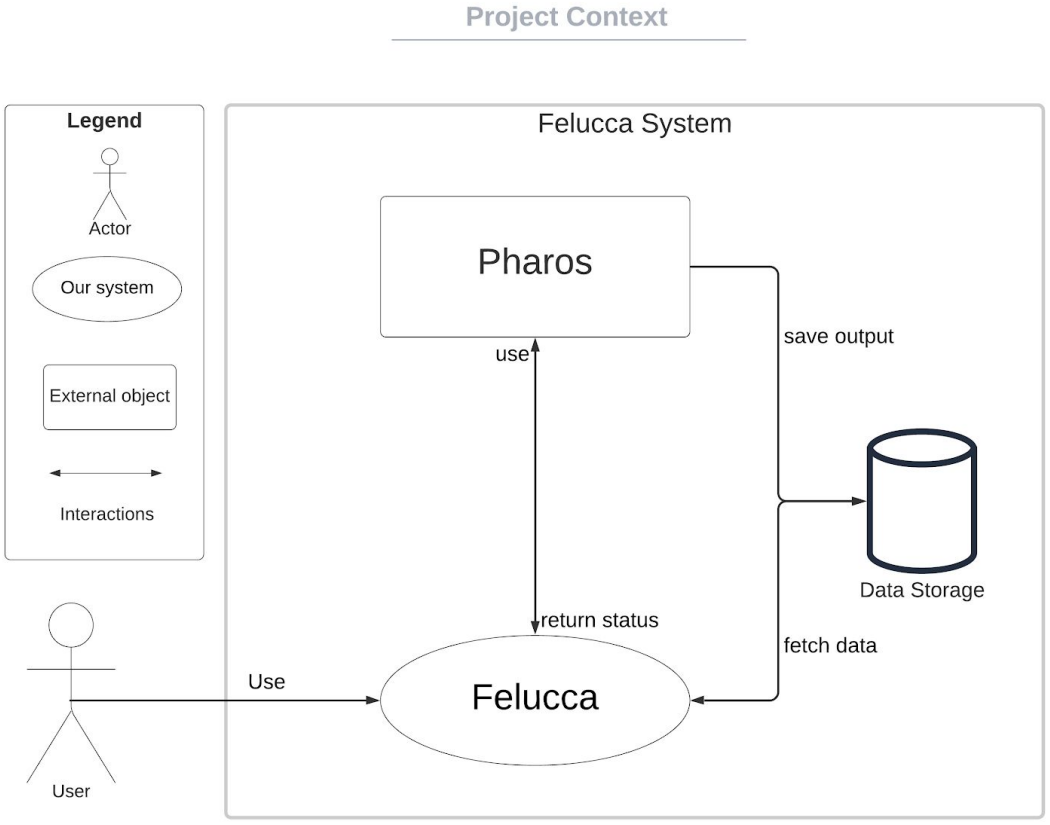

Figure 1.1 The system view of Felucca control center

2 Glossary

- **Binary:** a binary is an executable file needed to be analyzed using tools, it is the input of tools
- **Tool:** a tool is a program which inputs a binary with arguments and then generates several outputs, it could be tools in Pharos or customized tools from users
- **Pharos tools:** the original pharos toolset provided by the developer, it contains multiple tools
- **Customized tools:** A customized tool provided by users, users must provide program and arguments information and output information
- **Task:** a task is an execution of a single tool with a single binary executable
- **Job:** a job is a collection of tasks and the relationship of tasks, the tasks in jobs need to be executed by some pattern determined by their relationships, the job is finished only if all the tasks are finished in a good manner
- **Status:** if a task is finished or not
- **Outputs of tools**:
    - **Log:** the log file generated by tools, the inner information of the execution
    - **Output(file):** the output file generated by tools, the outcome of the execution
    - **Command-line output:** the output of standard output of tools, explicit information of the execution

3 Module Introduction

3.1 Module Diagram

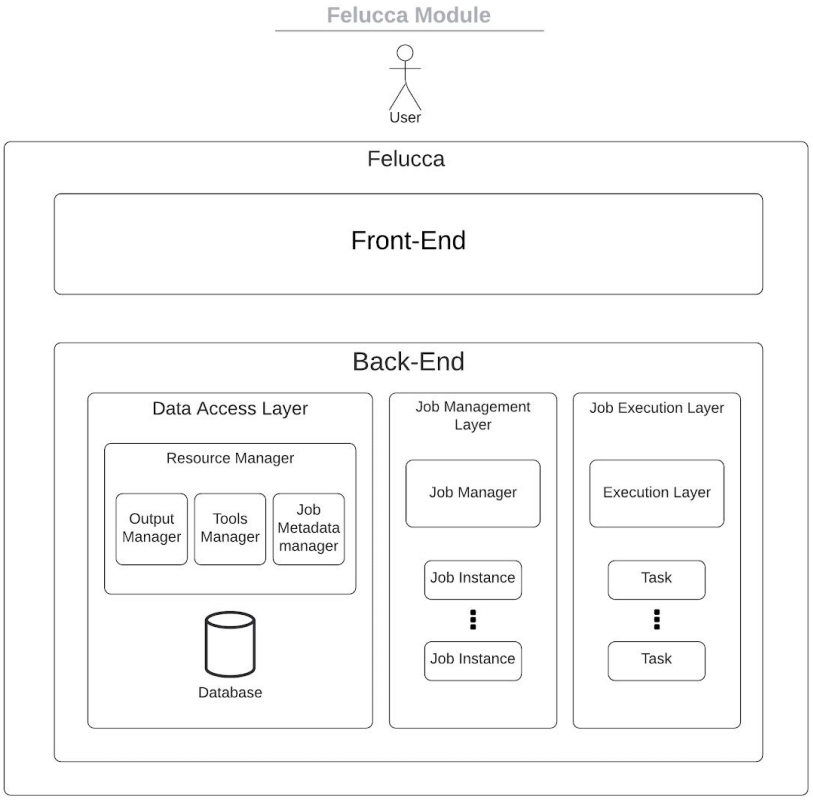

Figure 3.1 The Modules inside Felucca control center

3.2 Front-end

The front-end of Felucca architecture provides a user interface and communicates with the back-end layer.

