Bachelor Thesis

for the Bachelor of Engineering degree program in Mechatronics

Development of a Continuous Integration Framework for OptiSlang Workflows

In co-operation with

Robert Bosch GmbH

Author : Sathvick Bindinganavale Srinath
Matriculation Number : 4020025

Supervisor : Mr. André Haeitmann Dutra
1st Examiner : Prof. Dr.-Ing. Andreas Schiffler
2nd Examiner :

Submission Date : 06/10/2024

DECLARATION

I declare that the work in this dissertation, **"Development of a Continuous Integration Framework for OptiSlang Workflows"** was carried out in accordance with the regulations of the Technical University of Applied Sciences Würzburg-Schweinfurt. I declare that I have acknowledged all the sources I have used in writing this dissertation. I have clearly marked and acknowledged all direct quotations and all information obtained from other sources. I have not used any other sources or resources than those indicated. I have not submitted this dissertation to any other examination board. I am aware that any violation of these rules will result in the revocation of my degree. I am also aware that the dissertation may be checked for plagiarism using appropriate software.

_____ _____

Date Signature

Contents

List of Figures

Listings

Abbreviations

API ..... Application Programming Interface

CAE ..... Computer-Aided Engineering

CD ..... Continuous Deployment

CI ..... Continuous Integration

GUI ..... Graphical User Interface

JSON ..... JavaScript Object Notation

MOO ..... Multi Objective Optimization

PIDO ..... Process Integration and Design Optimization

SOO ..... Single Objective Optimization

YAML ..... YAML Ain't Markup Language

Abstract

This thesis explains us about the creation of a CI framework for opitSLang workflows.

Chapter 1

Introduction

1.1 Overview

This thesis explains about the creation and development of a framework for Optislang workflows.

1.2 Objective

The objective of this thesis are as follows:

- Development of a method to create standalone modules in Optislang, based in MATLAB and Python.(only code based, without using GUI).
- Implementation of modules and workflows without the help of GUI.
- Creation of a framework in Python to create and test modules and workflows in Optislang.
- Establishment of a strategy for automated integration testing in Github for modules based in Python and MATLAB.

1.3 Outline

- In Chapter 2, we are going to discuss about the MOO project, its role in the company and the modules created in Python and Optislang.
- In Chapter 3, we will discuss the motivation and the principles used in the development of the framework.
- In Chapter 4, we understand the creation of a framework to create standalone modules and workflows in Python.
- In Chapter 5, we will look into the automated integration testing in Github.
- In Chapter 6, we will discuss the results and evaluate the impact of the solution.
- Finally, Chapter 7 provides us the summary of the work, highlighting the achievements and feedback for development in potential areas.

Chapter 2

Multi Objective Optimization

2.1 Introduction

In today's increasingly complex world, decision-makers often face the challenge of optimizing several conflicting objectives simultaneously. Multi Objective Optimization (MOO) is an optimization that deals with such problems, where multiple objective functions are optimized simultaneously. To understand MOO better, let us consider an example.

Example: Let us consider an example of a car manufacturer. The car consists of many components like engine, body, wheels, etc which can be tweaked. In our case, the manufacturer wants to optimize the car for two objectives: lower manufacturing cost of the car and lower carbon emissions. With considering the input parameters and the objectives, we get many solutions as shown in Figure 2.1

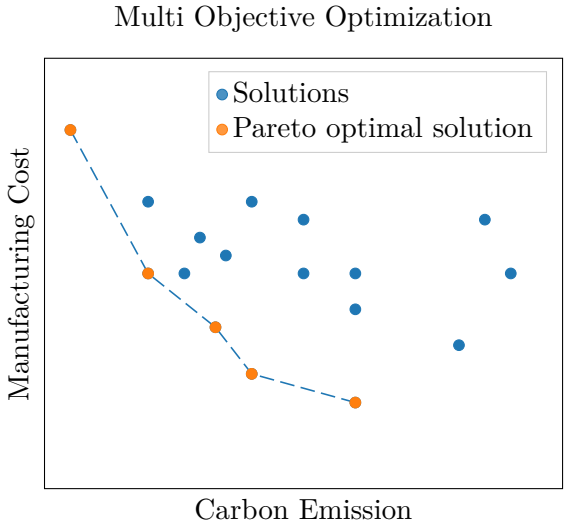

Figure 2.1: Example of MOO

In a MOO problem, there typically is no single best solution. Rather, the *goal* is to identify a set of solutions that are optimal in terms of all objectives. In Figure 2.1, the best solutions for the given objectives is indicated in orange known as pareto optimal solutions. A solution is said to be pareto optimal if no other solution can improve on any of the objectives without worsening at least one of the other objectives.

2.2 Difference between MOO and SOO

Optimization problems, whether single-objective or multi-objective, have the same goal: to find the best solution(s) to a given problem. However, the approach to solving these problems is different.

In Single Objective Optimization (SOO), the goal is to optimize a single objective function, which can either be maximized or minimized. The problem is simpler to define and solve because it involves only one objective. To calculate SOO, we can use methods like gradient descent, linear programming, etc.

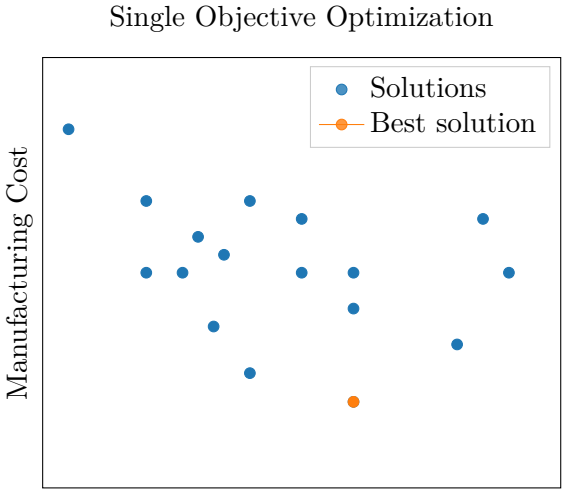

Figure 2.2: Example of SOO

In Figure 2.2, we have considered the same example given in section 2.1. But, here, we are considering only one objective, which is to minimize the manufacturing cost. The best solution is indicated in orange.

