



COMP.SE.140 - Continuous Development
And Deployment

Fall 2024 - Final Project Report

Yang Si Jun

Table of Contents

1. Instructions for the Examiner

- [1.1 Optional Features implemented](#)
- [1.2 Steps to Test the System](#)
- [1.3 Steps to Test the CI/CD Pipeline](#)
- [1.4 Platform used in Development](#)

2. Description of the CI/CD Pipeline

- [2.1 Version Management](#)
- [2.2 Build Tools](#)
- [2.3 Testing](#)
 - [2.3.1 Tools Used](#)
 - [2.3.2 Test Cases](#)
- [2.4 Deployment](#)
- [2.5 Operating and Monitoring](#)

3. Example Runs of the Pipeline

- [3.1 Logs of Failing Tests](#)
- [3.2 Logs of Passing Tests](#)

4. Reflections

- [4.1 Main Learnings](#)
- [4.2 Challenges and Difficulties](#)
- [4.3 Suggested Improvements](#)
- [4.4 Effort Estimation \(in Hours\)](#)

1. Instructions for the Teaching Assistant

1.1 Optional Features implemented

A few optional features were implemented in this project. Firstly, static analysis was performed during the pipeline using pylint. This is performed in a separate linting stage in our .gitlab-ci.yml. This helped to ensure good coding practices for good readability and handover. Another optional feature implemented was logging for troubleshooting. Logging was done when a user accessed an endpoint as well as if an exception or error popped up, formatted with the time together with the message. The last optional feature I did was to do proper deployment(non-local) on a CSC machine. The Teaching Assistant is able to access the application and endpoints via the public IP instead of just localhost.

1.2 Steps to test the system

1. Clone the Repository

Clone the project repository to your local machine:

```
git clone -b project
https://compse140.devops-gitlab.rd.tuni.fi/zedithx/final-project
cd <created folder>
```

2. Build and Start the System

Rebuild the containers from scratch and start the system in detached mode:

```
docker-compose build --no-cache
docker-compose up -d
```

1.3 Steps to Test the CI/CD Pipeline

1. Add Remote Repository

Add the GitLab remote repository to your local Git configuration:

```
git remote add origin
https://compse140.devops-gitlab.rd.tuni.fi/zedithx/final-project
```

2. Make a Change

Make a small change to your code to trigger the pipeline.

3. Commit and Push the Change

Commit your changes and push them to the project branch:

```
git commit -m "Testing pipeline"
git push origin project
```

This will trigger the CI/CD pipeline, which has build, linting, test and deploy stages. The deploy stage will deploy the containerized application on a CSC Machine where you can access the endpoints through its public IP. You can check the logs in the gitlab link under build > pipeline

4. Test the endpoint

You can test the endpoint accordingly with the public floating ip: 193.166.24.237 and the port 8197.

```
curl -u user1:Password1 -X PUT -H "Content-Type: text/plain"
http://193.166.24.237:8197/state -d "RUNNING"
curl -X GET http://193.166.24.237:8197/state
curl -X GET http://193.166.24.237:8197/request
curl -X GET http://193.166.24.237:8197/run-log
```

You can also test the browser UI previously done which is served at port 8198 by typing 193.166.24.237:8198 in your browser.

Remember to change the state to “SHUTDOWN” when done

Note:

1. Credentials are only needed when system needs to be changed from init state
2. When system is in paused state, all requests should not go through except for request to move it to “RUNNING” or “INIT” state

1.4 Platform used in development

Regarding the platform used, below are the specifications of it

Hardware: Apple M1 Pro

CPU Architecture: arm64

OS: Sequoia 15.1.1

Docker version: 27.1.1

Docker-compose version: v2.29.1-desktop.1

2. Description of CI/CD Pipeline

2.1 Version Management

For version management, our main branch is project. To ensure that we do not constantly deploy to production, we also have a develop branch that does not run the deploy stage of the CICD pipeline. Serving as our development upstream branch, we have further branches from the develop branch that target new features. For example, we have a “sharedvol” branch that experimented with using shared volumes to write and share the state of the system between 3 replicas of service1. As we complete these features as new branches, we make sure everything is correct through testing before merging it with develop. After making significant progress with anything broken, we then merge it to project, ensuring that the service is not faulty on production deployment.