3.3 Back-end

The back-end of Felucca architecture provides the functionality of job creation and management, tools management, and output file management. In the meantime, it decouples its control flow with Pharos toolset to support further versions of Pharos toolset.

- **Data Access Layer:** Provide data storage for tools, job metadata, and output for front-end and Job Execution Layer.
- **Job Management Layer:** Job Management Layer manages job execution logic and submits tasks to the Job Execution Layer.

- **Job Execution Layer:** Job Execution Layer is responsible for executing and monitoring tasks, returning the task's output to the Job Management Layer, and store outputs in the Data Access Layer.

4 Workflow

4.1 Overall Workflow

The overall workflow of Felucca is shown in Figure 4.1.

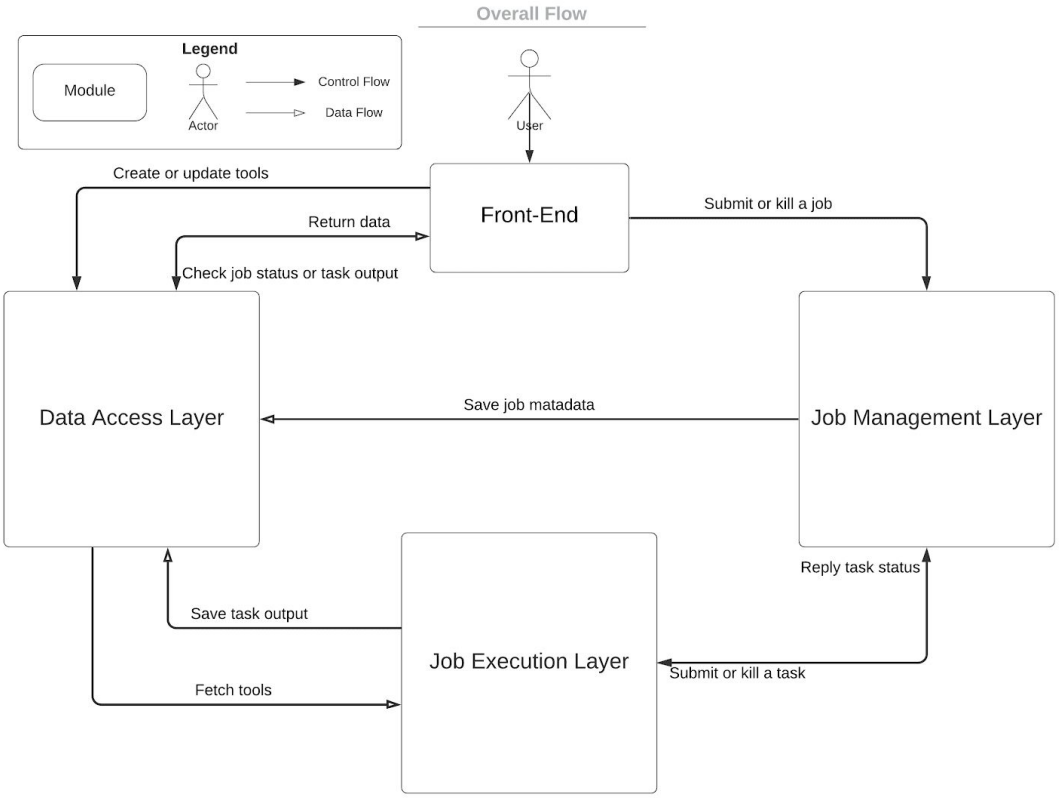

Figure 4.1 The overall flow diagram of the Felucca control center

4.1.1 Front-end

- In job management logic, the front-end encapsulates the users' arguments and binary of each job and submits it to the Job Management Layer and it also sends the kill signal of jobs to the Job Management Layer if users want to kill a job.
- Front-end also interacts with the Data Access Layer to fetch the job metadata and output for users.
- When users create or update tools, the front-end also saves it to the Data Access Layer for further use.

4.1.2 Job Management Layer

- Job Management Layer creates or kills jobs according to requests from the front-end.

- Job Management Layer manages jobs and submit requests to the Job Execution Layer to execute tasks inside each job. Job Management Layer is also responsible for tracking and status of jobs and tasks.
- Job Management Layer also needs to store job metadata to the Data Access Layer for users to check.

4.1.3 Job Execution Layer

- Job Execution Layer is where the task is executed. For every task request received, the Job Execution Layer would fetch tools from Data Access Layer and execute the task using the binary and arguments along with the fetched tools.
- After tasks finish, the Job Execution Layer returns its status to the Job Management Layer and saves its output to the Data Access Layer.

4.1.4 Data Access Layer

- Data Access Layer stores the job metadata and task output for users to check using the front-end.
- Data Access Layer also persists the tools for the Job Execution Layer to use in future tasks.