In MOO, the optimization involves two or more objective functions simultaneously. The problem is more complex because the objectives are often conflicting. Unlike SOO, where we have a single best solution, in MOO, we have pareto optimal solutions. To calculate MOO, we can use methods like pareto optimization, scalarization method, weighted sum method, ϵ-constraint method, etc.

While SOO focuses on finding the best solution according to a single criterion, MOO addresses the more complex task of balancing multiple, often conflicting objectives. The choice between SOO and MOO depends on the nature of the problem at hand and the goals of the decision-maker. Understanding the differences between these approaches is crucial for selecting the appropriate optimization technique and achieving the desired outcomes.

2.3 Optislang

To calculate MOO, we need a software platform that can handle the complexity of the problem. Ansys Optislang [1] is such a software platform, which is used for design exploration, CAE based sensitivity analysis and optimization in conjunction with any product development tool. It is a Process Integration and Design Optimization tool or in short, a PIDO tool. Process Integration refers to automate and orchestrate manual simulation processes and to realize complex workflows.

Design Optimization aims for better understanding of your design, optimizing the product, identify an improved design which has the desired qualities and resulting in a best design by reliability analysis and statistical analysis.

Optislang uses several solvers to look into aspects like mechanical, technical, mathematical and any other problems. This is easier in Optislang as it provides integration to create toolchains of many external programs like ANSYS, MATLAB, Excel, Python, CATIA and many more.

Our department utilizes Optislang for solving MOO problems, as it includes algorithms specifically designed for MOO.

2.4 Modules and Workflows

2.4.1 Modules

Modules are created by the system developers. Modules include a simulation model as a calculation with defined interfaces for coupling with other modules. These modules are either defined in MATLAB or Python. Each module is designed to tackle/improve a specific issue. To document and collaborate with other system developers, each module is versioned and stored in a specific manner in a repository in GitHub Enterprise.

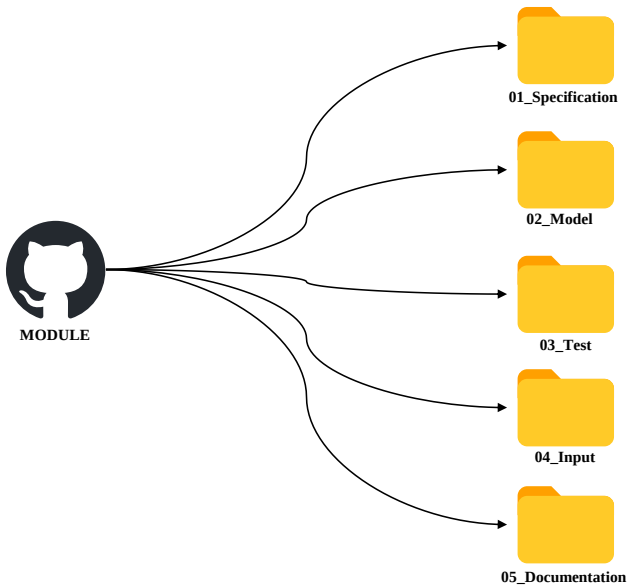

Figure 2.3: Example of a module's GitHub repository structure

Figure 2.3 shows us an example of how each module is maintained in our GitHub.

- `01_Specification` has all the requirements for the module to run.

- `02_Model` contains all parts to run the model. This can also be used as a playground for the development of a model.
- `03_Test` withholds all relevant documents regarding unit tests or integration test can be found here. This is one of the important part of the module.
- `04_Input` carries all the initial parameters or functions to be defined at the start of a module.
- Documentation explaining functioning and usage of the module can be found under `05_Documentation`.

2.4.2 Workflows

Our department develop automatized workflows for power electronics products considering functional loads to reliability indications. Workflows are a sequence of modules designed for a fast calculation performance characteristic like temperature or reliability indication. To develop a workflow, Optislang is used.

Every architectural workflow has a GitHub repository that is maintained in a similar way to how modules are being maintained.

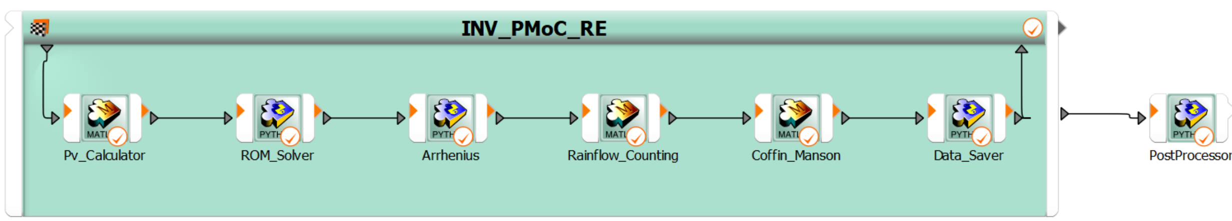

Figure 2.4: Example of a workflow in Optislang

2.5 Current Problem

lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc nec lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec auctor, nunc

Chapter 3

Practical Application of the Thesis

To overcome the problems discussed in section 2.5, this thesis proposes a solution to automate the process of testing standalone modules in Optislang. Since, the process is automated, the testing of modules needs to be done without the help of the GUI. To achieve this, a Python [2] framework is created to test the modules in an according manner. To use the framework in an automated manner, a CI pipeline is created using GitHub Actions. The pipeline is triggered whenever a new commit is pushed to the repository. The pipeline runs the tests on the modules in a virtual machine and checks if the results are as expected. If the tests fail, the pipeline notifies the developer about the failure. The developer can then look into the issue and resolve it.

To achieve this, the following sections explains the concepts and usage of DevOps, Continuous Integration and code quality in the context of this thesis.

3.1 DevOps

To create, test and maintain software applications, it is essential to have a well-defined process. Software development methodologies are models that help to define the process, roles and responsibilities in the software development lifecycle. Some of the earlier software methodologies like waterfall model, agile models, the deployment of software to production can only be done after the development and testing of the software is completed [6]. Therefore, to overcome this problem, DevOps was introduced, aiming to enhance productivity and efficiency in software development.

Devops is the combination of a set of practices, tools which helps to automate and integrate the processes between software and organizations. Here, developers and IT operations teams collaborate to build, test and release software, in order to deliver software faster and increase the quality of the software. This collaborative working is explained in the DevOps lifecycle shown in figure 3.1.

