GSoC 2024 Proposal for

Automatic reproducibility of COMPSs experiments through the integration of RO-Crate in Chameleon

Project Size: Large (350 hours)

Mentor: Raül Sirvent

Archit Dabral

INDEX:

	PAGE
Cover	1
Index	2
About me	3
1 Project Details	
1.1 About the project	3
1.2 Background	3-6
1.3 Effects of this Project	7-8
2 Proposal	
2.1 Proposal abstract	
2.2 RS(ReproducibilityService Program)	
2.2.1 ReproducibilityService Class	11-13
2.2.2 Parse metadata function	14
2.2.3 Generate automate.sh	15
2.2.4 Generate and build Docker Image	16
2.3 Comments on Trovi Integration	17
3 Project Timeline	
3.1 Deliverables and Scheduling	19
3.2 Detailed Project Scheduling	19-20
4 Personal Details	
4.1 More about me and Why me?	21
4.2 My Projects	22
4.3 Commitments	23
4.4 My Availability	23
5 Post GSoC	23
6 Some Local Tested Results	24-25
7 References	26

Archit Dabral

Github:	Minimega12121
Linkedin:	Archit Dabral
Email Address:	architdabral1234567890@gmail.com (personal)
	archit.dabral.mat22@itbhu.ac.in (university)
Phone number:	(+91)9997690204
Timezone:	Indian Standard Time (UTC +5:30)
Resume/CV:	Link

About me:

Hi! I am Archit Dabral, a Mathematics and Computing Engineering sophomore from IIT BHU, India. I am a tech enthusiast and love to keep exploring new fields related to computers. Most of my projects are based on backend, operating systems, and some on machine learning. This is my first time participating in GSoC, and I am really excited to contribute to open-source.

1.1 About the project:

The project aims to develop a service that facilitates the automated replication of COMPSs experiments within the Chameleon infrastructure. This service will have the capability to take a COMPSs crate (an artifact adhering to the RO-Crate specification) and, through analysis of the provided metadata, construct a Chameleon-compatible image for replicating the experiment on the testbed. Minor adjustments to the COMPSs RO-Crate are anticipated, such as incorporating third-party software necessary for the application.

1.2 Background:

The three technologies COMPSs, RO-Crate and Chameleon exist stand-alone but there is no technology currently that integrates the three.

So why are these technologies chosen?

(1) Programming is done with the help of COMPs:

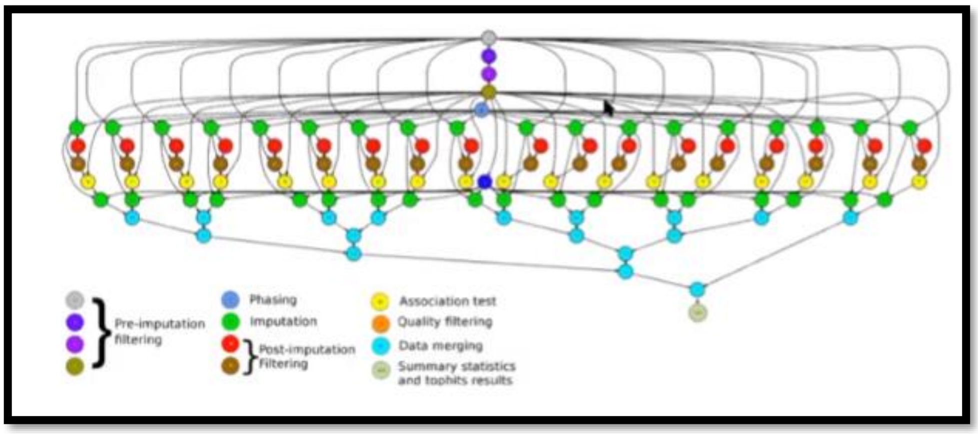
COMPs provide the following advantages:

- **Sequential programming**: COMPSs programmers can develop their applications following the sequential programming paradigm. In this sense, the programmer does not need to take care of the parallelization and distribution aspects, such as thread creation and synchronization, data distribution, messaging or fault tolerance.
- **Infrastructure agnosticism**: COMPSs abstracts applications from the underlying distributed infrastructure. COMPSs programs do not include any

detail that could tie them to a particular platform, like deployment or resource management. This makes applications portable between infrastructures with diverse characteristics.

- **Standard programming languages:** COMPSs applications can be developed in Java, Python and C/C++. The use of a general-purpose programming language facilitates adoption, since these languages are between the more common and popular between programmers.
  - **Task based:** task is the unit of work
  - **Task Dependency Analysis:** tasks are the basis for the parallelism in COMPSs. The runtime automatically finds the data dependencies between tasks based on the direction of their parameters. With this information, it dynamically builds a task dependency graph.
  - **Data synchronization:** data accesses from the main program of the application are automatically synchronized by the runtime, when necessary.

Example of a task dependency graph build :



(2) The format of metadata generated in COMPSs experiment follows RO-Crate specification

More about RO-Crate and how it used:

- **What is RO-Crate?**

  RO-Crate (Research Object Crate) is a method for aggregating and describing research data along with associated metadata in a structured manner.

- **Format Type of RO-Crate:**

  The core of RO-Crate is a JSON-LD (JavaScript Object Notation for Linked Data) file, named ro-crate-metadata.json, which contains structured metadata about the dataset and its files.

- **Why JSON-LD is Better Than JSON?**

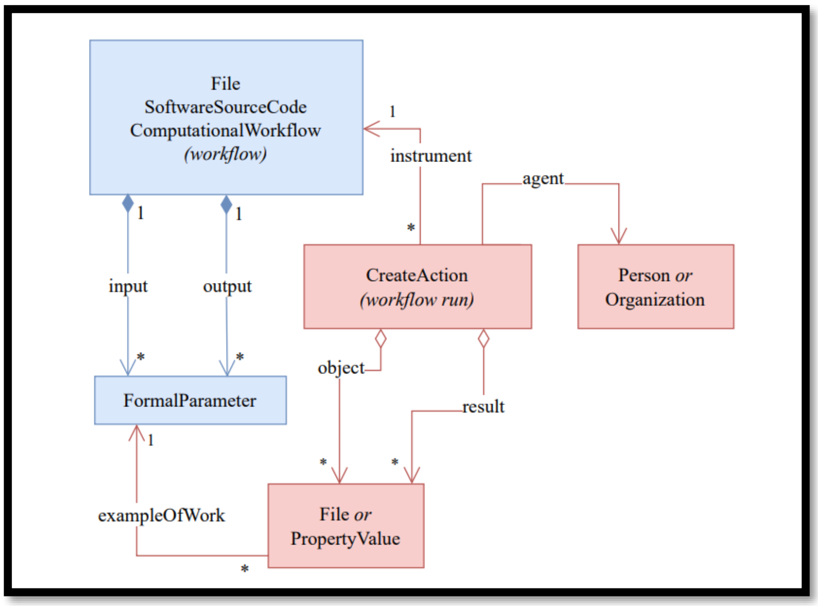
  JSON-LD is preferred over JSON due to its ability to express metadata using linked data, allowing for easier integration and interoperability between different systems and vocabularies. JSON-LD also provides a standardized way to represent relationships between entities, making it more suitable for describing complex data structures such as research provenance.

- **How RO-Crate Makes Provenance of Research Easier:**

  RO-Crate enhances research provenance by providing a standardized format to capture metadata about who created the data, what equipment and software were used, under which license the data can be reused, where it was collected, and other relevant contextual information. This structured approach makes it easier to track and understand the lineage and origins of research data, which is crucial for ensuring reproducibility and trustworthiness in scientific research.

NOTE: The format of the metadata generated in COMPSs experiments follows the RO-Crate specification, and, more specifically, two profiles: the Workflow and Workflow Run Crate profiles:

Understanding Workflow and Workflow Run Crate Profiles in COMPSs Experiments



Workflow Crate Profile in COMPSs:

In COMPSs experiments, the Workflow Crate profile captures the static description of the workflow. This includes metadata about the workflow structure, tasks, dependencies, parameters, inputs, and outputs. It provides a detailed blueprint of how the workflow is designed and organized.

Workflow Run Crate Profile in COMPSs:

On the other hand, the Workflow Run Crate profile in COMPSs deals with dynamic information during the execution of a specific workflow instance. It includes runtime data, such as execution context, resource allocation, task scheduling, intermediate results, and logs generated during execution.

Benefits for Provenance and Reproducibility in COMPSs Experiments:

- **Provenance Tracking in COMPSs:** Using Workflow and Workflow Run Crate profiles, researchers can track the provenance of data and processes within COMPSs experiments. They can trace back the lineage of data, identify processing steps, and understand how results were derived, enhancing transparency and accountability.
- **Reproducibility Support in COMPSs:** These profiles facilitate experiment reproducibility by documenting the workflow design, task dependencies, input data, execution parameters, and runtime configurations. Researchers can precisely replicate the experiment setup and execution environment, ensuring reproducibility across different platforms or by different users.
- **Comprehensive Documentation in COMPSs:** By structuring workflow descriptions and execution details, the profiles provide comprehensive documentation for COMPSs experiments. This documentation aids in understanding, sharing, and collaborating on workflows, contributing to better research practices and knowledge dissemination.