4.2 Job Management Layer Workflow

The workflow of the Job Management Layer is shown in Figure 4.2.

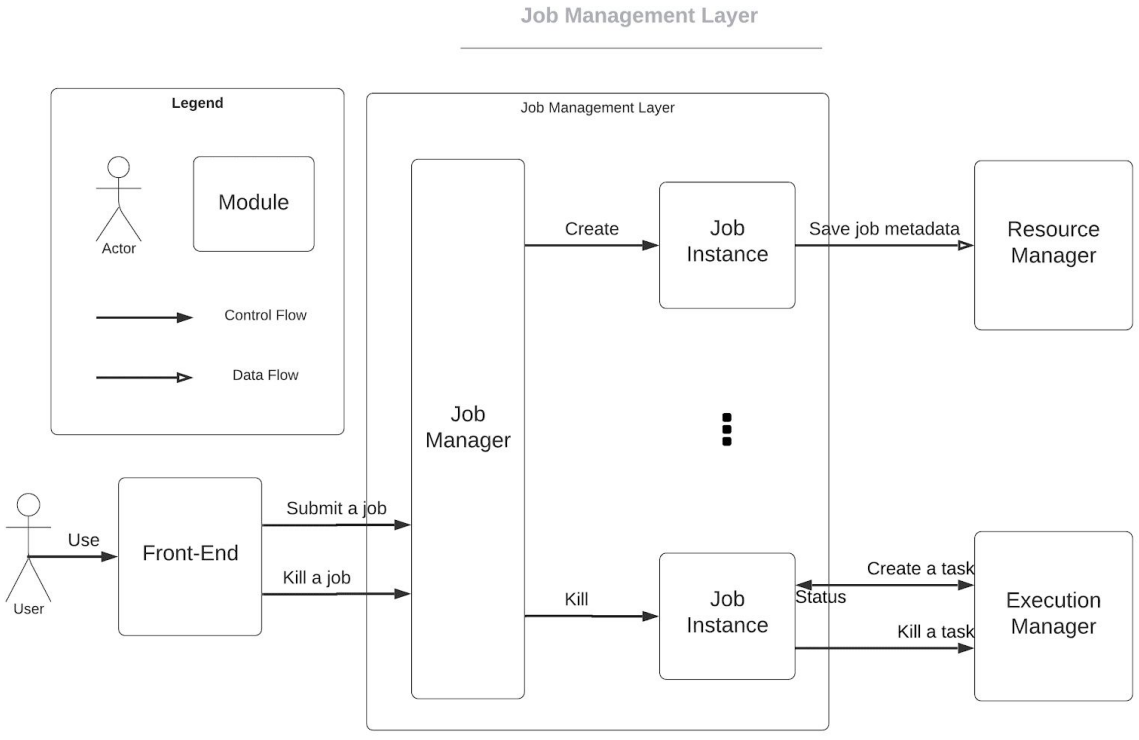

Figure 4.2 The flow diagram of Job Management Layer

4.2.1 Job Manager

The Job Manager behaves like a **manager** of Job Instance and it receives the instructions from the front-end. For each job, a Job Instance is required to manage the execution of all tasks in this job. When the front-end submits a job to the Job Manager, a job instance is created and managed by the Job Manager. Job Manager could also kill a Job Instance by giving it a signal to let it kill the running tasks and cancel all remaining tasks.

4.2.2 Job Instance

A Job Instance is created by the Job Manager and it will add its metadata into Resource Manager. Then, this Job instance will start to interact with the Execution Manager to create tasks. The Job instance is responsible for managing the order of execution of tasks to fulfill the potential user's requirements. After a job finishes, its Job Instance needs to update job metadata. So, there will be multiple Job Instances frequently interacting with Resource Managers to add new jobs or update the status of jobs.

4.3 Job Execution Layer Workflow

The workflow of the Job Execution Layer is shown in Figure 4.3.