An important thing about making new branches for every new feature is that if the new feature does not work out, we can always ditch it and delete the branch, so there will be no risk in the code becoming too messy. This way, features that have multiple ways of implementing need not much worry.

2.2 Build Tools

For building of the application, we have 2 dockerfiles. One dockerfile is for the python service thats internally exposed and serves most of the api requests, while the other dockerfile is for the ruby service that is only internally exposed and called by the python service(Figure 1.1).

Two docker-compose file in the main directory helps to containerise both containers and the nginx api gateway, helping to build the entire system with one command(Figure 1.2). One is used for the test environment: docker-compose.test.yml, where the only difference is that for the test compose file, our states stored in the txt file is mapped to the /dev/shm volume(temporary space) rather than our actual local files of the docker container. This is so that during testing, our code do not actually write to the state file or the state logs, but rather just a temporary space in the docker container.

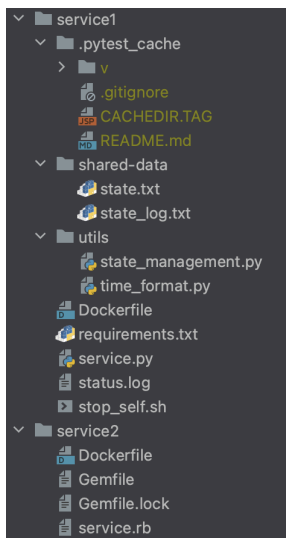


Figure 1.1

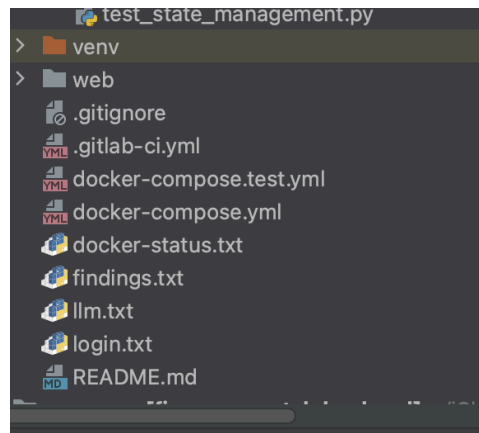


Figure 1.2

2.3 Testing

2.3.1 Tools used

For testing, we only performed integration testing, covering all the different cases for each api call exposed in service1, which is our python service. To do this, we use pytest to execute our test code. We take note to do it in a Test Driven Development approach, writing the test cases first as well as its doc string, before actually writing code for the api endpoints. We split our test cases into two files, one being test_request.py and another being test_state_management.py. The test_request.py contains the test case for the request endpoint, which was written in the previous iteration, while the test_state_management.py contains the test cases for the new feature of state we were supposed to implement. This ensures that the code is written as modular as possible, as well as easy to navigate.

2.3.2 Test cases

test_request.py

Endpoint: GET /request

```
def test_request(self):
    """
    GET /request
    Similar function as REQUEST-button of the GUI (see instructions for nginx exercise),
    but as a text/plain response to the requester.
    """
    response = requests.get(f"{BASE_URL}/request", auth=HTTPBasicAuth(USERNAME, PASSWORD))
    assert response.status_code == 200
    response_text = response.text
    assert "Service 1" in response_text
    assert "Service 2" in response_text
    assert "IP Address" in response_text
    assert "Processes" in response_text
    assert "Disk Space" in response_text
    assert "Time since last boot" in response_text

    # Extract the IP Address
    ip_address = json.loads(response_text).get("Service 1", {}).get("IP Address", None)

    # Validate the presence of an IP address using regex
    ip_pattern = r"^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$"
    assert re.match(ip_pattern, ip_address), f"Invalid IP Address: {ip_address}"

    # Validate that uptime includes "minutes" or "hours"
    assert "minutes" in response_text or "hours" in response_text, "Uptime not in expected format"
```

The code tests that the expected format of response is present, as well as using regex to test the presence of an ip address. Since the responses are generally dynamic such as time since last boot, we cannot test for some of these actual results.

test_state_management.py

Fixture

```
Yang Si Jun
@pytest.fixture
def reset_state():
    """Fixture to reset state to INIT before each test."""
    requests.put(f"{BASE_URL}/state", data="INIT", auth=HTTPBasicAuth(USERNAME, PASSWORD))
    yield
```

We have a fixture that resets the state to INIT before each test. This is used for all the apis as we do not want our PUT /state endpoint to alter the state and affect our apis. For example, if our test case changes the state to PAUSED, all the apis would then fail. Hence all test cases should start from INIT so that they are independent of any circumstances.