In conclusion, the Workflow and Workflow Run Crate profiles are instrumental in enhancing provenance tracking and reproducibility in COMPSs experiments. They provide detailed descriptions of workflow design, execution context, and runtime data, promoting transparency, trustworthiness, and effective collaboration in scientific research using COMPSs.

---

(3) Chameleon Cloud provides a cloud infrastructure that is useful for replicating hardware to reproduce the experiment in the testbed.

What is a testbed?

A testbed is a platform for conducting rigorous, transparent, and replicable testing of scientific theories, computing tools, and new technologies.

What is Chameleon?

Chameleon is an NSF-funded testbed system for Computer Science experimentation. It is designed to be deeply reconfigurable, with a wide variety of capabilities for researching systems, networking, distributed and cluster computing and security. Chameleon's features include:

- Bare metal access to hardware, via cloud
- A wide variety of hardware types (eg: Infiniband, NVMe, GPUs/FPGAa, Low-power Xeon and ARM processors)
- A powerful set of networking capabilities,including: Isolated layer-2 networks , SDN/OpenFlow, and Dedicated WAN layer-2 links

1.3 Effects of this Project

This is where the project comes in and proposes to create a service that incorporates the three which would have the following effects:

Enhanced Traceability and Provenance Recording:

Problem: The complexity of supercomputers and distributed systems poses challenges in tracking executed tasks and their outcomes.

Solution: The project integrates COMPSs with RO-Crate's Workflow and Workflow Run Crate profiles to record comprehensive provenance data. This enhances traceability, reproducibility, and quality assessment of scientific data products.

Outcome: Researchers can trace back data lineage, understand processing steps, and reproduce experiments accurately, promoting transparency and accountability in scientific research.

Improved Reproducibility and Collaboration:

Problem: Ensuring experiment reproducibility across different platforms or by different users is challenging due to varied execution environments.

Solution: The service facilitates experiment reproducibility by documenting workflow design, task dependencies, input data, and runtime configurations using the Workflow and Workflow Run Crate profiles.

Outcome: Researchers can precisely replicate experiment setups and execution environments, enabling effective collaboration and knowledge dissemination among scientific communities.

Comprehensive Documentation and Knowledge Sharing:

Problem: Lack of structured documentation hinders understanding, sharing, and collaboration on workflows.

Solution: By structuring workflow descriptions and execution details, the profiles provide comprehensive documentation for COMPSs experiments.

Outcome: Researchers have access to detailed documentation, aiding in understanding, sharing, and collaborating on workflows. This contributes to better research practices and facilitates knowledge dissemination in scientific communities.

Efficient Experiment Replication on Chameleon Infrastructure:

Problem: Replicating COMPSs experiments on different infrastructures for testing purposes requires manual adjustments and compatibility checks.

Solution: The service automates the replication of COMPSs experiments within the Chameleon infrastructure by analysing metadata and constructing Chameleon-compatible images.

Outcome: Researchers can efficiently replicate experiments on the Chameleon testbed, enabling faster testing, validation, and experimentation of scientific applications.

Promoting Transparent and Trustworthy Research Practices:

Problem: Lack of transparency and trustworthiness in research practices can lead to challenges in verifying and validating scientific findings.

Solution: The integration of COMPSs with RO-Crate and Chameleon infrastructure promotes transparent and trustworthy research practices by enhancing provenance tracking, reproducibility, and collaboration.

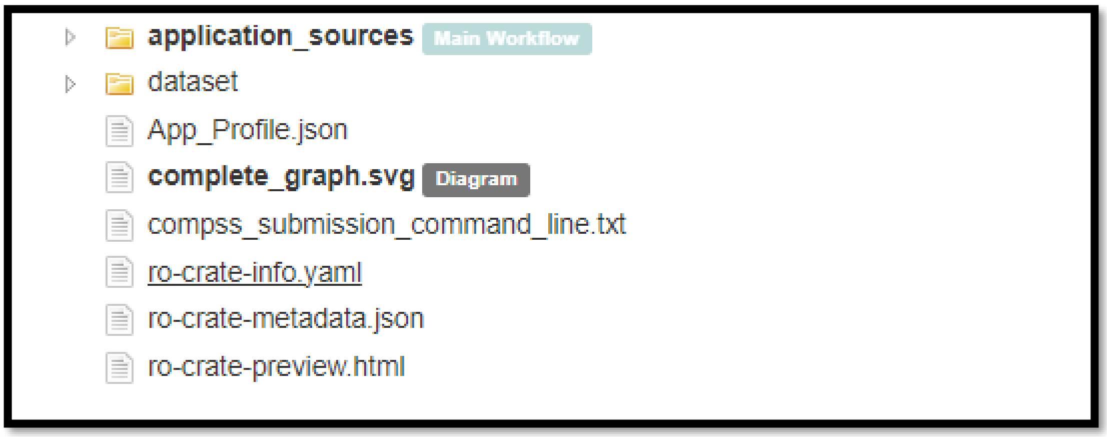
Outcome: Researchers can ensure the reliability and credibility of their research outcomes, fostering trust and credibility within the scientific community.

Proposal:

2.1 Proposal Abstract:

Let's say we want to reproduce an Experiment X via a Program RS (*Reproducibility Service*), then the process would be divided into the following steps:

Pre-requisites: (1) We already have the Workflow folder of the experiment to be reproduced, which would look like this:



```
application_sources  Main Workflow
dataset
App_Profile.json
complete_graph.svg  Diagram
compss_submission_command_line.txt
ro-crate-info.yaml
ro-crate-metadata.json
ro-crate-preview.html
```

(2) The program RS is inside a folder where it stores all the information while running this is termed as the ROOT. Here it contains ROOT/Workflow where store the workflow files as shown above.

Step1: RS parses through the ro-crate-metadata.json file of the Experiment X and gets information about:

(1) Dependencies and Configurations needed (stores in a folder named ROOT/APP-REQ/app-req.txt)

Note: The integration of third-party dependencies into the experiment will be addressed in the project's second phase due to its complexity. Details regarding this phase are discussed later in the '**Timeline**' section.

(2) Data required as input (stores in folder named ROOT/APP-REQ/dataset/)

Basically, the above step just separated the information regarding dependencies needed to be installed and the data needed.

Step2: RS checks the APP-REQ/app-req.txt to extract the dependencies and configuration and stores the requires cli commands needed in an automate.sh file in the root directory.

Step3: Now RS makes a dockerfile to make the image of the experiment and during the making of image include the automate.sh file as a RUN command.

Step4: The program RS then finally runs the docker file to make the Chameleon compatible image (satisfies all the pre-requisite of chameleon compatible image) that can reproduce the experiment in the testbed.

Additional:

Logging and Reporting:

- Implement logging mechanisms in RS using Python's logging module to track the execution process, errors, and status updates.
- Generate reports or logs summarizing the reproducibility process, including any issues or discrepancies encountered.

Terminologies:

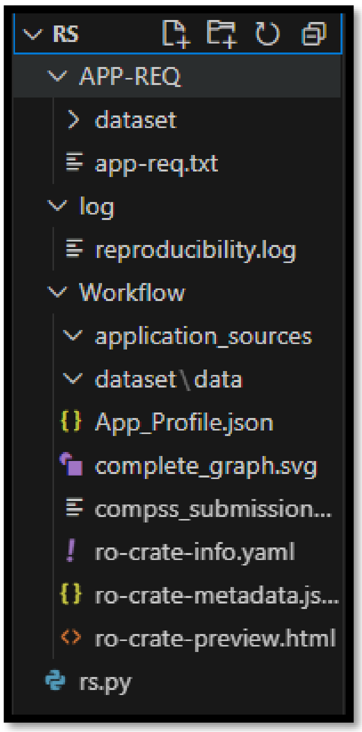
- **What is meant by Dependencies and Configuration?**

Dependencies refer to both external (third-party software) and internal (e.g., libraries related to pyCOMPSs) requirements that must be installed to ensure the proper reproducibility of the experiment. It is essential to capture these dependencies to replicate the experiment accurately.

- **What is a Dockerfile ?**

A Dockerfile is a script-like text file used to define the steps and configurations required to build a Docker image. It includes instructions such as defining a base image, installing dependencies, setting environment variables, and running commands to configure the container environment.

The Final Directory would be like



```
RS
  APP-REQ
    dataset
    app-req.txt
  log
    reproducibility.log
  Workflow
    application_sources
    dataset\data
    App_Profile.json
    complete_graph.svg
    compss_submission...
    ro-crate-info.yaml
    ro-crate-metadata.js...
    ro-crate-preview.html
  rs.py
```

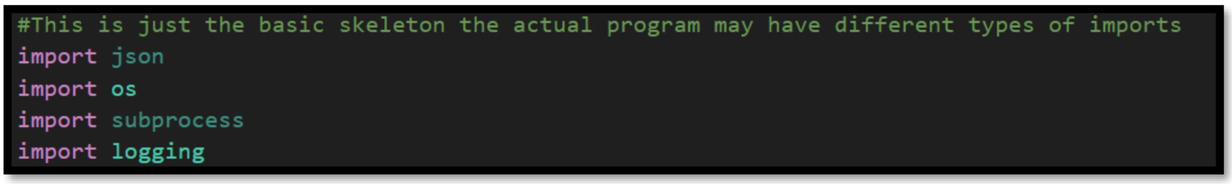
This is just the basic summary of the proposal ,below is each step in detail:

2.2 RS (Reproducibility Service) Program:

This is the actual program that automates the whole process of reproducing the experiment.