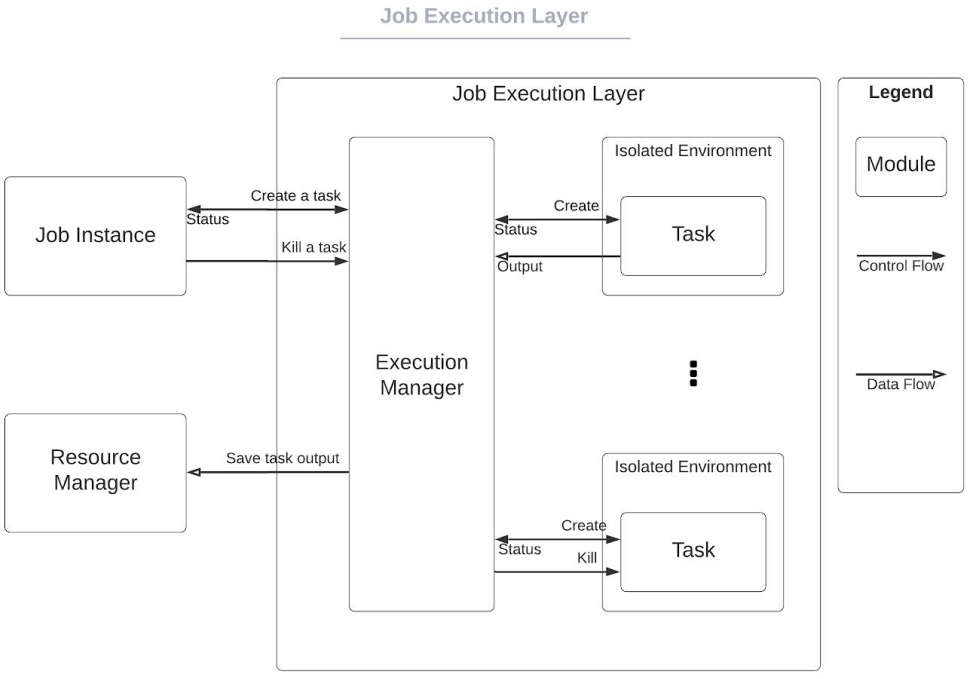

Figure 4.3 The flow diagram of Job Execution Layer

4.3.1 Execution Manager

The Execution Manager is responsible for executing and killing tasks, monitoring the status of current tasks, and also saving the output to the database. For every submitted task, the Execution Manager keeps track of its status and retrieves the output from it. So, it needs to interact with the Resource Manager to get the binary and tools information to execute the task and also save all the outputs.

4.3.2 Task

Tasks are created by the Execution Manager and it's executed in an isolated environment to contain the damage inside. Its status is monitored by the Execution Manager and all output is returned back to the Execution Manager.

4.4 Data Access Layer Workflow

The workflow of the Data Access Layer is shown in Figure 4.4.

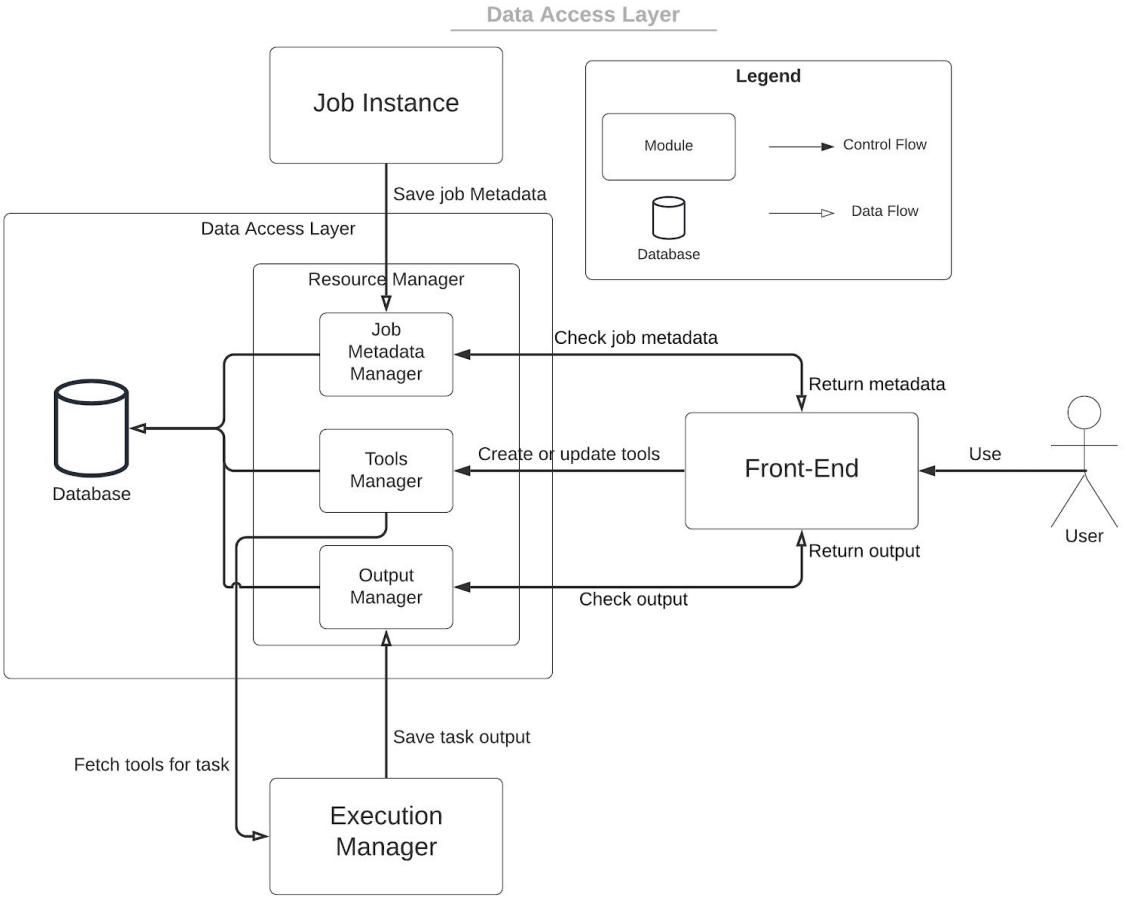

Figure 4.4 The flow diagram of the Data Access Layer

4.4.1 Resource Manager

Resource Manager manages all persistent data in Felucca, it consists of three components.

- Tools Manager
- Job Metadata Manager
- Output Manager

4.4.2 Tools Manager

Tools Manager is responsible for tools update and fetch (both Pharos and customized), it receives tools from the front-end updated by users and the tools pulled by the Execution Manager to execute tasks.

4.4.3 Job Metadata Manager

The Job Metadata Manager is responsible for updating and fetching metadata of each job. Job metadata is saved to the database when Job Instance updates its metadata, and the front-end will fetch the job metadata for users.

4.4.4 Output Manager

The Output Manager is responsible for persisting the output of each executed task. The Execution Manager would save all the outputs to the Output Manager and the users could fetch them using the front-end.

4.4.5 Database

The database provides reliable storage for all the components.