Figure 3.1: DevOps lifecycle

DevOps lifecycle is a continuous process, where the developers and operations team work together to deliver the software faster and with high quality. The left part of the loop represents the

development phase, where the developers are responsible for planning, coding, building and testing the software. The right part of the loop represents the operations phase, where the operations team is responsible for deploying, monitoring and maintaining the software. The

DevOps practices play a crucial role in the development of the automation process described in this thesis. By integrating Continuous Integration (CI) and Continuous Deployment (CD) pipelines, we ensure that the testing of modules is efficient and reliable. This helps us to improve productivity and reduce human error. According to [7], DevOps is not only helping to bridge the gap between development and operations teams, but also helping to improve the quality of the software. The DevOps approach allows for seamless collaboration between development and operations teams, ensuring that the testing framework and the modules it tests are consistently maintained and updated. This integration of DevOps practices not only enhances the quality of the software but also accelerates the development lifecycle, enabling faster delivery of new updates and features.

In summary, the application of DevOps in this thesis demonstrates how modern software engineering practices can be applied to automate and streamline the testing process, leading to more robust, efficient and reliable software solutions.

3.2 Continuous Integration

Continuous Integration (CI) is the practice of automating the integration of code changes from multiple contributors into a single software project. This practice allows to frequently merge code changes into a central repository where builds and tests are ran later on. A CI pipeline can be applied to automate and streamline the testing process, leading to more robust, efficient and reliable software solutions [3].

For the CI pipeline to work, usually the code changes are stored in a version control system which can be collaborated by multiple developers. Examples of such version control systems are Bitbucket, GitLab, GitHub. A crucial practice of CI is to commit the code changes to the repository frequently [5]. In this thesis, GitHub is used as the primary version control system. After the code changes are committed, the CI pipeline is triggered automatically.

A detailed implementation of the CI pipeline is discussed in Chapter **??**.

3.3 Code Quality

Code quality is a critical aspect of software development that directly impacts the maintainability, reliability, and performance of the software. High-quality code is easier to understand, test, and modify, which is essential for the long-term success of any software project [4]. In the context of this thesis, ensuring code quality is particularly important for several reasons:

- **Maintainability**:
  High-quality code is well-structured and well-documented, making it easier for developers to understand and maintain. This is crucial for the continuous integration and continuous deployment (CI/CD) pipelines, as it ensures that the codebase remains manageable and scalable over time.

- **Reliability**:
  Code quality directly affects the reliability of the software. Well-written code is less prone to bugs and errors, which reduces the likelihood of failures during the testing and deployment phases. This is particularly important in a DevOps environment, where the goal is to deliver reliable software quickly and efficiently.

- **Efficiency**:
  High-quality code is optimized for performance, which can lead to faster execution times and more efficient use of resources. This is important for the automation processes described in this thesis, as it ensures that the testing and deployment pipelines run smoothly and efficiently.

- **Collaboration**:
  In a collaborative environment, such as GitHub, high-quality code is essential for effective teamwork. Clear, well-documented code allows multiple contributors to work on the same project without confusion or conflicts, which is a key aspect of successful CI practices.

In summary, maintaining high code quality is essential for the success of the automation processes and the overall framework described in this thesis. It ensures that the software is maintainable, reliable, efficient, and conducive to collaboration, all of which are critical for achieving the goals of Continuous Integration and Continuous Deployment.

Chapter 4

Creation of Framework

4.1 Introduction

4.1.1 What is a Framework?

A framework is a pre-built structure that provides a foundation for developing applications. It includes libraries, tools, and best practices that accelerate the development process. Frameworks serve as templates that can be customized to meet project requirements. Framework allow developers to work on the core, i.e, the application of logic, rather than worrying about the underlying structure. Many frameworks are open-source and are easily available. Developers can also contribute to the framework by adding new features or fixing bugs.

It is crucial to first understand your project requirements and determine which programming language and corresponding framework best suit those needs. Each framework is designed for a specific purpose and offers unique features. Having a fundamental understanding of the chosen programming language is essential for effectively working with the framework. Popular frameworks used today include Django, Flask, Angular, React, PyTorch, TensorFlow, and more. These frameworks empower developers to create robust and feature-rich applications.

4.1.2 Why is a Framework used?

Developing code from scratch can be a tedious and error-prone task. Clean, well-tested, and bug-free code is essential, but achieving this can be challenging. Additionally, developers must adhere to coding standards and best practices to ensure code quality. Therefore, using frameworks that meet your requirements is a better choice. Frameworks simplify the development process, reduce errors and provide a general template that can be customized as needed. They also make it easier for others to understand your code, as they are likely familiar with the frameworks used. Frameworks offer several advantages, including:

- Simplified testing and debugging of code.
- Clean and understandable code.
- Reduced code redundancy within the project.
- Decreased project time and cost.
- Modifiable and extendable features and functionalities provided by the framework.

4.1.3 Libraries vs Frameworks

It is a common misconception that libraries and frameworks are the same. But they serve different purposes and have distinct characteristics.

Libraries:

- A library is a collection of pre-written code that developers can use to optimize tasks.
- It provides specific functionality that can be called upon when needed.
- Developers have control over the flow of the application and decide when to use the library.
- Some of the popular libraries include NumPy, Pandas, Matplotlib, and more.

Frameworks:

- A framework is a pre-built structure that provides a foundation for developing applications.
- It dictates the architecture and flow of the application.
- Developers must adhere to the structure and guidelines set by the framework.
- Examples of popular frameworks include Django, Flask, Angular, React, and more.

4.2 Overview of the Framework

Before, building a framework, it is essential to understand the requirements and objectives of the project. The main objective of this framework is to automate the process of running parametric simulations in Optislang, standardize and to verify the output files generated. The framework should be user-friendly, easy to use, and provide detailed error logs in case of any issues.

After understanding the requirements, the next step is to design the framework. The framework should be designed in such a way that it is scalable, modular, and easy to maintain. It should also be flexible enough to accommodate future changes and updates.

This framework is built by with the help of classes, functions and libraries like `Pandas`,`NumPy` and other built in modules in Python. Figure 4.1 shows a brief overview and working of the framework.