The RS program is designed to facilitate the reproducibility of experiments by automating the setup process within a Docker container. It parses experiment metadata, extracts dependencies and configurations, creates a Docker image, and runs the Docker container to reproduce the experiment in a testbed environment.

Here is the basic code snippet:



```python
#This is just the basic skeleton the actual program may have different types of imports
import json
import os
import subprocess
import logging
```

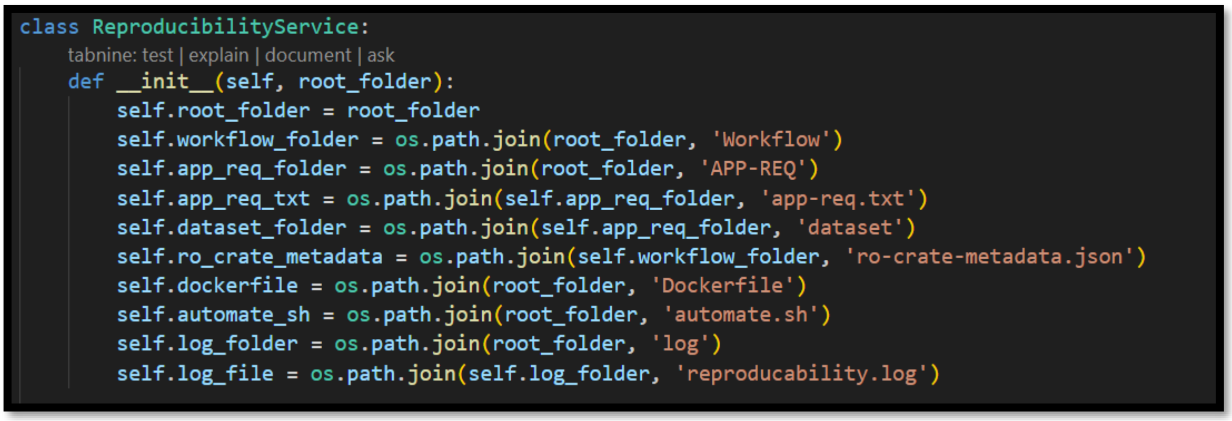
2.2.1 ReproducibilityService Class:

The ReproducibilityService class includes different methods which are essential for the running of RS program and divides the work for better understanding of the program.

It includes:

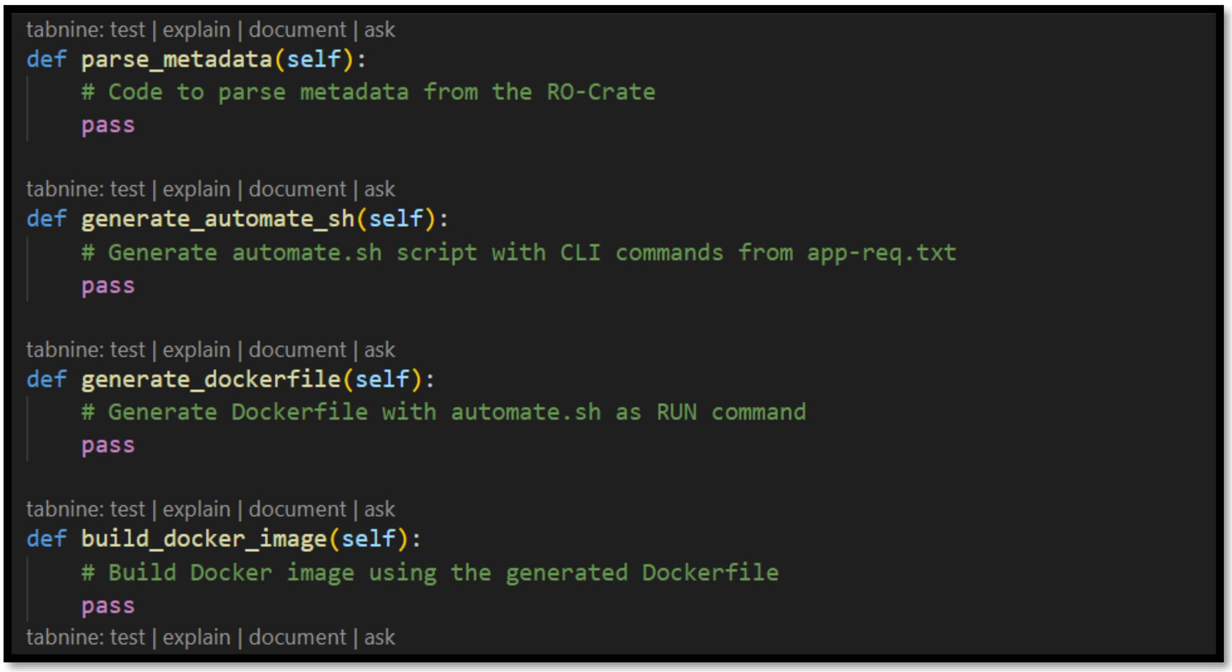
(1) def _init_(self, root_folder) :

The init method sets the value for the required attributes. The code creates necessary folders (if they don't already exist) and sets paths to relevant files and folders. Here root_folder is string that contains the path to root directory.



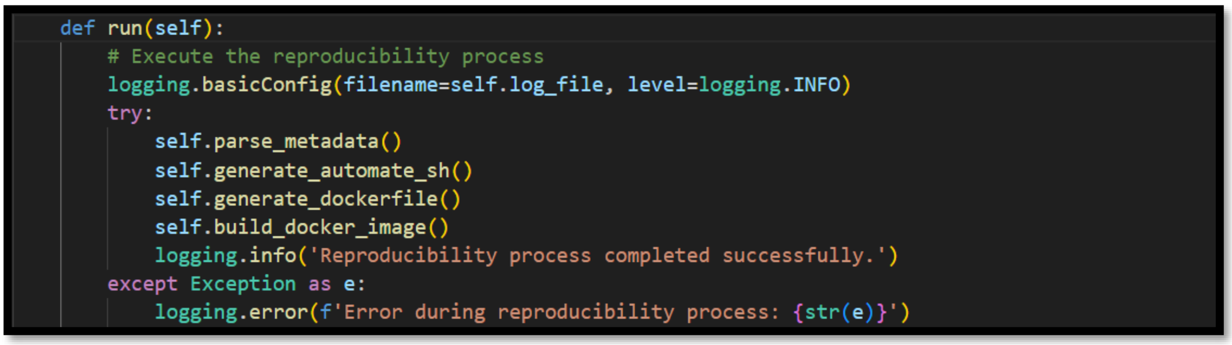
```python
class ReproducibilityService:
    tabnine: test | explain | document | ask
    def __init__(self, root_folder):
        self.root_folder = root_folder
        self.workflow_folder = os.path.join(root_folder, 'Workflow')
        self.app_req_folder = os.path.join(root_folder, 'APP-REQ')
        self.app_req_txt = os.path.join(self.app_req_folder, 'app-req.txt')
        self.dataset_folder = os.path.join(self.app_req_folder, 'dataset')
        self.ro_crate_metadata = os.path.join(self.workflow_folder, 'ro-crate-metadata.json')
        self.dockerfile = os.path.join(root_folder, 'Dockerfile')
        self.automate_sh = os.path.join(root_folder, 'automate.sh')
        self.log_folder = os.path.join(root_folder, 'log')
        self.log_file = os.path.join(self.log_folder, 'reproducability.log')
```

(2) The main functionality of the RS program is divided in the form of different methods within ReproducibilityService Class. They will be discussed in details individually ahead.



```
tabnine: test | explain | document | ask
def parse_metadata(self):
    # Code to parse metadata from the RO-Crate
    pass

tabnine: test | explain | document | ask
def generate_automate_sh(self):
    # Generate automate.sh script with CLI commands from app-req.txt
    pass

tabnine: test | explain | document | ask
def generate_dockerfile(self):
    # Generate Dockerfile with automate.sh as RUN command
    pass

tabnine: test | explain | document | ask
def build_docker_image(self):
    # Build Docker image using the generated Dockerfile
    pass
tabnine: test | explain | document | ask
```

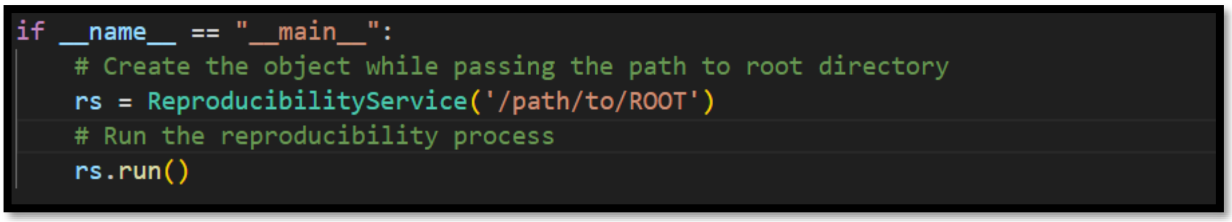
(3) def run(self) : This function acts as a link between the various methods involved by calling them and logging if the Reproducibility was success or not.



```
def run(self):
    # Execute the reproducibility process
    logging.basicConfig(filename=self.log_file, level=logging.INFO)
    try:
        self.parse_metadata()
        self.generate_automate_sh()
        self.generate_dockerfile()
        self.build_docker_image()
        logging.info('Reproducibility process completed successfully.')
    except Exception as e:
        logging.error(f'Error during reproducibility process: {str(e)}')
```

Running of RS program:

In the main function first the ReproducabilityService object is created which calls the __init__ function , then the ' rs.run() ', command is used for executing the reproducibility process.



```
if __name__ == "__main__":
    # Create the object while passing the path to root directory
    rs = ReproducibilityService('/path/to/ROOT')
    # Run the reproducibility process
    rs.run()
```

Overall, the run method orchestrates the execution of the experiment and the generation of reports/logs, encapsulating the core functionality of the RS program in managing reproducibility tasks from start to finish.