5 Time Sequence Diagrams

Combining with the architecture design above, all of our user case requirements could be converted into workflow between the modules. This time sequence diagram provides a clearer view of how our architecture supports our user cases.

5.1 Submit a Job

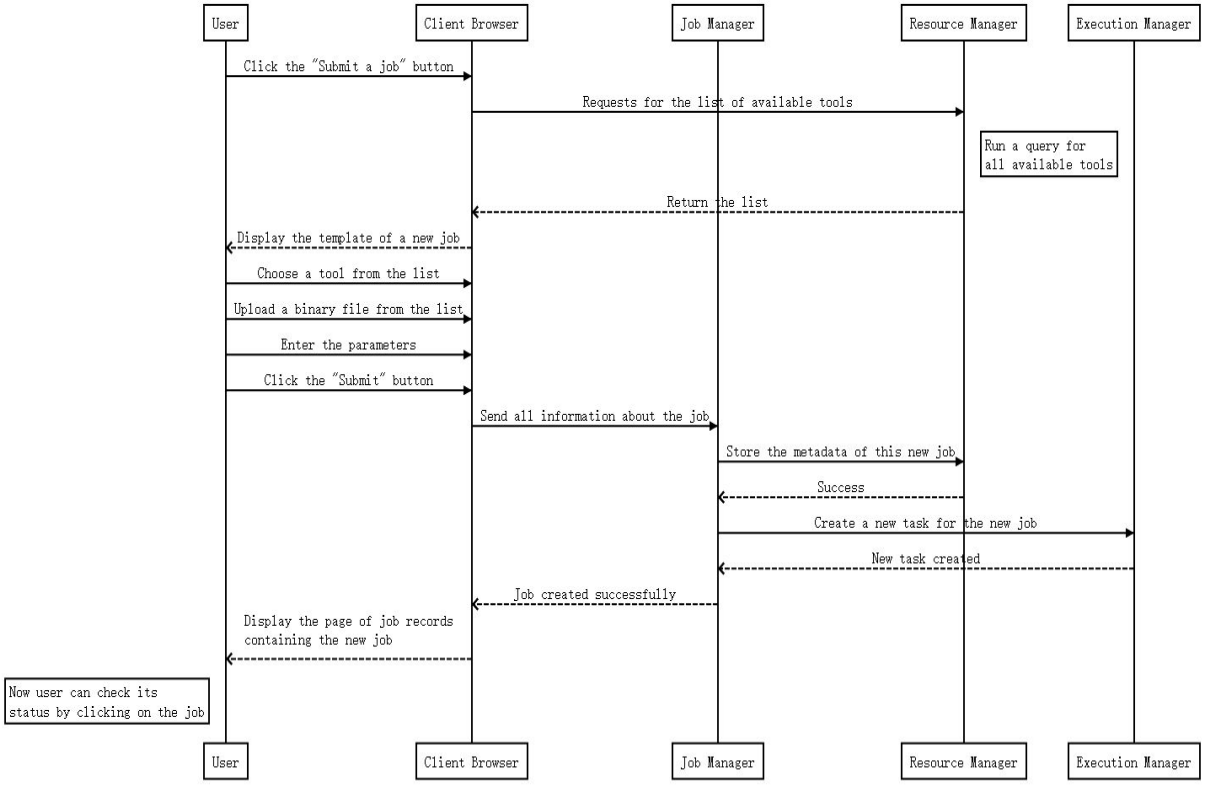

Figure 5.1 The time sequence diagram of submitting a job

Submit a single-task job, multi-task job or chained-task job have similar logic, some of the steps will be repeated for multiple tasks.