Endpoint: PUT /state

```
class TestPutState:
    """
    PUT /state (payload "INIT", "PAUSED", "RUNNING", "SHUTDOWN")
    PAUSED = the system does not response to requests
    RUNNING = the system responses to requests normally
    If the new state is equal to previous nothing happens.
    There are two special cases:
    INIT = everything (except log information for /run-log) is set to the initial state and
    login is needed get the system running again.
    SHUTDOWN = all containers are stopped
    """
    Yang Si Jun
    def test_put_state_running(self, reset_state):
        response = requests.put(f"{BASE_URL}/state", data="RUNNING", auth=HTTPBasicAuth(USERNAME, PASSWORD))
        assert response.status_code == 200
        response_json = response.json() # Convert response to JSON
        assert response_json["message"] == "State updated to RUNNING"

    Yang Si Jun
    def test_put_state_init(self, reset_state):
        response = requests.put(f"{BASE_URL}/state", data="INIT", auth=HTTPBasicAuth(USERNAME, PASSWORD)) # No state change
        assert response.status_code == 200 # Should not trigger an error
        response_json = response.json() # Convert response to JSON
        assert response_json["message"] == "No change in state"

    Yang Si Jun
    def test_put_state_paused(self, reset_state):
        response = requests.put(f"{BASE_URL}/state", data="PAUSED", auth=HTTPBasicAuth(USERNAME, PASSWORD)) # No state change
        assert response.status_code == 200 # Should not trigger an error
        response_json = response.json()
        assert response_json["message"] == "State updated to PAUSED"
```

All test cases under the PUT /state is placed in a class under TestPutState for better readability and organisation. We have the generic test cases that test the change of state from init to all the other states except shutdown.

```

Yang Si Jun
def test_init_failed_authentication(self, reset_state):
    """In INIT state, shouldn't work without authentication"""
    response = requests.put(f"{BASE_URL}/state", data="RUNNING")
    assert response.status_code == 401

Yang Si Jun
def test_no_authentication_needed(self, reset_state):
    """In other states, you should not need authentication"""
    requests.put(f"{BASE_URL}/state", data="RUNNING", auth=HTTPBasicAuth(USERNAME, PASSWORD))
    # When state is running already, authentication should not be needed
    response = requests.put(f"{BASE_URL}/state", data="PAUSED")
    assert response.status_code == 200
    response_json = response.json() # Convert response to JSON
    assert response_json["message"] == "State updated to PAUSED"

Yang Si Jun
@patch("requests.put")
def test_put_state_shutdown_mocked(self, mock_put, reset_state):
    mock_put.return_value.status_code = 200
    mock_put.return_value.json = lambda: {"message": "State updated to SHUTDOWN"}

    response = requests.put(f"{BASE_URL}/state", data="SHUTDOWN", auth=HTTPBasicAuth(USERNAME, PASSWORD))
    assert response.status_code == 200
    response_json = response.json()
    assert response_json["message"] == "State updated to SHUTDOWN"

```

We then have a test case for failed authentication: `test_init_failed_authentication`, where the username and password is not given to change state from init, which was one of the specifications of the project.

We also have a test case where no authentication should be needed in other states: `test_no_authentication_needed`. We make two requests in this test case as we always start from init as discussed earlier in our fixture.

Lastly, we test our change in state to shutdown. Since shutdown actually shutdown and removes the service containers, we need to mock the response so that it does not actually shutdown. Otherwise, the rest of our test cases would not even be able to run

Endpoint: GET /state

```
class TestGetState:
    Yang Si Jun
    def test_get_state(self, reset_state):
        """
        GET /state (as text/plain)
        get the value of state
        """
        response = requests.get(f"{BASE_URL}/state")
        assert response.status_code == 200
        assert response.text == "INIT"

    Yang Si Jun
    def test_get_state_paused(self, reset_state):
        """
        GET /state (as text/plain)
        Should not work when the system is paused
        """
        requests.put(f"{BASE_URL}/state", data="PAUSED", auth=HTTPBasicAuth(USERNAME, PASSWORD))
        response = requests.get(f"{BASE_URL}/state")
        assert response.status_code == 401
```

All test cases under the GET /state is placed in a class under TestGetState for better readability and organisation. The first test case tests for a normal get state, while the second one tests for when the system is in the PAUSED state. As specified, the system should not be able to respond when it is in paused except for changes of state.