WHY Object-Oriented structure ?

Using object-oriented programming (OOP) for execution in the ReproducibilityService (RS) program offers several advantages. One of these advantages is encapsulation, which allows hiding the internal details of how methods and data are implemented. This encapsulation reduces complexity and makes the code more manageable and easier to understand for other developers. Additionally, encapsulation promotes better organization and modularization of code, making it easier to maintain and update in the future.

2.2.2 Parse_metadata function:

So, one of the most important part of the project to parse through the ro-crate-metadata.json for finding dataset used as well as to find the dependencies and configurations required.

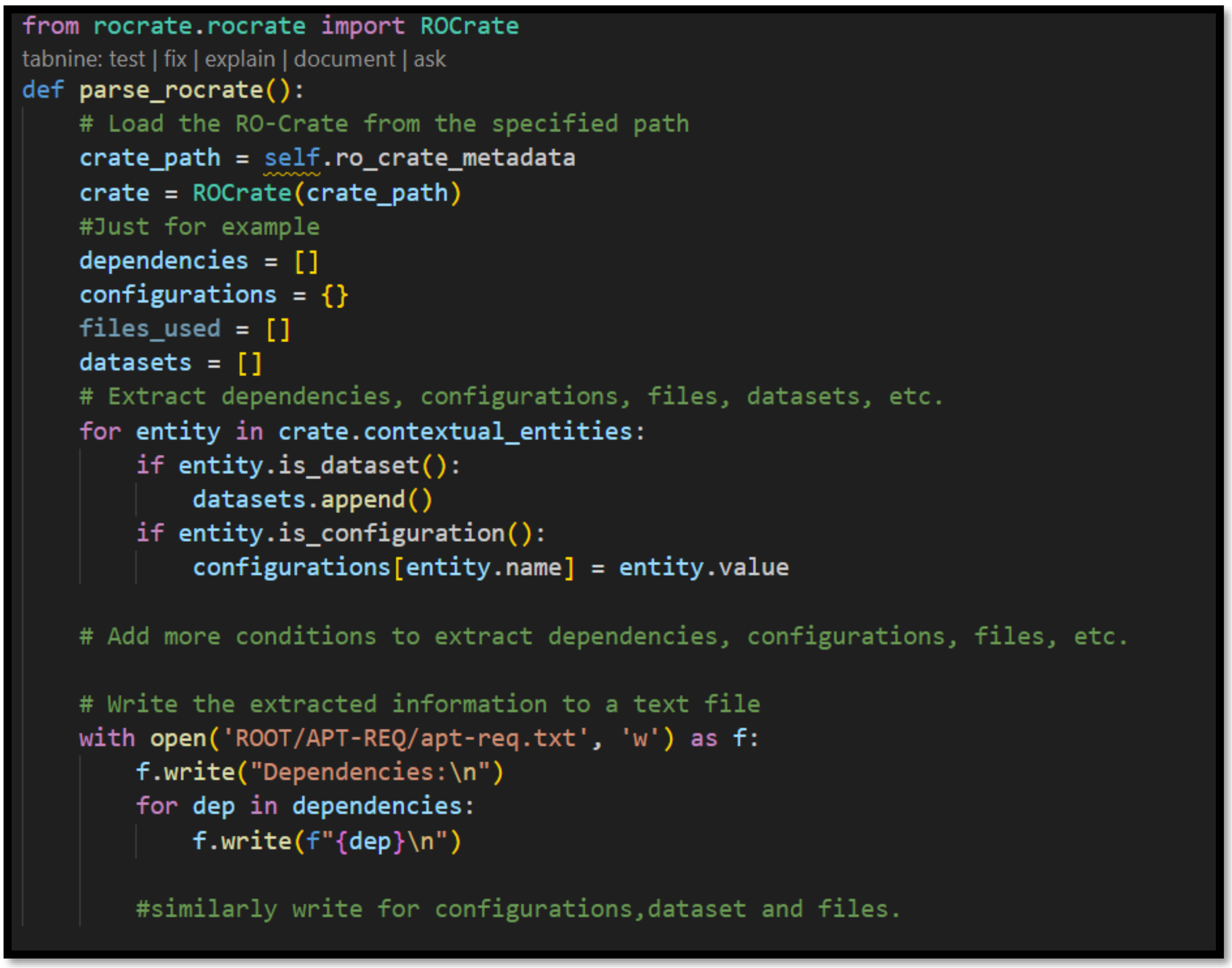
So, for proper functioning of the **function we would need to study the RO-crate structure in detail** like:

- How different ‘@type’ have different usecase senerio and how they vary from one another.
- What ‘@type’ refers to which type of data and how it get used as the input in the experiment.

Some examples of ‘@type’ includes:

- **Core Metadata For Data enitites:** ‘File’, ‘Dataset’
- **Web-based Data Entities**

The parsing function skeleton would look like



```
from rocrate.rocrate import ROCrate
tabnine: test | fix | explain | document | ask
def parse_rocrate():
    # Load the RO-Crate from the specified path
    crate_path = self.ro_crate_metadata
    crate = ROCrate(crate_path)
    #Just for example
    dependencies = []
    configurations = {}
    files_used = []
    datasets = []
    # Extract dependencies, configurations, files, datasets, etc.
    for entity in crate.contextual_entities:
        if entity.is_dataset():
            datasets.append()
        if entity.is_configuration():
            configurations[entity.name] = entity.value

    # Add more conditions to extract dependencies, configurations, files, etc.

    # Write the extracted information to a text file
    with open('ROOT/APT-REQ/apt-req.txt', 'w') as f:
        f.write("Dependencies:\n")
        for dep in dependencies:
            f.write(f"{dep}\n")

        #similarly write for configurations,dataset and files.
```

Logic behind ‘is_dataset()’ , ‘is_file()’ , ‘is_creativework()’ etc:

For this **in depth research needs to be done on how RO-crate** and how experiment reproducibility are co-related to one another. Proper algorithm would be needed to be developed and this would be the **main soul of the RS implementation.**

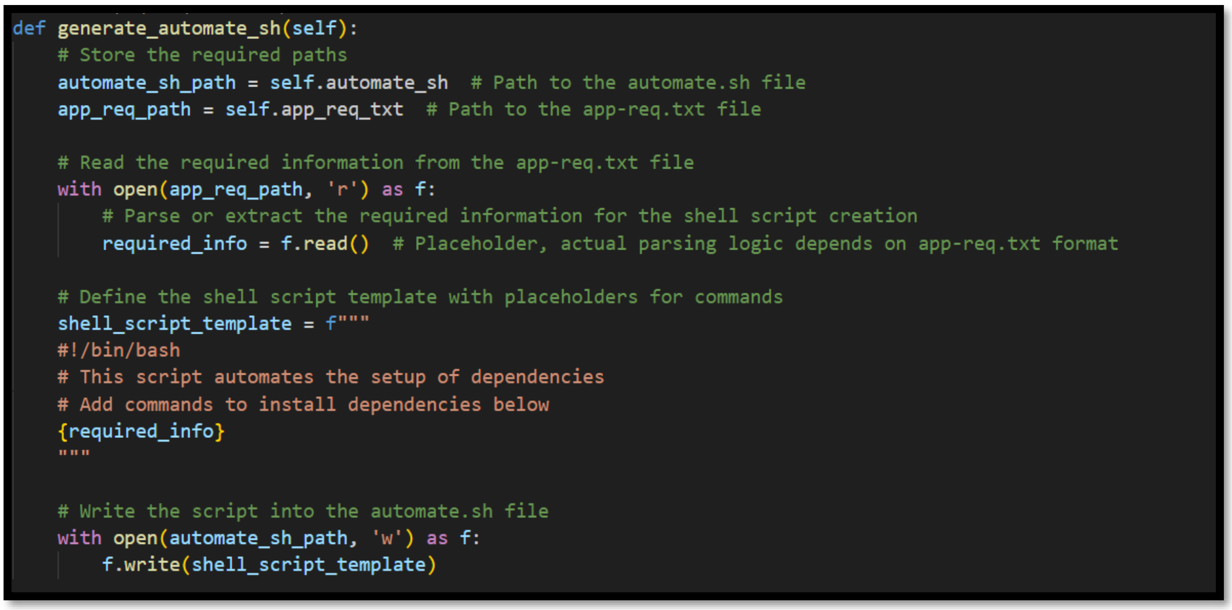
2.2.3 Generate Automate.sh:

The primary objective of the RS program is to streamline the reproducibility procedure. To achieve this, the central concept revolves around creating an automate.sh file, which serves as a shell script. This shell script is incorporated into the Dockerfile creation process to ensure that the container image formation renders the environment compatible for reproducing the experiment.