5.2 Check Job Status & Check Output/files of a Job

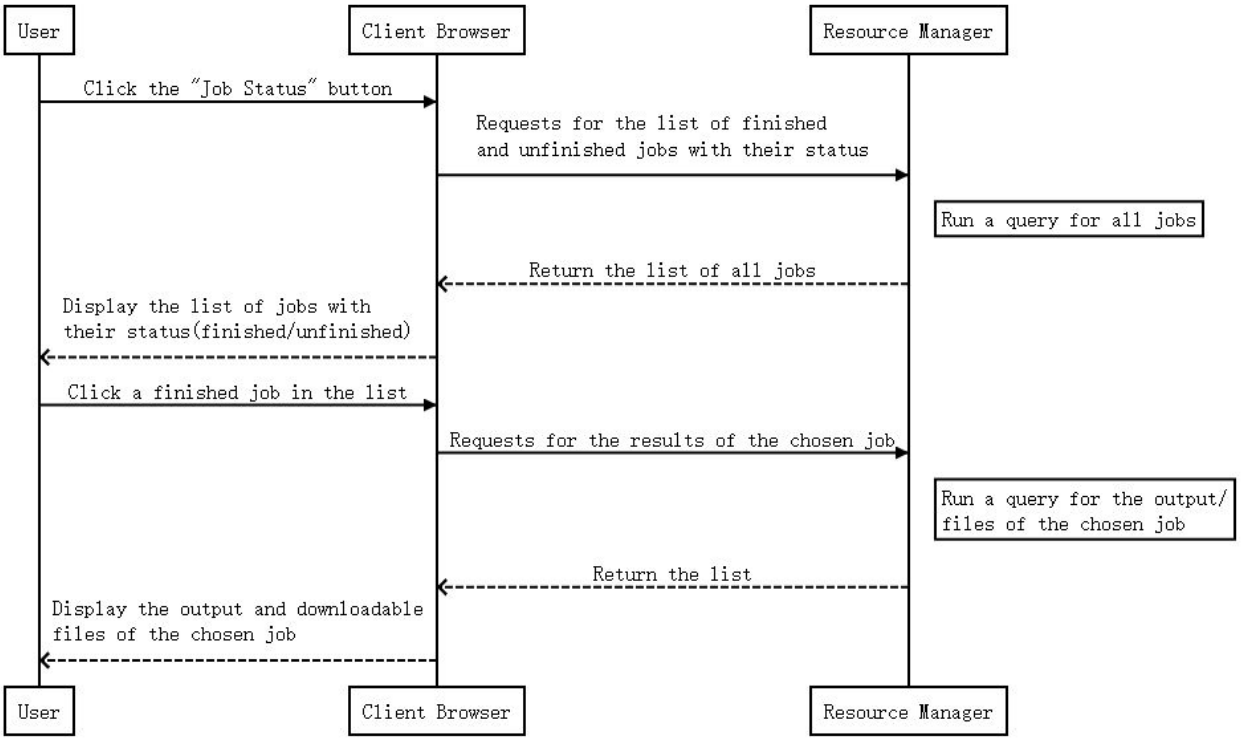

Figure 5.2 The time sequence diagram of check job status and output

5.3 Add a Customized Tool

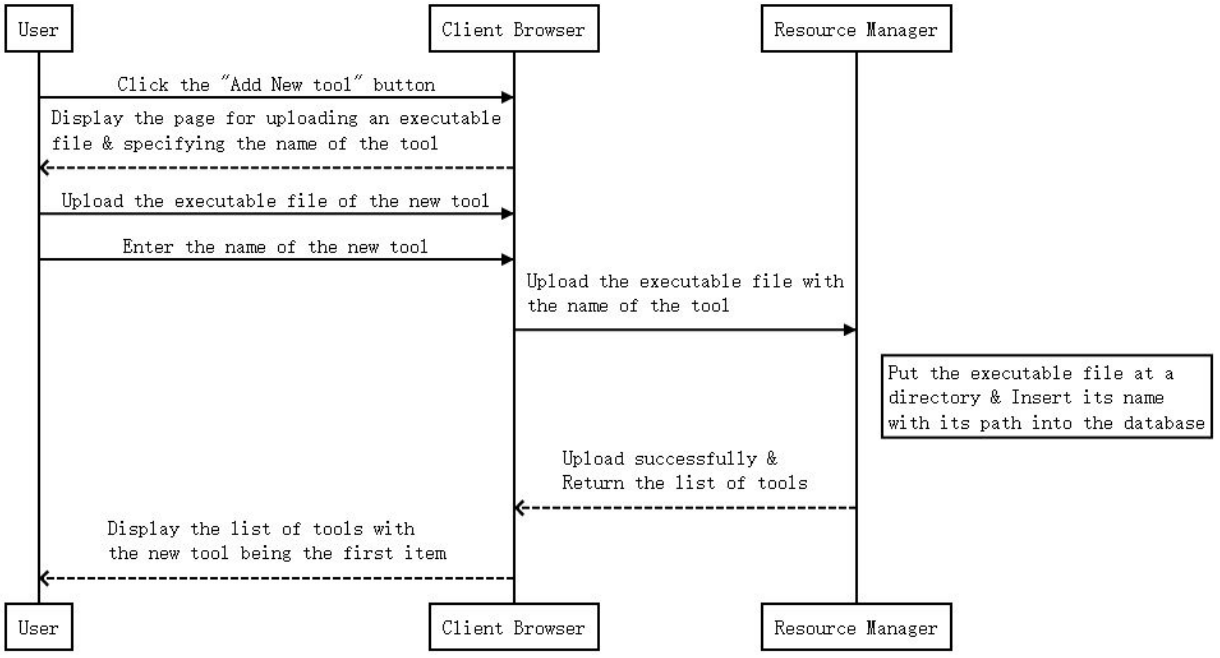

Figure 5.3 The time sequence diagram or add a customized tool

5.4 Update a Customized Tool

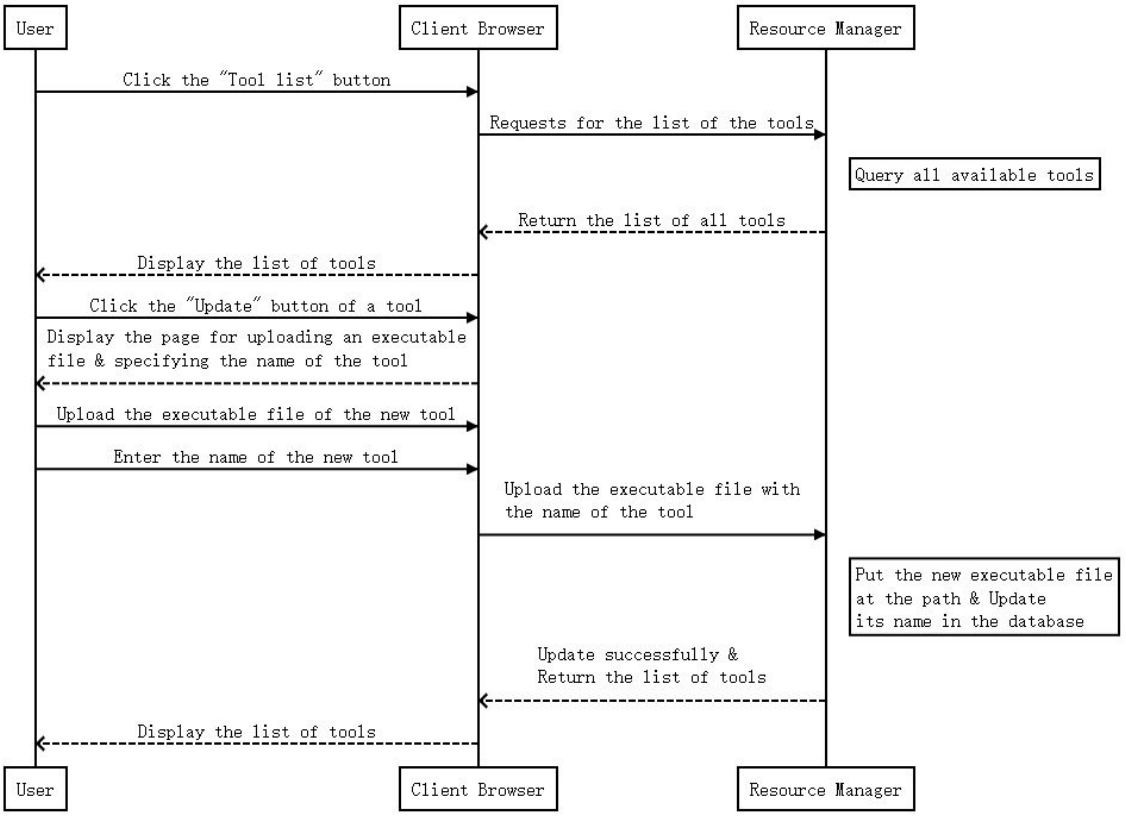

Figure 5.4 The time sequence diagram of update a customized tool

5.5 Remove a Customized Tool

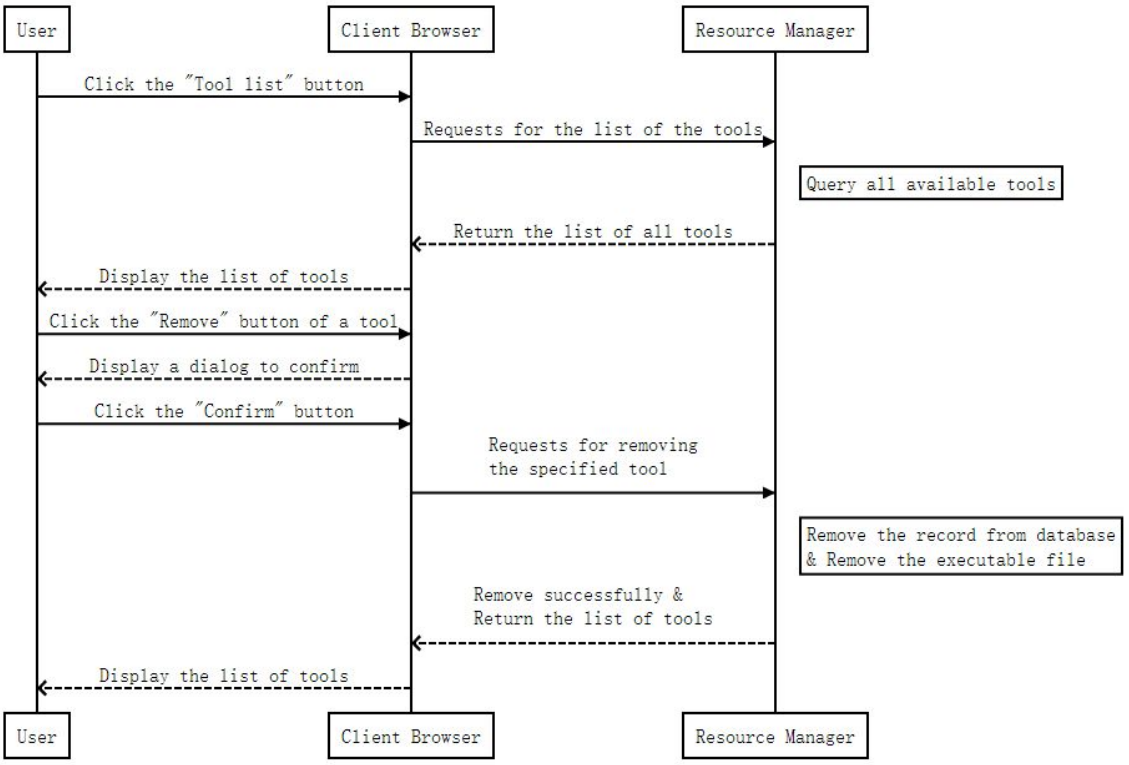

Figure 5.5 The time sequence diagram of remove a customized tool

6 Technical Stack

This section contains the technical stack we explored and chose to implement each module in architecture. For each part, we explored several options and also considered their pros and cons. After we analyzed each option, we chose the best options we thought for this project.

6.1 Front-end

6.1.1 Bootstrap

Bootstrap is an open-source Front-end library for website and web application. It contains several components like grid, button, sliding bar, forms, etc. It has its own library for CSS, HTML, JavaScript frameworks to provide comprehensive functionalities of these components. It can support many browsers and have good documentation and community support.
In this project, we will choose to use this framework, because it can satisfy all our UI requirements, and some of our developers have previous experience on this so it will be easy to use.