Endpoint: GET /run-log

```
Yang Si Jun *
class TestGetStateLog:
    Yang Si Jun *
    @responses.activate
    def test_get_state_log_mocked(self, reset_state):
        """
        GET /run-log (as text/plain)
        """
        responses.assert_all_requests_are_fired = True
        responses.add(
            responses.GET,
            f"{BASE_URL}/run-log",
            body="2023-11-01T06:35:01.380Z: INIT->RUNNING\n"
                "2023-11-01T06:40:01.373Z: RUNNING->PAUSED\n"
                "2023-11-01T06:40:01.373Z: PAUSED->RUNNING",
            status=200,
            content_type="text/plain"
        )

        response = requests.get(f"{BASE_URL}/run-log")
        assert response.status_code == 200
```

Once again, run-log is placed in a separate test class. Since the run-log actually reads from the file, we mock the api response since we don't know the time as well as what state changes occur before this test case.

2.4 Deployment

For deployment, I utilized the CSC virtual machine instance, as suggested (commonly used by Finnish students). The following steps were undertaken to configure the deployment pipeline:

1. Setup of CSC Virtual Machine:

- Configured the CSC VM to run Docker, including installing and enabling Docker services.
- Re-registered the GitLab Runner to operate on the CSC virtual machine instead of the local machine. This ensures that pipeline executions occur on the CSC machine directly.

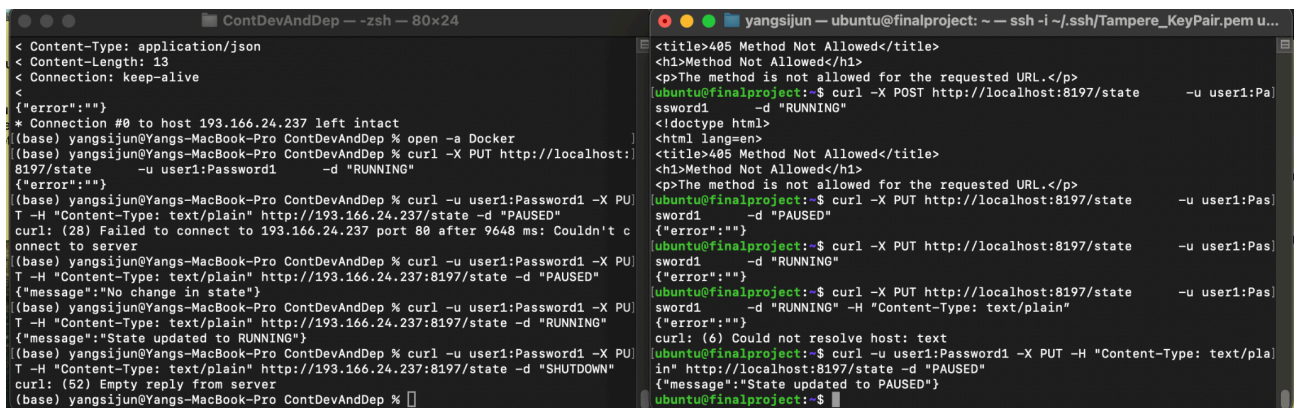
2. Pipeline Integration:

- Modified the GitLab CI/CD pipeline to export the PATH of gitlab runner to detect the scripts installed. Upon any push to the repository, the pipeline automatically runs on the CSC machine, deploying the containerized application.

3. Public Access Configuration:

- Updated the security group assigned to the VM to allow public access to the service:
 - Opened **port 8197** for the NGINX API gateway to serve as the entry point for external clients.
 - Added an **SSH rule** to allow administrative access for setup and troubleshooting purposes.