Technologies required for this would be: Shell Scripting Language such as Bash (Bourne Again Shell), sh (Bourne Shell), csh (C Shell), ksh (Korn Shell), and more.

Therefore, **a key objective is to explore the most suitable language and approach for the RS program's purpose.**

The basic structure of generate_automate_sh program would look as follows:



```python
def generate_automate_sh(self):
    # Store the required paths
    automate_sh_path = self.automate_sh  # Path to the automate.sh file
    app_req_path = self.app_req_txt  # Path to the app-req.txt file

    # Read the required information from the app-req.txt file
    with open(app_req_path, 'r') as f:
        # Parse or extract the required information for the shell script creation
        required_info = f.read()  # Placeholder, actual parsing logic depends on app-req.txt format

    # Define the shell script template with placeholders for commands
    shell_script_template = f"""
    #!/bin/bash
    # This script automates the setup of dependencies
    # Add commands to install dependencies below
    {required_info}
    """

    # Write the script into the automate.sh file
    with open(automate_sh_path, 'w') as f:
        f.write(shell_script_template)
```

Description:

- The generate_automate_sh() method is used to generate the automate.sh shell script file based on the information extracted from the app-req.txt file.
- It first stores the paths to the automate.sh file (automate_sh_path) and the app-req.txt file (app_req_path).
- It then reads the content of the app-req.txt file to extract the required information for setting up dependencies (represented by required_info in the script).
- The shell script template is defined with a placeholder {required_info} where the extracted commands or information will be inserted.
- Finally, the script template is written into the automate.sh file

Note that the automate.sh script will also incorporate a command to copy the ROOT/Workflow/dataset file into the image container. This action ensures that the necessary inputs are transferred inside the environment. The specific command for achieving this is:

RUN cp -r /path/to/ROOT/Workflow/dataset /path/inside/container

2.2.4 Generate and Build Docker Image:

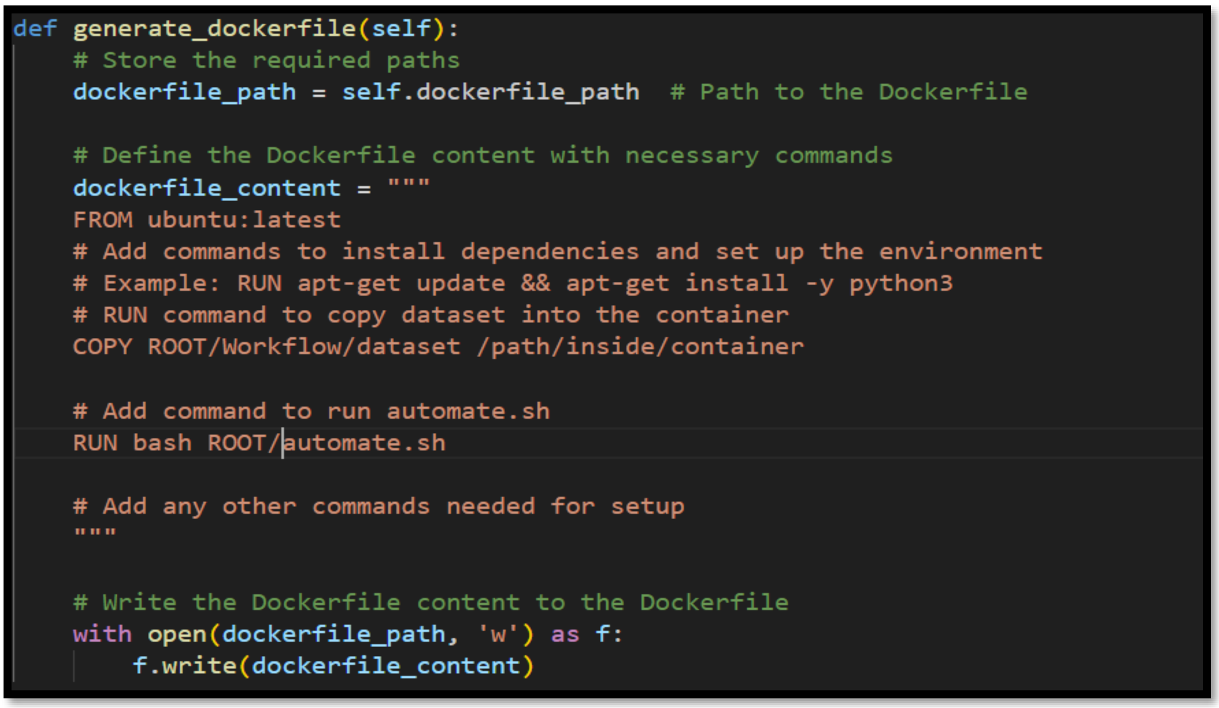
So far we have parsed through the RO-crate , extracted the required information needed to reproduce the experiment and made the automate.sh file needed for creation of the Chameleon compatible docker image.

Now to generate and build the docker image **we have to research the following**:

1. What type of image does Chameleon infrastructure allow?
2. Which type of image is best needed for reproducibility of pyCOMPS experiments?
3. And what optimisation needs to be done for proper functioning in the cloud infrastructure

Generate_docker image method():

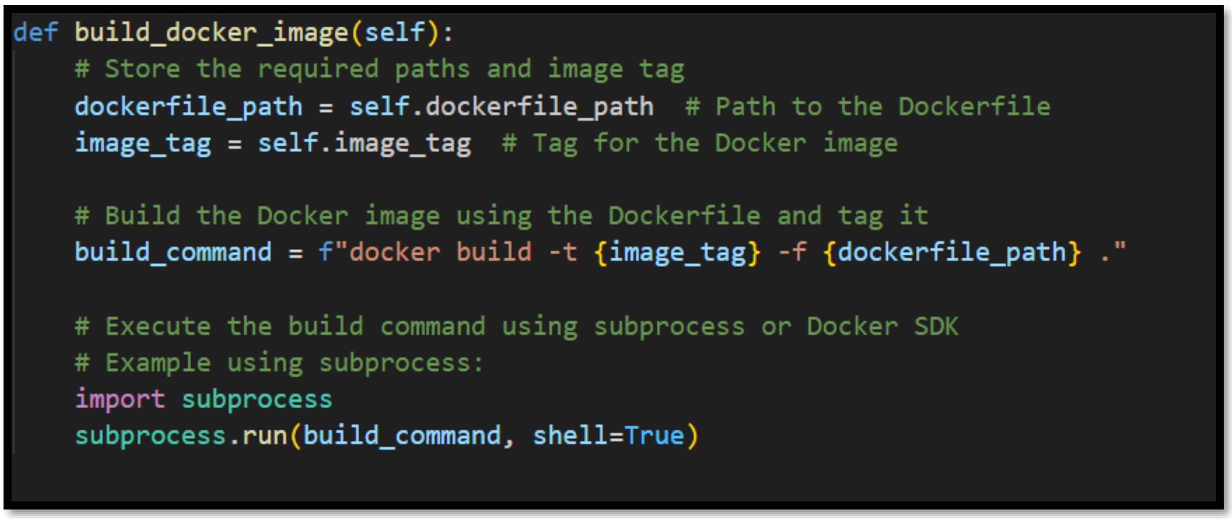
For now the snippet of generation of docker image would look like this but this might change drastically depending on the image preferences of the Chameleon infrastructure:



```
def generate_dockerfile(self):
    # Store the required paths
    dockerfile_path = self.dockerfile_path  # Path to the Dockerfile

    # Define the Dockerfile content with necessary commands
    dockerfile_content = """
    FROM ubuntu:latest
    # Add commands to install dependencies and set up the environment
    # Example: RUN apt-get update && apt-get install -y python3
    # RUN command to copy dataset into the container
    COPY ROOT/Workflow/dataset /path/inside/container

    # Add command to run automate.sh
    RUN bash ROOT/automate.sh

    # Add any other commands needed for setup
    """

    # Write the Dockerfile content to the Dockerfile
    with open(dockerfile_path, 'w') as f:
        f.write(dockerfile_content)
```

The Docker image build function:

With the Dockerfile created, the next step is to build an image using the generated Dockerfile. This process can be facilitated using the Python subprocess library, which allows executing bash commands directly from within the program.



```
def build_docker_image(self):
    # Store the required paths and image tag
    dockerfile_path = self.dockerfile_path  # Path to the Dockerfile
    image_tag = self.image_tag  # Tag for the Docker image

    # Build the Docker image using the Dockerfile and tag it
    build_command = f"docker build -t {image_tag} -f {dockerfile_path} ."

    # Execute the build command using subprocess or Docker SDK
    # Example using subprocess:
    import subprocess
    subprocess.run(build_command, shell=True)
```

The project has now laid out a comprehensive process for automating the reproducibility of experiments using the Reproducibility Service (RS). By parsing through the RO-Crate metadata, extracting necessary information, and generating essential files like automate.sh and Dockerfile, the RS program streamlines the setup of dependencies and the creation of Chameleon-compatible Docker images. With these tools in place, researchers can seamlessly reproduce experiments in a consistent and efficient manner, ensuring reproducibility and reliability in scientific workflows.