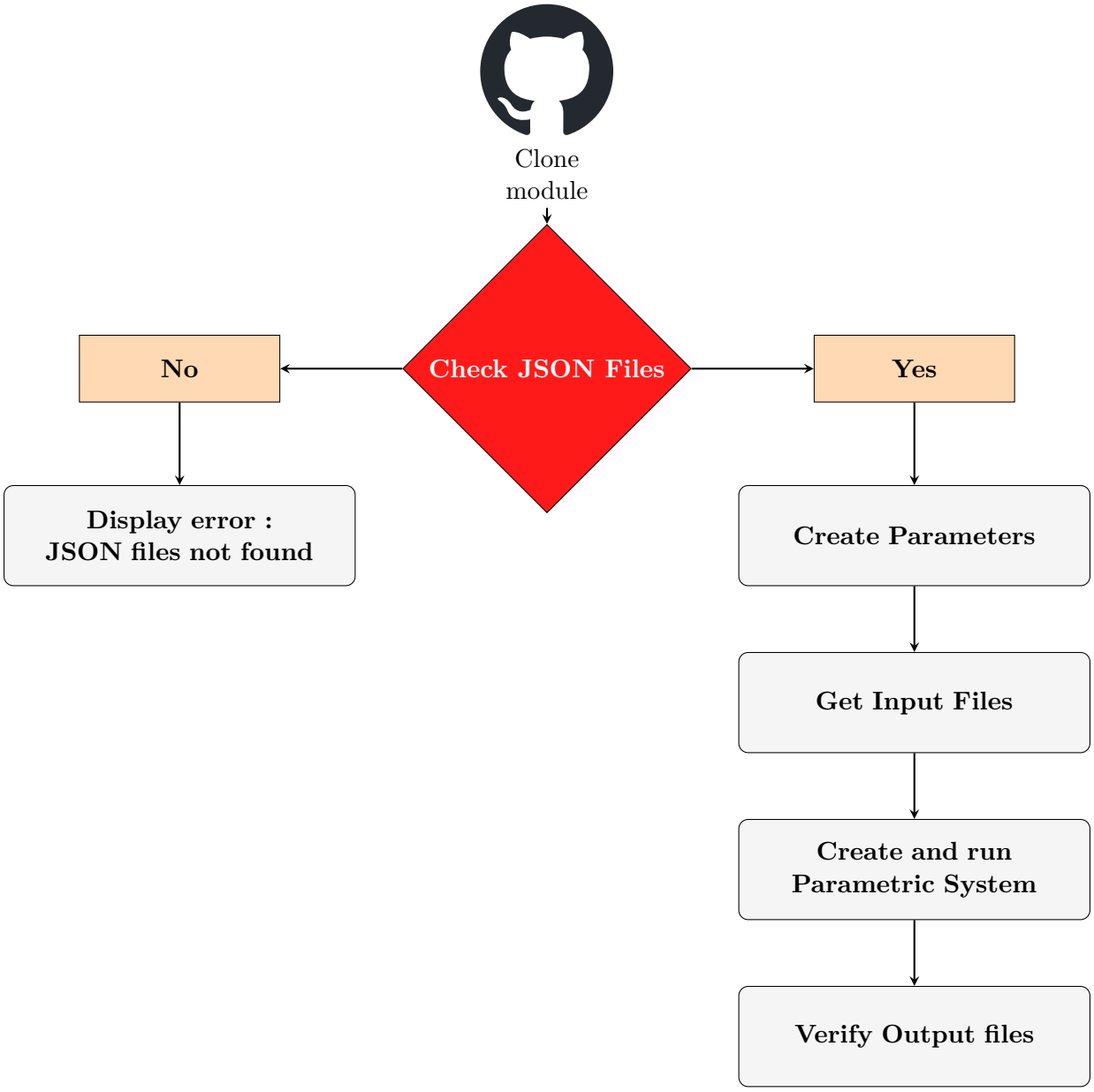

Figure 4.1: Flowchart of the framework

Let us understand the working of the framework in detail. The framework is mainly built using Python and uses Optislang's Python API to create and run the parametric simulations. The primary requirement for the framework to run is to have module present. Therefore, the first step is to clone the required module from the specific repository and branch from GitHub. These serve us as the main arguments needed to run the framework. After cloning, the framework checks for `module_config.json` and `parameters.json` files. These JSON files are crucial to be present in the module as they contain the information required to run the parametric simulations automatically. The `module_config.json` file contains information of the module like description of the module,

name of the main script containing the algorithm, type of framework, input and output files and their properties.These data are important to create, run and verify the parametric system generated. The `parameters.json` file contains information about the parameters required to be set as input in the parametric system.

At this stage, a decision is implemented. If the JSON files are present, the framework proceeds to further steps. If the JSON files are not found, the framework comes to a halt and displays an error message being shown to the user. Figure 4.1 shows the implemetation of the error message. This function is present inside the class `ParametricSystem`.

```
def json_files_log(self):
    try:
        if not self.get_module_config_path().exists():
            raise FileNotFoundError(
                f"{self.get_module_config_path()} does not exist"
            )
    except FileNotFoundError as e:
        print(
            f"{e} \nPlease ensure {self.get_module_config_path()} exists and re-run"
        )
    except Exception as e:
        print(e)
    try:
        if not self.get_parameters_path().exists():
            raise FileNotFoundError(f"{self.get_parameters_path()} does not exist")
    except FileNotFoundError as e:
        print(f"{e} \nPlease ensure {self.get_parameters_path} exists and re-run")
    except Exception as e:
        print(e)
```

Code 4.1: Function to verify existance of JSON files

If the framework identifies the JSON files, it process to the next step to create parameters which are required to run the parametric system. These parameters are created based on the information present in the `parameters.json` file. These parameters are then fed as input the parametric system during runtime.

The next step is to provide input files which are required by the parametric system in order to execute the simulations. These input files are being stored in a pod in OpenShift. To retrieve these input files from OpenShift, an API is setup using FastAPI. A detailed explanation of retrieving input files from OpenShift is provided in Section **??**.

After the input files are retrieved, the framework then needs to create the parametric system. This should be achieved without the user's intervention, i.e, automatically. At this stage, we will be using the Python interpreter provided by Optislang as it includes the necessary libraries and functions to create and run the parametric system. Another terminal pops up which displays the progress of the execution of the system.