By leveraging the CSC virtual machine, the deployment pipeline was effectively automated and made publicly accessible while ensuring a secure configuration.



```
ContDevAndDep --zsh -- 80x24
< Content-Type: application/json
< Content-Length: 13
< Connection: keep-alive
{
  "error": ""
}
* Connection #0 to host 193.166.24.237 left intact
(base) yangsijun@Yangs-MacBook-Pro ContDevAndDep % open -a Docker
(base) yangsijun@Yangs-MacBook-Pro ContDevAndDep % curl -X PUT http://localhost:8197/state -u user1:Password1 -d "RUNNING"
{"error": ""}
(base) yangsijun@Yangs-MacBook-Pro ContDevAndDep % curl -u user1:Password1 -X PUT -H "Content-Type: text/plain" http://193.166.24.237/state -d "PAUSED"
curl: (28) Failed to connect to 193.166.24.237 port 80 after 9648 ms: Couldn't connect to server
(base) yangsijun@Yangs-MacBook-Pro ContDevAndDep % curl -u user1:Password1 -X PUT -H "Content-Type: text/plain" http://193.166.24.237:8197/state -d "PAUSED"
{"message": "No change in state"}
(base) yangsijun@Yangs-MacBook-Pro ContDevAndDep % curl -u user1:Password1 -X PUT -H "Content-Type: text/plain" http://193.166.24.237:8197/state -d "RUNNING"
{"message": "State updated to RUNNING"}
(base) yangsijun@Yangs-MacBook-Pro ContDevAndDep % curl -u user1:Password1 -X PUT -H "Content-Type: text/plain" http://193.166.24.237:8197/state -d "SHUTDOWN"
curl: (52) Empty reply from server
(base) yangsijun@Yangs-MacBook-Pro ContDevAndDep %

yangsijun -- ubuntu@finalproject: ~ -- ssh -i ~/.ssh/Tampere_KeyPair.pem u...
<title>405 Method Not Allowed</title>
<h1>Method Not Allowed</h1>
<p>The method is not allowed for the requested URL.</p>
[ubuntu@finalproject:~]$ curl -X POST http://localhost:8197/state -u user1:Password1 -d "RUNNING"
<!doctype html>
<html lang=en>
<title>405 Method Not Allowed</title>
<h1>Method Not Allowed</h1>
<p>The method is not allowed for the requested URL.</p>
[ubuntu@finalproject:~]$ curl -X PUT http://localhost:8197/state -u user1:Password1 -d "PAUSED"
{"error": ""}
[ubuntu@finalproject:~]$ curl -X PUT http://localhost:8197/state -u user1:Password1 -d "RUNNING"
{"error": ""}
[ubuntu@finalproject:~]$ curl -X PUT http://localhost:8197/state -u user1:Password1 -d "RUNNING" -H "Content-Type: text/plain"
{"error": ""}
curl: (6) Could not resolve host: text
[ubuntu@finalproject:~]$ curl -u user1:Password1 -X PUT -H "Content-Type: text/plain" http://localhost:8197/state -d "PAUSED"
{"message": "State updated to PAUSED"}
[ubuntu@finalproject:~]$
```

Here, I test on both the vm(right terminal) and my own computer(left terminal), ensuring that it works both internally and externally. If it worked on the vm but not through the public ip, it meant that the firewall rules were not configured properly. We can see that it works perfectly.

2.5 Operating and Monitoring

There was no operating or monitoring done due to time constraint.

→  job_test

4. Reflections

4.1 Main Learnings

I learnt many things from this project, where I got to really see under the hood of code development.

For starters, while I had done a CI/CD pipeline before using GCP's cloud trigger, writing the CI yml file was new for me. In addition, registering a local runner was something I did not know was possible as I had always relied on a given server to carry out the pipeline for deployment.

I also learnt in depth on how the nginx gateway can act as a load balancer between the replicas of services which I had never done before, since I had only used it before as a reverse proxy for my Django API service. Nginx authentication was also a new thing for me, as well as its ability to serve static pages.

One big learning point I had was the thorough understanding of containers, how they can be orchestrated by the docker-compose file to restrict public access and facilitate data transfer internally instead. This also provides security as there is only entry way which is through the nginx api gateway, making it less vulnerable to attacks. The mapping of volumes was something that I had to really understand, as there were many times my code did not run due to incorrect/missing mapping of the files to the container.

Another thing I learnt was syncing data between multiple replicas of a service, as they would give different values as the services rotated in a round robin manner based on the nginx gateway. While I learnt about the possibilities of RabbitMQ to execute a publish-subscriber model, where the working container can act as a publisher while the rest work as subscribers, the other method to have a shared volume seemed much easier. There were many other methods, but I found these two ways to be the cleanest.