6.1.2 AngularJS

AngularJS or Angular2 is a Front-end framework developed by Google. It has many popular features like MVC(model-view-controller), data binding, dependency injection, and so on. It simplifies application development by presenting developers with a higher level of abstraction. It has many comprehensive components which have a great abstraction and can handle many user requirements.
In this project, we will choose to use this Front-end framework because it has a good performance. The component-based architecture for creating the UI components will be easy to use and maintain. Also, some of our team members have previous experiences on these frameworks, so it will shorten our learning curve.

6.1.3 React

React is also a popular Front-end framework created by Facebook. Different from the MVC framework, it contains an innovative idea which results in higher performance and simple code logic. Also it will make good use of re-using the React components which will make the code easy and clean.
In this project, we will not use this framework. Since none of us have previous experience on this so it requires a long learning curve. What's more, due to high pace development, it lacks comprehensive documentation which will make it difficult to get started.

6.2 Back-end

Both Flask is a Python-based framework, which is easy for us to code. Therefore, we decided to choose one from them to be our back-end.

6.2.1 Flask

Flask is a simple, lightweight, and minimalist web framework, which is easy for us to get started and suitable for small websites even if we don't have much experience in web development. However, it lacks some of the built-in features but developers can grab tools from the community.

6.2.2 Django

Django is a full-stack Python web framework. The existing built-in tool in Django helps developers to build a variety of web applications without using third-party tools and libraries. However, it includes functional modules that are not needed for some lightweight applications, and many classes and methods are encapsulated and it is difficult to change them.

Both two back-ends discussed above can be used for our project. Because Flask is lightweight and some of our group members have Flask development experience, we decided to use Flask as our back-end.

6.3 Database

After careful consideration, we decided to use **MongoDB** as our database, instead of some relational databases like MySQL. There are several reasons for that.

1. First of all, our data is pretty simple and doesn’t require complex schemes. Thus a non-relational database is enough for our requirement.
2. Moreover, using relational databases requires extra SQL skills. Some of our team members have not used SQL before. Meantime, the interfaces provided by MongoDB are more intuitive than the one of MySQL.
3. Besides, we have decided to use Django as our back-end, which is Python-based. And Python is good at manipulating JSON files, which are used by MongoDB to store information. Using MongoDB with Django can provide more convenience.
4. Furthermore, MongoDB is more efficient than relational databases, in the presence of simple data schemes. This can bring higher concurrency to Felucca.
5. Last but not least, MongoDB supports flexible data types. For example, data of different types can be stored in the same column. Relational
6. Databases are not capable of this since the scheme of a table is pre-defined.

6.4 Execution Environment

To manage software dependency and also meet the QA-1 attributes. An execution environment management is required to create an isolated environment for each execution

with Pharos toolset dependency. Two options for this are Docker Container and Virtual Machines.

6.4.1 Docker Container

Docker provides a standard unit of software that packages up code and all its dependency, so it is lightweight and fast to boot. It also provides some degree of isolation, but the containers in docker are sharing the same OS. Docker starts with the Linux operating system, but now it has a generalized version in windows, too.

6.4.2 Virtual Machines

Virtual machines provide full hardware virtualization that provides more freedom for users with higher authority. Virtual machines also provide higher isolation levels. But to fully boot up a virtual machine has considerable overhead. A comparison between Docker and Virtual Machines is shown in Figure 6.1.

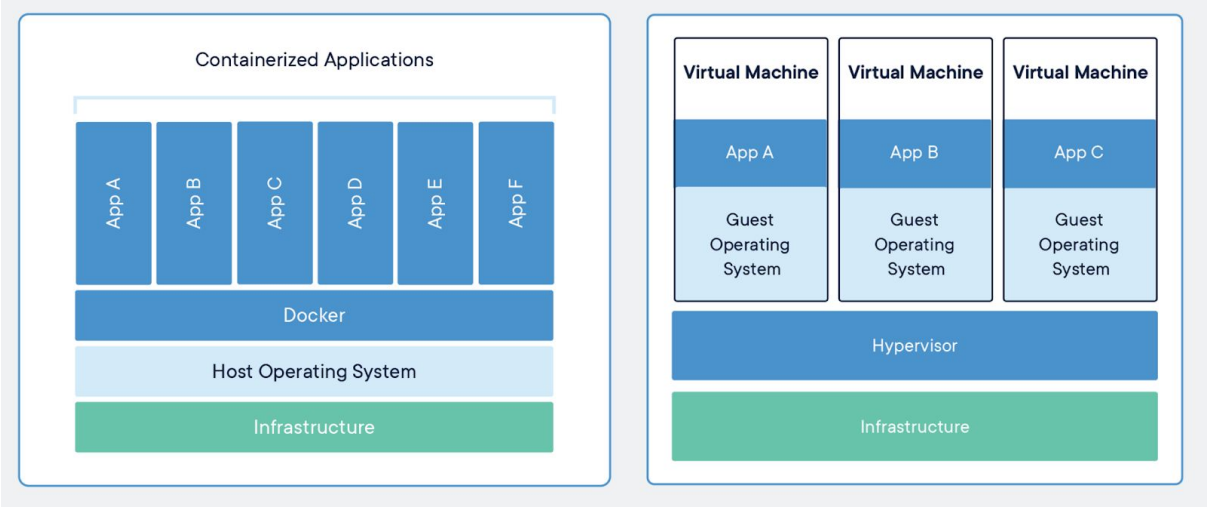

Figure 6.1 Docker Container VS Virtual Machines

Because the isolation provided by Docker Containers is sufficient and users might be sensitive to execution overhead. We decided to use Docker Containers as execution environments. Moreover, Pharos toolset is well-dockerized, which means managing tools using Docker is preferable for this project.