Since the whole process in the framework is automated, we need to ensure that the files generated by the parametric system are correct and are produced as expected. This is done by the function `verify_output_files` present in the class `ParametricSystem`.

4.3 Directory Structure

Before discussing the implementation of the framework, let us first understand how the files are structured within the framework. The directory structure is depicted in Figure 4.2. The framework consists of the following files and directories:

- **`moo_framework_workflow.yaml`**:
  This file contains the workflow for running the framework automatically. It is written in YAML and is later used inside GitHub Actions.

- **`src`**:
  This directory contains the source code of the framework. It consists of files which are used to create and run the parametric system in Optislang. It also consists of some helper functions which are used to build complex functions and classes for the creation of framework. We will discuss more about these files in section .

- **`tests`**:
  This directory contains test cases for the framework. Section explains the test cases in detail.

- **`main.py`**:
  This python file calls the files which are responsible for the framework creation from `src` directory . It is the main file for running the framework.

- **`requirements.txt`**:
  This file contains the list of libraries which helps in running the framework. It is important to install these libraries before running the framework.

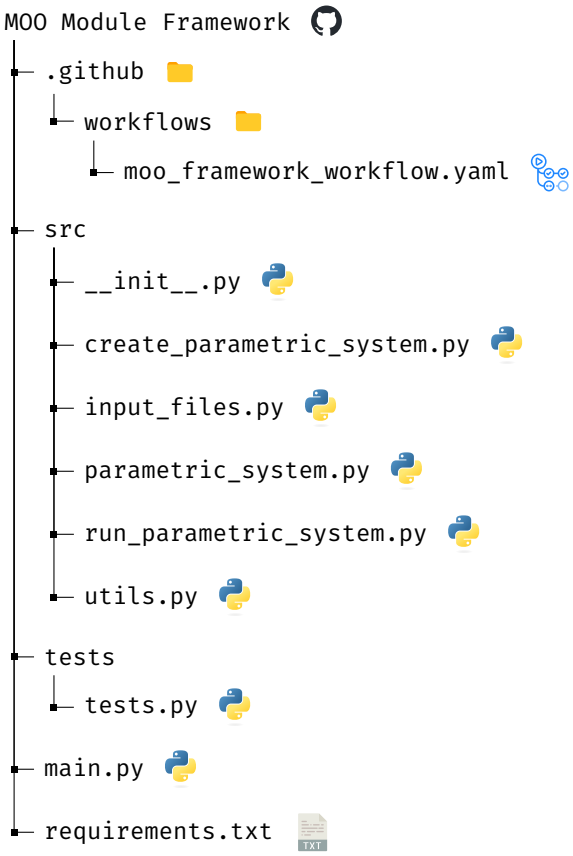

Figure 4.2: Directory structure of the framework

4.4 Implementation of the Framework

In this section, we will discuss the working and functioning of each of the files present in the framework.

4.4.1 Source Code Overview

The source code of the framework is present in the `src` directory. Figure 4.2 shows the files present in the `src` directory. Let us understand what each file contributes to the proper functioning of the framework.

`__init__.py` :

This is a special file in Python that is executed when a package is imported. It is used to define packages and can be used to initialize their namespaces. Python would not recognize the directories as packages without this file.

Here, we have used this file to import the class `ParametricSystem` from the file `src/parametric_system.py` which is later being called inside `main.py`.

```
1 from .parametric_system import ParametricSystem
```

Code 4.2: `__init__.py`

utils.py :

This file is crucial as it serves as a backbone of the framework. This file contains functions that are used to build complex functions and classes which are responsible for the proper implementation of the framework.

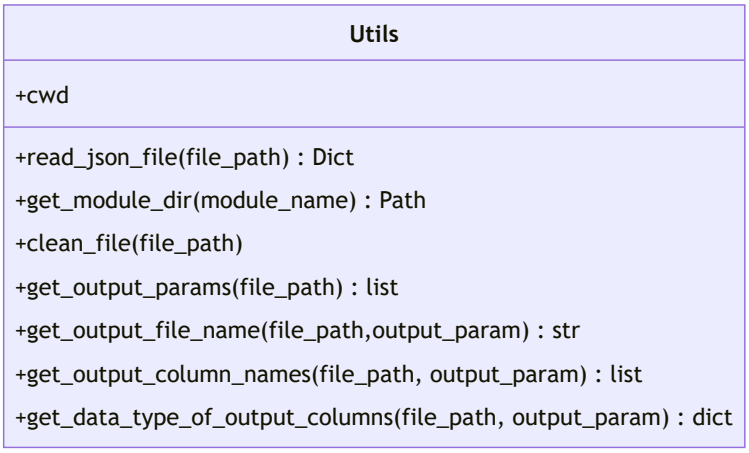

Figure 4.3: Overview of functions in `utils.py`

- The variable `cwd` stores the path of the current working directory by using the `os` module.
- The function `read_json_file` reads the JSON file and returns the data in the form of a dictionary. The function requires the name of the JSON file as an argument to get the data.
- `get_module_dir` function takes in the name of the module as an argument and returns the absolute path of the module directory.
- `clean_file` function takes in the file path as an argument. This function is responsible for cleaning the file by removing the file.
- The function `get_output_params` takes in the path of `module_config.json` file as an input and returns the list of output parameters defined in the JSON file.
- `get_output_file_name` function returns the name of the output file based on the output parameter and the file path of `module_config.json` provided in the argument.
- `get_output_column_names` function takes in `module_config.json`'s file path and the output parameter as input and returns a list of column names present in the output file.
- `get_data_type_of_output_columns` function is used to get the data type of the columns present in the output file. This function takes in the file path of `module_config.json` and the output parameter as input and returns a dictionary containing the data type of columns.

The functions `get_output_params`, `get_output_file_name`, `get_output_column_names`, and `get_data_type_of_output_columns` are mainly used in the last part of the framework where we verify the output files generated by the parametric simulations. Whereas, the functions `read_json_file`, `get_module_dir`, and `clean_file` are used in the core process of building the framework.

create_parametric_system.py :

This file is used to create and build a parametric system without the use of Optislang's GUI. This file is being ran in Optislang's Python interpreter which has libraries and functions to create the system. Therefore, the file `create_parametric_system.py` and `run_parametric_system.py` are being ran outside the class `ParametricSystem`.