2.3 Comment on Trovi Integration:

The incorporation of Trovi into the RS project is presented as a beneficial option to consider rather than a mandatory requirement. Trovi serves as a valuable tool within the Chameleon infrastructure, offering features that can significantly enhance the reproducibility service. Here's a breakdown of how Trovi can be integrated and the benefits it brings:

About Trovi:

- Trovi is a sharing portal within Chameleon designed for collaborative sharing of digital research and educational artifacts.
- It facilitates uploading, packaging, and sharing of experiments, tutorials, class materials, and other digital content.
- Trovi offers artifact browsing, packaging, version control, sharing settings adjustment, and integration with Git and Zenodo.

Incorporation into the RS Project:

- **Artifact Packaging:** Trovi's functionality can be used to package and upload experiment artifacts generated by the RS program.
- **Version Control:** Utilize Trovi's version control features for managing different artifact versions, tracking changes, and maintaining a history of modifications.
- **Sharing and Collaboration:** Trovi's sharing settings allow control over access to artifacts, enabling collaboration and contribution from collaborators.
- **Zenodo Integration:** For artifacts needing permanent hosting and citation, Trovi's integration with Zenodo provides a seamless way to publish artifacts with DOIs.
- **Importing and Exporting:** Trovi supports importing artifacts from Git repositories and exporting artifacts via Git for versioning and collaborative development.

Benefits to the Project:

- **Centralized Management:** Trovi acts as a centralized repository, streamlining artifact storage, access, and sharing.
- **Versioning and Traceability:** Trovi's version control ensures traceability and reproducibility by maintaining a history of artifact changes.
- **Collaborative Workflows:** Trovi facilitates teamwork and knowledge sharing among project collaborators.
- **Integration with External Repositories:** Trovi extends the project's reach to external repositories, enhancing artifact accessibility and preservation.
- **Enhanced Visibility and Attribution:** Trovi's integration with Zenodo improves artifact visibility, citation potential, and proper attribution.

In summary, integrating Trovi into the RS project aligns with the goal of enhancing reproducibility, collaboration, and accessibility within the research and educational community, providing a robust platform for managing experiment artifacts and fostering open science practices.

Project Timeline

3.1 Deliverables and Scheduling:

The project deliverables are divided into 4 major sections as mentioned by the mentor in the project problem statement, which are as follows:

1. Study the different environments and specifications (COMPSs, RO-Crate, Chameleon, Trovi, …)
2. Design the most appropriate integration, considering all the elements involved.
3. Integrate PyCOMPSs basic experiments reproducibility in Chameleon.
4. Integrate PyCOMPSs complex experiments reproducibility in Chameleon (i.e. with third party software dependencies)

The initial task will require 2 weeks as it entails setting up and exploring various local technologies on my machine to understand their functionality.

The second task is expected to take approximately 5 weeks as it forms the core of the project, involving the design of the integration aspect that was briefly outlined in the proposal section.

The third and fourth tasks are estimated to take 5 weeks total (2+3) with some flexibility, as the completion of one deliverable acts as a precursor to the other.

In addition to the same, I would spend approximately **40%** of the time researching about the technologies involved, **35%** of the time during the actual implementation, **15%** in adding proper documentation and extensive and quality test (to make sure all the features work as they are supposed to and do not break functionality) and rest **10%** of my time in correcting the bugs that maybe created due to the same.

3.2 Detailed Project Scheduling:

Week 1-2	Week 1: • Study the pyCOMPSs documentation as per project requirements. • To locally set up the environment for running pyCOMPSs. • Study RO-Crate profiles in detail. Week 2: • To explore basic use cases of Chameleon and conduct simple pyCOMPSs experiments on it. • Study in detail about Trovi sharing portal on Chameleon and Trovi artifacts.
Week 3-7	Week 3: • Reviewing the proposed idea (mentioned in the 'Proposal Abstract') based on previous week's research. • Update and discuss any changes with the mentor. • To research about python libraries involved or searching for better alternatives. Week 4: • Making the basic structure directory of the RS program

	• To create the structure of ReproducibilityService Class • Create an efficient parsing function for the ro-crate JSON file. Week 5: • Conduct further research on RO-Crate profiles to devise an effective method for extracting relevant information. (required for reproducibility) • Develop a mechanism to store this information for use in the parsing function. Week 6: • Conduct comprehensive research on Chameleon-compatible images. • Design a basic Dockerfile structure for such an image. • Implement the automated shell function, continuing from the previous week's progress. Week 7: • Implement the Docker image build function. • Integrate different methods of the RS program and analyse its functionality. • Generate a report and discuss the level of automation achieved in the RS program with the mentor. • Discuss potential automation enhancements with the mentor.
MID-TERM EVALUATION	
Week-8-9	Week 8: • Researching about Trovi in further detail and incorporating it with the RS program. • Intergrating basic PyCOMPSs experiments reproducibility in Chameleon using the RS program. • Investigate any bugs encountered during the integration process. Week 9: • Update the RS program based on previous bug fixes. • Incorporate additional automation features within the RS program. • Repeat the earlier experiment for validation. • Create a report and discuss the results with the mentor.
Week 10-12	Week 10: • Study the RS program to ensure compatibility for reproducing complex experiments with third-party software dependencies. • Develop a work plan based on the study's findings and discuss its feasibility with the mentor, adjusting the plan accordingly for the upcoming weeks. Week 11 and 12: • Adjust RS functions to accommodate complex experiment reproducibility as required within the specified timeframe. • Add any feature/functionality as per the mentor’s need.
PROJECT COMPLETION AND FINAL SUBMISSION	

NOTE:

- The Timeline/Plan is not rigid and is flexible as per the discussion during weekly deliverables with the mentor.
- The Timeline is drafted considering the delays in work and any changes that must be made during the Coding period so that the Project can be completed earlier than the Timeline states.

4.1 More about me and Why me?

I am an 2nd year student at **IIT BHU**, which is one of the most prominent institute in India and secured a Rank of **1868 in JEE Advance** which more than a million people give,

My technology stack has a variety of technologies such as

Backend: **Node.js, Express.js**

Frontend: **HTML, CSS, JQuery**

Programming language: **Python, C/C++, RUST, Javascript** and beginner on **JAVA**

I am also an **active member of COPS**(Club of Programmers) here at my institute. The whole point of this club is to create a student-led community that helps other students and novices in the technical aspects and provides a shared ground for everyone to upskill themselves.

I have secured **A-grade on all programming related courses** taught till now and have done an **exploratory project** on Digital Image Processing.

I also practice DSA and competitive programming, and I am rated **almost Specialist (1386) on** Codeforces which can be used to justify my problem-solving skills.

RELEVANT COURSEWORK:

- Information Technology and Computing Workshop (About Python and Java programming language)
- Operating Systems
- Algorithms
- Data Structures
- Computer System Organization
- Computer Programming (About C language)
- Exploratory project on Digital Image processing: Link

4.2 My Projects:

PASSWORD VAULT

Language: Python

The password vault script uses **SQLite** for database storage, **hashlib** for password hashing, **Tkinter** for GUI, and **simpledialog** for pop-up inputs.

It includes functions for

- creating a master password,
- logging in,
- saving passwords securely
- displaying/removing stored passwords

NEWSLETTER SIGNUP

The project is a Newsletter Signup using **Node.js** and **the Mailchimp API**. It involves setting up a Node.js server with Express, integrating Axios for API requests, and handling email subscriptions by sending data to Mailchimp's mailing list using their API endpoints.

MULTI-THREADED RUST BASED SERVER

This project is a simple HTTP server implemented in Rust using the standard library (std). It creates a thread pool to handle incoming TCP connections concurrently.

- Binds the server to 127.0.0.1:7878.
- Creates a thread pool with 4 threads
- Accepts incoming TCP connections in a loop and dispatches them to the thread pool for handling

OPERATING SYSTEM EXPERERIMENTS

I have been learning more about Parallel Programming under the Operating Systems Course I have this semester, where I worked on implementing several **Parallel Programming** and **Deadlock Prevention** algorithms. I am well versed in Parallel Programming concepts like **mutex locks, semaphores, deadlocks etc**. Programs I have implemented in due course of time using C/C++ include :

- Producer-Consumer Problem using Mutex Locks, Semaphores and Threading.
- Banker's Algorithm using Threading
- Matrix Multiplication using Threading
- Process communication using Pipes
- Process Scheduling

NOTE- All the repositories can be found at my github: Minimega12121

4.3 Commitments:

- **How many hours will you work per week on your GSoC project?**
  I am planning to spend 30-40 hours or more on this project per week
- **Other Commitments**
  I have no other commitments during the GSoC period for this summer.
- **Do you plan to apply for any other organisation for GSoC ’24 ?**
  I am only applying to UC OSPO for GSoC’24 and have no plan to contribute to any other organisation
- **If you’re selected as a GSoC student, would you like to work on other tasks besides the projects of your choice?**
  Yes, I would love to work on other tasks that are not related to my GSoC project.
- **If you’re not selected as a GSoC student, would you like to work on the projects as a general contributor?**
  Yes, even if I am not selected as a GSoC participant, I would happily want to contribute to this project.
- **Would you like to contribute to UC OSPO in the long term, after the GSoC program ends?**
  I am really fascinated by the COMPSs technology and would love to contribute to it’s codebase on github and get to know it’s working.
- **What motivated you the most towards applying for GSoC?**
  Google Summer of Code (GSoC) allows us to gain real-world software development experience and contribute to open-source projects. It allows us to work on challenging projects with experienced mentors, develop their technical skills, and build their professional network. I participated in GSoC'24, with this project only, and had a wonderful learning experience with the mentors.