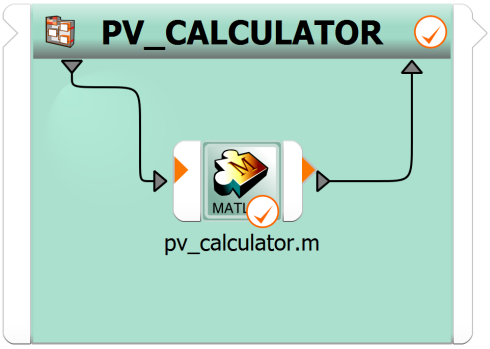

Figure 4.4: Example of a parametric system in Optislang

Since, the whole creation process needs to be automated, the file `module_config.json` helps us in creating the parametric system without the user's intervention. The JSON file contains key pair values which contain information like the name of the script containing the algorithm, type of framework which helps in creating the system. For example, in Figure 4.4, the system is getting the algorithm from the script `pv_calculator.m`. Type of framework refers to the type of module, Python or MATLAB.

```
def create_parametric_actor(actor_name:str = get_module_name()):

    copy_file = actors.PythonActor('copy_input_files')
    parametric_system.add_actor(copy_file)

    if get_framework_type() == 'python':
        actor = actors.PythonActor(actor_name)
        parametric_system.add_actor(actor)
    elif get_framework_type() == 'matlab':
        actor = actors.MatlabActor(actor_name)
        parametric_system.add_actor(actor)

    #Connecting the python actor
    connect(parametric_system, 'IODesign', copy_file, 'IDesign')
    connect(copy_file, 'ODesign', actor, 'IDesign')
    connect(actor, 'ODesign', parametric_system, 'IIDesign')

    #Defining the path of the algorithm
    copy_file.path = str(Path(cwd,'src\\input_files.py'))
    if get_framework_type() == 'python':
        actor.path = str(path_to_algorithm())
    elif get_framework_type() == 'matlab':
        actor.file_path = str(path_to_algorithm())

    #Importing the parameters from the csv file
    params = parametric_system.parameter_manager
```

```
    params.import_from_csv(f'{cwd}\\optislang_actor_parameters.csv',',')
    parametric_system.parameter_manager = params

    #Reads the file 'optislang_parameters.py' which contains the necessary parameters for the actor to run
    exec(open(Path(cwd,'optislang_parameters.py')).read())
```

Code 4.3: Function to create a parametric system

The function `create_parametric_actor()` takes in the argument actor name, which is the name of the actor to be displayed in the parametric system. For the automation to function, we need to input the files required for the system to run automatically. The actor `copy_file` is used to copy the input files from OpenShift using Fast API. A detailed implementation of this is provided in section Based on the type of module present, the actor `copy_file` connects to either Python or MATLAB actor. This logic of connecting the actors depending on the module type is created as Optislang's Python API defines different functions to implement to create different actors. After connecting the actors, the path for the actors is defined. The parameters are being created and fed to the system using the functions `generate_optislang_parameters_from_json()` and `generate_optislang_parameters_to_csv()` which are present inside the class`ParametricSystem`.

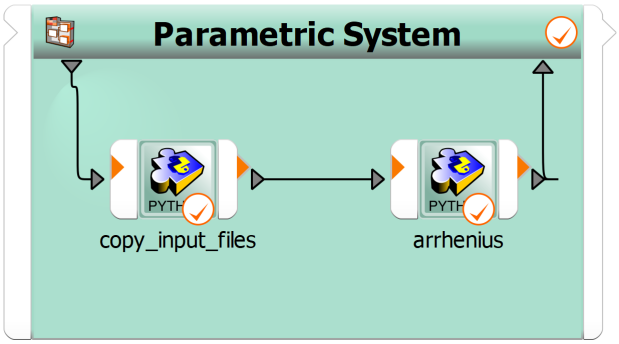

Figure 4.5: Example of a parametric system with copy actor

`run_parametric_system.py` :

This file mainly calls the parametric system created in `create_parametric_system.py` and runs the system in Optislang's instance. This file is later being called in the class `ParametricSystem` to run the parametric system.

```
#Creating a parametric system
obj = actors.ParametricSystemActor("Parametric System")
obj.auto_save_mode = AS_ACTOR_FINISHED
add_actor(obj)

#Reads the file 'optislang_parameters.py' which contains the design connections for the module
parametric_system = obj
cwd_ = os.getcwd()
cwd = Path(cwd_).parents[1]
exec(open(Path(cwd,'src','create_parametric_system.py')).read())
obj = parametric_system
```

Code 4.4: Overivew of `run_parametric_system.py`

parametric_system.py :

This is the most important script in the framework. This file contains the class `ParametricSystem` which is mainly responsible for the creation of the framework. Due to the complexity of the framework, we use `utils.py` to build the functions and classes.

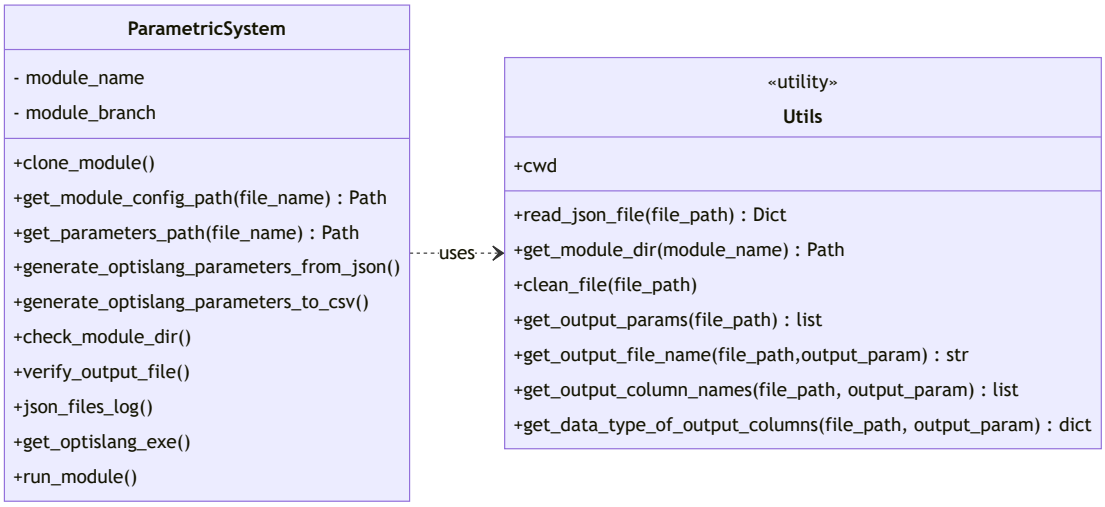

Figure 4.6: Overview of the class `ParametricSystem`

Figure 4.6 describes the methods used in the class `ParametricSystem`. The user needs to provide the name of the module and the name of the branch from where the module needs to be cloned. The class `ParametricSystem` contains the following methods:

- **`clone_module()`**:
  This method is used to clone the module from the GitHub repository by using the arguments provided by the user. The function `clone_module()` tries to delete if an existing module is present and clones the module from the repository.

- **`get_module_config_path()`**:
  This function returns the path of `module_config.json` file. This functions takes in name of the file as an argument and the returns the absolute path of the file.

- **`get_parameters_path()`**:
  This functions acts similar to `get_module_config_path()` but returns the path of `parameters.json` file. This function take sin the name of the file as an argument and returns the absolute path of the file.

- **`generate_optislang_parameters_to_csv()`**:
  To input the parameters inside the parametric system, the parameters are being fed in a `csv` format. To achieve this, the function `generate_ optislang_parameters_to_csv()` is used. This function takes in the path of `parameters.json`, reads the data in it and converts it into a `csv` format. Later, this `csv` is saved in the current working directory as `optislang_actor_parameters.csv`.

- **`generate_optislang_parameters_from_json()`**:
  Optislang requires the user to provide the parameters not only in `csv` format, but

also while creating the system. To overcome this, this function creates the parameters from the data provided in `parameters.json`. Later, the data is being saved as `optislang_parameters.py` containing the parameters required. This is later being used inside `create_parametric_system.py` as shown in figure 4.3.

- **`check_module_dir()`**:
  While creating the modules, the system developers have hardcoded the path of the module in many of their functions. To work around this, this function mocks the module directories by creating a new directory called **`Module`** containing the child directories, `02_Model` and `04_Input` respectively. `02_Model` contains the Optislang simulation file and its corresponding input and output files. Modules like `ARRHENIUS` and `COFFIN MANSON` needs a specific input file to be saved in `04_Input`. If there is a previously existing folder named `Module`, it ensures to remove the contents of it and create a new one.

- **`verify_output_files()`**:
  This function is primarily used to verify the existence and correctness of the output files generated by the parametric system. This function is used after the parametric system is created. Figure 4.7 shows a brief overview of the function.

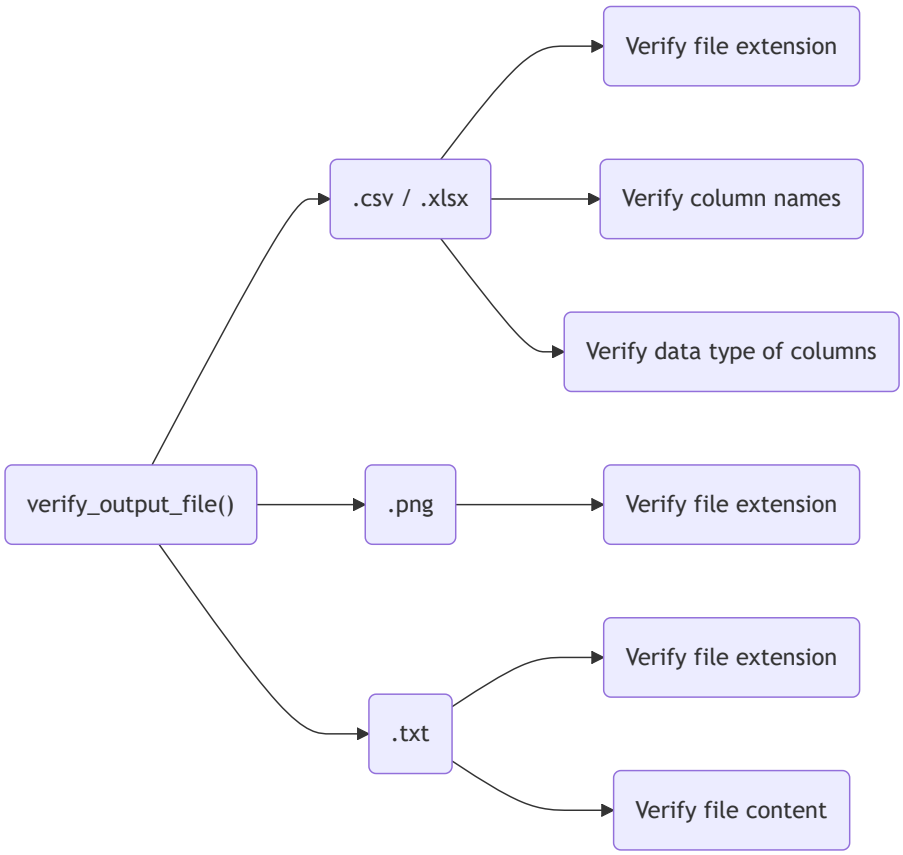

Figure 4.7: Overview of `verify_output_files()` function

To verify the output, we first get the data to verify from the `module_config.json`. This data

contains the properties of output files like column names, name of the output file, it's format and data type of the columns. The functions gets the folder containing the output files and iterates it through one by one to verify if all the output files are present. After verifying the existence of the output files, the algorithm checks if the columns present in the output files match the columns present in the `module_config.json`.Here, it uses the library `Pandas` to read the `csv` files. After verifying the columns, the algorithm continues to check for the data type of the columns. If all the checks are passed, it shows a success message stating that the output files are correct. If any of the checks fail, it shows that the output files have an issue displaying the specific error message.

Since all the outputs are not in a `csv` file, the function also needs to check for output files of type `.txt` and `.png`. To check for `.png`, we only just verify if the file is present and the file name is correct. For `.txt` files, we do the same by verifying the existence of the file and the file name. In addition to that, it also opens and verifies if the generated `.txt` file is empty or not.

- **`json_files_log()`**:
  This function provides us a detailed error message if any one of the `JSON` files are not found. A detailed explanation of this function is already explained in Figure 4.1.

- **`get_optislang_exe()`**:
  Since the framework is designed to run automatically, it is essential not to hardcode the path any of the files. This function's goal is to provide the location of Optislang's executable file. This is achieved by using `os.getenv()` function from the built-in module `os`, which returns the value of the environment variable containing the path of the Optislang executable file. If the path is not found, the function raises an error message.

- **`run_module()`**:
  This function calls the other functions present in an orchestrated manner as shown in figure 4.1 to run the module. In this function, initially, it clears all the existing files and folders cerated during the previous run to avoid any conflicts. Later, it clones the module, checks for the JSON files, creates the parameters, retrieves the input files, creates the parametric system and finally runs it.

4.5 Testing of Framework

While building the framework, it is also essential to test the framework to ensure that it is working as expected. One way to do is to include breakpoints, add print statements, and debug the code. However, this method is not efficient when the codebase is huge. Therefore, another way is to write unit cases for the framework.

Unit testing is a software testing method that involves testing a small unit of code, typically a function or method. They are crucial part of the development process as they help in identifying bugs and errors early in the development cycle. Python has two frameworks for unit testing, `unittest` and `pytest`. I have implemented `unittest` for testing since it is part of the Python's standard library. Here, the unit tests can be found in the `tests` directory. The file `tests.py` contains the test cases for the framework. Unit tests generally should cover the following aspects:

- **Unit tests**:
  This is used to test the functionalities of each individual methods and functions.
- **Integration tests**:
  These tests are implemented to verify if the integration to the files are working as expected.
- **Boundary tests**:
  This ensures to check the edge cases of the functions. For example, if the provided input is an empty string, the function should return an error message.
- **Negative tests**:
  To check if the function handles incorrect input properly, negative tests are implemented. An example for this would be to handle if the argument is of a different data type than expected.

Figure 4.5 shows an example of a unit test implemented in Python to test the existence of JSON files.

```python
class TestParametricSystem(unittest.TestCase):
    def setUp(self) -> None:
        self.parametric_system = ParametricSystem('MOO_M_ARRHENIUS','MOO-1355_py_framework_poc')
        self.cwd = os.getcwd()

    def get_module_name(self):
        for file_name in os.listdir(self.cwd):
            if file_name.startswith('MOO'):
                return Path(self.cwd,file_name)
        return None

    def test_get_module_config_path(self):
        self.assertIsNotNone(self.get_module_name(), 'Module folder not found.')
        expected_path = (Path(self.get_module_name(),'01_Specification', 'module_config.json'))
        actual_path = self.parametric_system.get_module_config_path()
        self.assertEqual(expected_path, actual_path)
```

Code 4.5: Example of a unit test

Firstly, we create a class `TestParametricSystem` which inherits from `unittest.TestCase`. To avoid initialization of the same variables in each test case, we take advantage of the **`setUp()`** method. Here, we create an object an initialize the arguments for the class `ParametricSystem`. The functions in the unit tests needs to start with the prefix `test`. This convention is used to identify the function which the test cases. For example, in Figure 4.5,the function `test_get_module_config_path` recognizes that it is a test case whereas the function `get_module_name` is a helper function and not a test case. In this example the function `test_get_module_config_path` is responsible to check if the path of the folder containing the JSON file is correct. To ensure this, we use the function `assertEqual` which checks if the expected path is equal to the actual path. If the paths are equal, the test case passes, else it fails. `unittest` provides several other functions to test the code.

4.6 Execution of Framework

To execute the framework, the user needs to run the file `main.py`. This file is the main file to run the framework. This script calls the functions from the class `ParametricSystem` and runs the module.

Figure 4.6 shows the execution of the framework.

```python
from src import ParametricSystem

module_name = "MOO_M_ARRHENIUS"
module_branch_name = 'MOO-1355_py_framework_poc'

def main():
    system = ParametricSystem(module_name, module_branch_name)
    system.clone_module()
    if (
        system.get_module_config_path().exists() & system.get_parameters_path().exists()
    ) == True:
        system.run_module()
        system.verify_output_file()

    else:
        system.json_files_log()

if __name__ == "__main__":
    main()
```

Code 4.6: Execution of framework using `main.py`

In code snippet 4.6, an example of cloning the module `MOO_M_ARRHENIUS` is shown. An instance of the class `ParametricSystem` is created with the arguments `module_name` and `module_branch_name`. Then, the check for the `JSON` files is done. The verifying of the files is done outside the function `run_module()` as we do need to verify the existence of the files before running the module. Once the existence of the files are verified, the module is being run and after the successful execution in Optislang, it displays the status of the output files.

The working of `input_files.py` will be explained in section wherein a detailed explanation of retrieving input files from OpenShift using Fast API is provided.

4.7 Retrieve input files for Framework

4.7.1 Introduction

4.7.2 Docker

4.7.3 OpenShift

4.7.4 Fast API

4.7.5 Implementation

Bibliography

[1] Ansys optislang. Online; Accessed 10/05/2024.

[2] Python. Online; Accessed 07/08/2024.

[3] S.A.I.B.S. Arachchi and Indika Perera. Continuous integration and continuous delivery pipeline automation for agile software project management. In *2018 Moratuwa Engineering Research Conference (MERCon)*, pages 156–161, May 2018.

[4] Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser, and Andy Zaidman. Test code quality and its relation to issue handling performance. *IEEE Transactions on Software Engineering*, 40(11):1100–1125, Nov 2014.

[5] Mathias Meyer. Continuous integration and its tools. *IEEE Software*, 31(3):14–16, May 2014.

[6] E Naresh, S V N Murthy, N. Sreenivasa, Seshaiah Merikapudi, and C R Rakhi Krishna. Continuous integration, testing deployment and delivery in devops. In *2024 International Conference on Knowledge Engineering and Communication Systems (ICKECS)*, volume 1, pages 1–4, April 2024.

[7] Pulasthi Perera, Roshali Silva, and Indika Perera. Improve software quality through practicing devops. In *2017 Seventeenth International Conference on Advances in ICT for Emerging Regions (ICTer)*, pages 1–6, Sep. 2017.