4.4 My Availability:

I have no other commitments this summer, so GSoC will be my only priority, Also I am working on this project only for GSOC-2024, so I can provide my total commitments to this project

I will be able to devote ample time to work on the project. I am comfortable with working hours but prefer to work during office hours, i.e. (9:00 -14:00) and (16:00-19:00) +5:30 GMT.(flexible)

Also, I have a sheer interest in Open Source and am sure that I will land upon the expectations of my mentors and my organization.

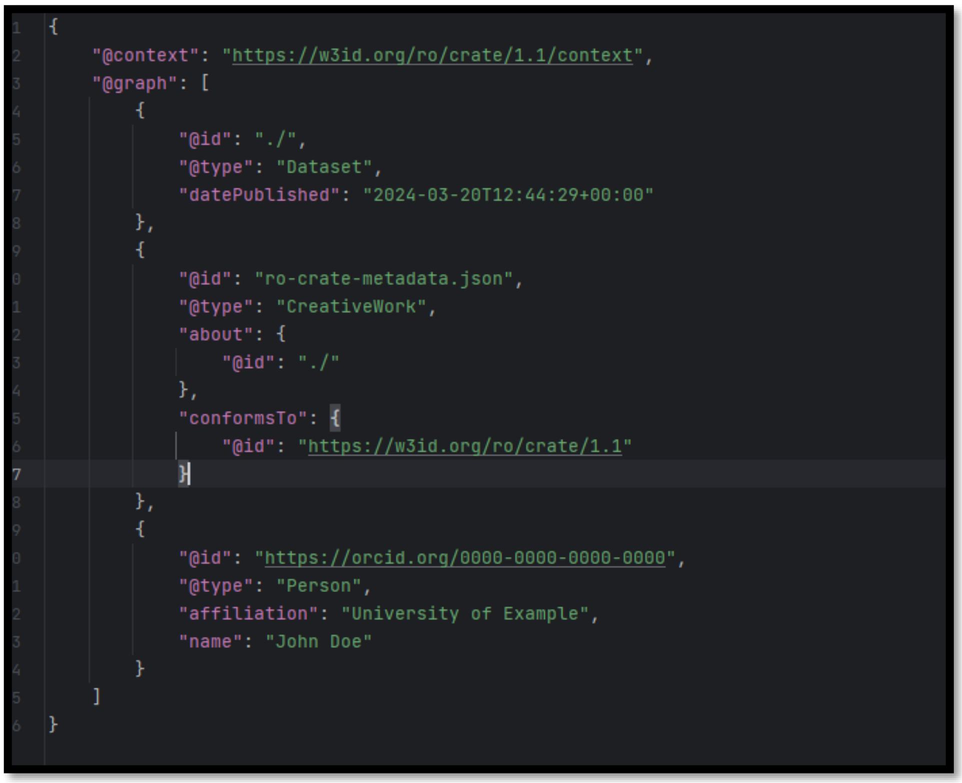
Post GSoC:

Post GSoC I would love to **contribute to COMPSs** codebase and other open-source projects organized by OSPO also. I would also love to further collaborate on this project to **add more features** to the **RS program** in the future.

Some Local Tested Results:

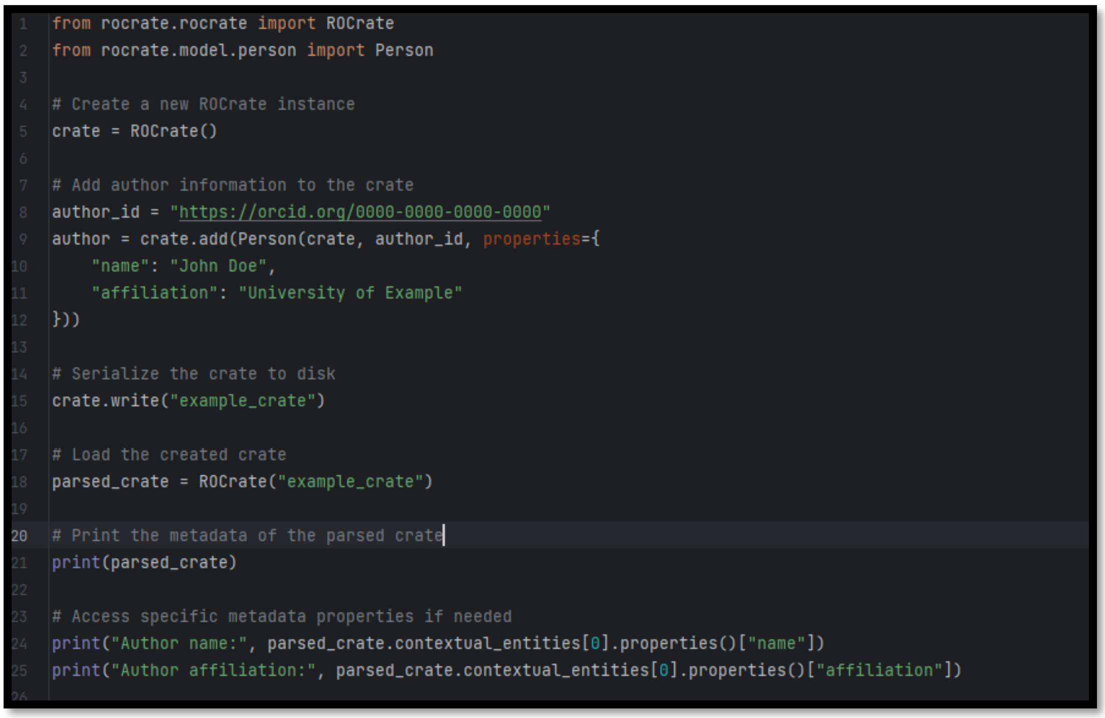
(1) Parsing on an example ro-crate created by me:

Example crate created:



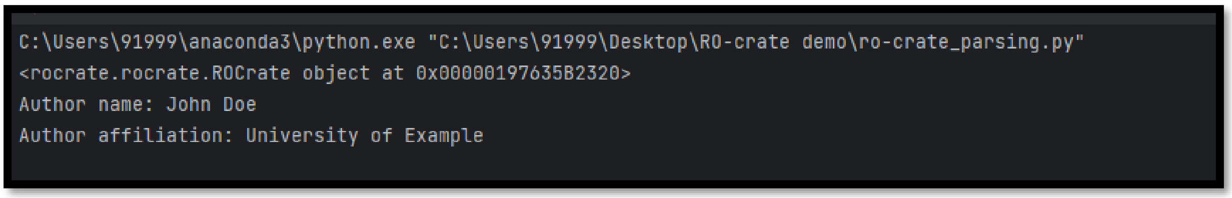
```json
{
    "@context": "https://w3id.org/ro/crate/1.1/context",
    "@graph": [
        {
            "@id": "./",
            "@type": "Dataset",
            "datePublished": "2024-03-20T12:44:29+00:00"
        },
        {
            "@id": "ro-crate-metadata.json",
            "@type": "CreativeWork",
            "about": {
                "@id": "./"
            },
            "conformsTo": {
                "@id": "https://w3id.org/ro/crate/1.1"
            }
        },
        {
            "@id": "https://orcid.org/0000-0000-0000-0000",
            "@type": "Person",
            "affiliation": "University of Example",
            "name": "John Doe"
        }
    ]
}
```

Python code used:



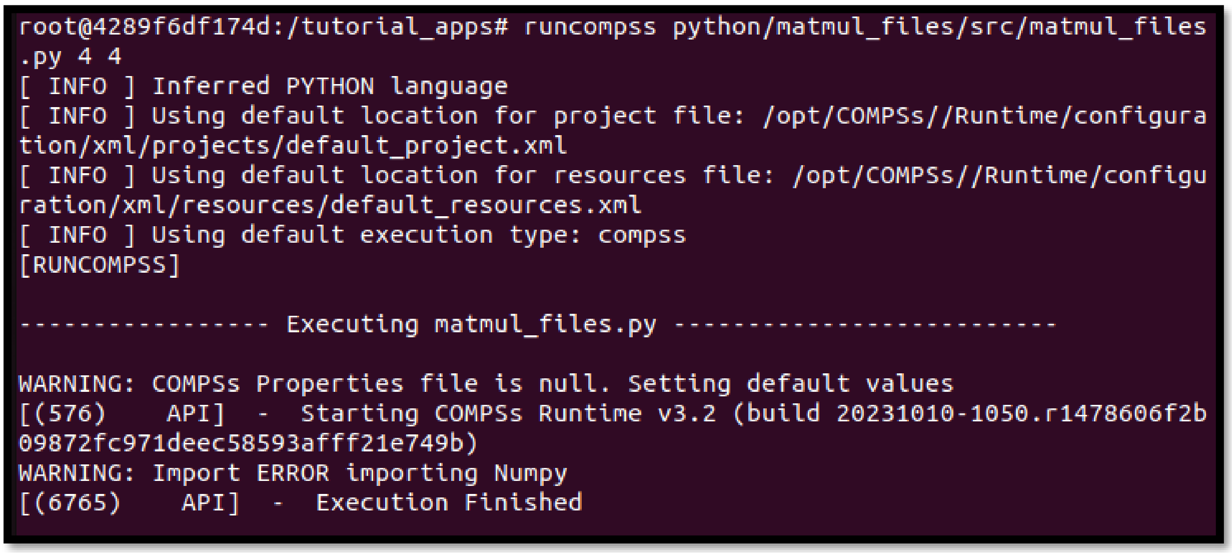
```python
from rocrate.rocrate import ROCrate
from rocrate.model.person import Person

# Create a new ROCrate instance
crate = ROCrate()

# Add author information to the crate
author_id = "https://orcid.org/0000-0000-0000-0000"
author = crate.add(Person(crate, author_id, properties={
    "name": "John Doe",
    "affiliation": "University of Example"
}))

# Serialize the crate to disk
crate.write("example_crate")

# Load the created crate
parsed_crate = ROCrate("example_crate")

# Print the metadata of the parsed crate
print(parsed_crate)

# Access specific metadata properties if needed
print("Author name:", parsed_crate.contextual_entities[0].properties()["name"])
print("Author affiliation:", parsed_crate.contextual_entities[0].properties()["affiliation"])
```

Output produced:



```
C:\Users\91999\anaconda3\python.exe "C:\Users\91999\Desktop\RO-crate demo\ro-crate_parsing.py"
<rocrate.rocrate.ROCrate object at 0x00000197635B2320>
Author name: John Doe
Author affiliation: University of Example
```

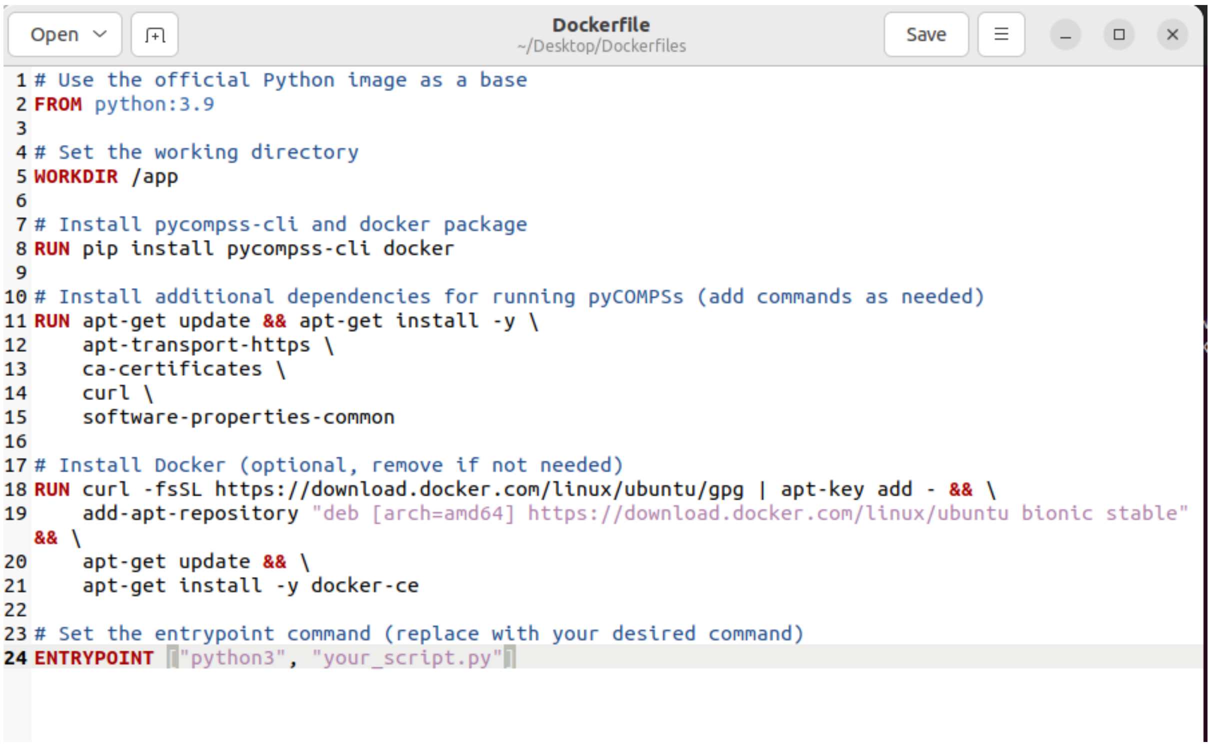
(2) Running COMPSs basic experiment locally on Ubuntu Virtual Machine:



```
root@4289f6df174d:/tutorial_apps# runcompss python/matmul_files/src/matmul_files
.py 4 4
[ INFO ] Inferred PYTHON language
[ INFO ] Using default location for project file: /opt/COMPSs//Runtime/configura
tion/xml/projects/default_project.xml
[ INFO ] Using default location for resources file: /opt/COMPSs//Runtime/configu
ration/xml/resources/default_resources.xml
[ INFO ] Using default execution type: compss
[RUNCOMPSS]

----------------- Executing matmul_files.py --------------------------

WARNING: COMPSs Properties file is null. Setting default values
[(576)    API]  -  Starting COMPSs Runtime v3.2 (build 20231010-1050.r1478606f2b
09872fc971deec58593afff21e749b)
WARNING: Import ERROR importing Numpy
[(6765)    API]  -  Execution Finished
```

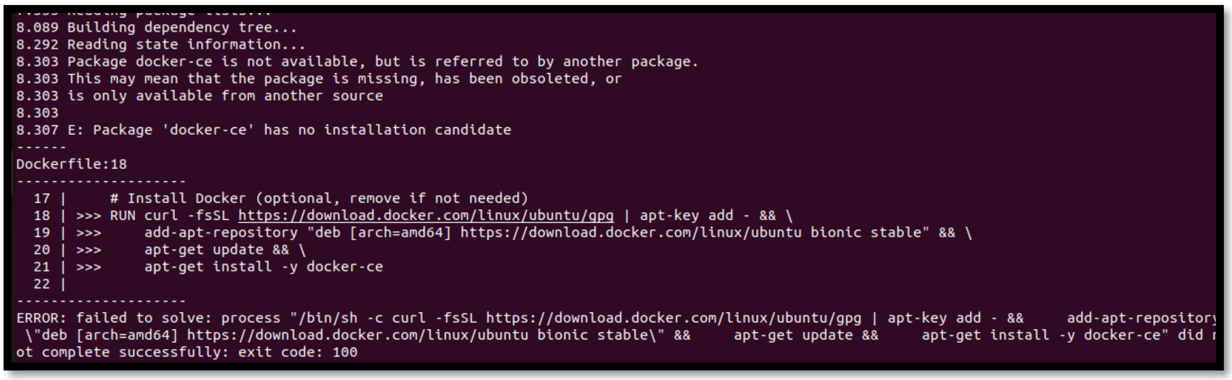
(3)Exploring Docker and making a default docker-image with required dependencies to run COMPSs:

Dockerfile:



```
# Use the official Python image as a base
FROM python:3.9

# Set the working directory
WORKDIR /app

# Install pycompss-cli and docker package
RUN pip install pycompss-cli docker

# Install additional dependencies for running pyCOMPSs (add commands as needed)
RUN apt-get update && apt-get install -y \
    apt-transport-https \
    ca-certificates \
    curl \
    software-properties-common

# Install Docker (optional, remove if not needed)
RUN curl -fsSL https://download.docker.com/linux/ubuntu/gpg | apt-key add - && \
    add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu bionic stable"
&& \
    apt-get update && \
    apt-get install -y docker-ce

# Set the entrypoint command (replace with your desired command)
ENTRYPOINT ["python3", "your_script.py"]
```

Image created:



```
8.089 Building dependency tree...
8.292 Reading state information...
8.303 Package docker-ce is not available, but is referred to by another package.
8.303 This may mean that the package is missing, has been obsoleted, or
8.303 is only available from another source
8.303
8.307 E: Package 'docker-ce' has no installation candidate
------
Dockerfile:18
--------------------
  17 |     # Install Docker (optional, remove if not needed)
  18 | >>> RUN curl -fsSL https://download.docker.com/linux/ubuntu/gpg | apt-key add - && \
  19 | >>>     add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu bionic stable" && \
  20 | >>>     apt-get update && \
  21 | >>>     apt-get install -y docker-ce
  22 |
--------------------
ERROR: failed to solve: process "/bin/sh -c curl -fsSL https://download.docker.com/linux/ubuntu/gpg | apt-key add - &&     add-apt-repository
 \"deb [arch=amd64] https://download.docker.com/linux/ubuntu bionic stable\" &&     apt-get update &&     apt-get install -y docker-ce" did r
ot complete successfully: exit code: 100
```

References:

- **About RO-Crate:**

  https://www.researchobject.org/ro-crate/1.1/

- **About COMPSs (and task-dependency graph image):**

  https://zenodo.org/records/10046567

- **About Chameleon:**

  https://chameleoncloud.readthedocs.io/en/latest/index.html\

- **Research paper about Workflow profiles:**

  https://www.researchobject.org/workflow-run-crate/

In closing, I would like to express my heartfelt gratitude to Mentor Raül Sirvent and the entire UC OSPO community for providing me with the invaluable opportunity and guidance throughout this proposal process. Your support, expertise, and encouragement have been instrumental in shaping this proposal into a comprehensive and impactful document. I am truly honoured to have had the chance to work alongside such dedicated individuals, and I look forward to continuing our collaboration in the future. Thank you once again for your unwavering support.