The biggest takeaway for me was the importance of threading. This became especially evident while implementing the shutdown containers function. Without threading, the shutdown logic would block the main process, or worse, the process would terminate before the shutdown function had completed. Using threading allowed the shutdown logic to execute in the background, ensuring that all containers were properly stopped and removed while keeping the main application responsive.

4.2 Challenges and Difficulties

Initially there were multiple challenges face along the way during development.

Firstly, when writing the tests, some responses were difficult to obtain, such as the shutdown state – it would end the services and hence be unable to serve the remaining test cases. Hence, it had to be mocked. Another was the request, which had a different response depending on which local machine it ran on, hence we could only check for the format and the presence of a valid IP address.

Secondly, the presence of 3 replicas of service1, which served most of our api, made it impossible to have local variables for our state and statelog. As the replicas serve the requests in a round robin format, determined by our nginx api gateway, the state would be incorrect each time as it rotates. To do this, we had to sync the data shared by the services using rabbitMQ or shared volumes. Since shared volume seemed easier, I mapped files to be written to as shared volumes in the docker-compose, and it worked smoothly.

```
service1:
  volumes:
    - /var/run/docker.sock:/var/run/docker.sock
    - ./service1/shared-data:/shared-data
  # Define which docker file to use
  build: ./service1
  # Map inside port to outside port
  expose:
    - "8199"
  # Ensure Service2 is started before Service1
  depends_on:
    - service2
  networks:
    - app-network
```

Thirdly, there was issues with having the nginx api gateway requiring authentication when there is any state change from "INIT" to another state. Since the authentication was at the nginx level, we were unable to specify it in the actual api code. As a result, we had to find a way to be able to check the state on the nginx config file. While there was a javascript way of dynamically allocating nginx access to the state file so that we could do if statement checks, I did not manage to work it out. Eventually, I discovered the auth_request keyword, which redirects to another nginx entry point for checking purposes, where we could actually write python code. I experimented with checking for the INIT state first and if so, login was required using the

`error_page 401 = @auth_required` clause. However this did not work for the curl command as the redirection to another nginx endpoint meant that login was required only after the state check. This means the credentials inputted at the point of the curl command would not register. To fix this, we decided to do the authentication in the `auth_request` endpoint instead, forwarding the auth header in nginx gateway to be accessed in the request. We could then directly return a successful or failed status code.

```
@app.route('/check_state', methods=['GET'])
def check_state():
    """Check if the state is INIT. If state is init when attempting to
    change state, auth is required"""
    state_file_path = "/shared-data/state.txt"
    # Read the current state from the volume
    if not os.path.exists(state_file_path):
        logging.info("Making state file...")
        with open(state_file_path, "a", encoding="utf-8"):
            pass # Create the file if it does not exist
        state = "INIT"
    else:
        with open(state_file_path, "r", encoding="utf-8") as f:
            state = f.read()
            if not state:
                state = "INIT"
    if state == "INIT":
        auth = request.authorization
        if not auth or not (auth.username == "user1" and auth.password == "Password1"):
            return Response("Unauthorized", status=401)
    return Response("OK", status=200)
```

Another difficulty was having the nginx api gateway do selective authentication based on the request method. As specified, we were not allowed to make requests to the system except for change in state if the system was in the “PAUSED” state. To do this, we imposed the restriction at the nginx level. However, there were issues with having `auth_request`(redirects to an api for authentication) in `if` and `limit_except` blocks in the nginx config file. Eventually, I found sources that used the `rewrite` keyword in nginx config to redirect to another function, which we could then place the `auth_request`, and manipulating the `rewrite` keyword allowed us to choose authentication only for the PUT(changes state) but not GET requests(gets the current state).

```
location /state {
    if ($request_method = PUT) {
        rewrite .* /_state last;
    }
    # Authenticate PUT requests
    auth_request /check_pause;
    proxy_pass http://service1_backend;
}

location /_state {
    rewrite .* /state break;
    auth_request /check_state;
    proxy_pass http://service1_backend;
    internal;
}
```

Lastly, testing posed an issue for our solution using shared volumes. Testing should not be actually affecting the system, but because we were writing to files for the state and state log, our code actually changed the systems state and logs during our pytest. We needed a temporary/separate space that would clear itself out during testing, and hence we decided to create a test docker-compose file, mapping our shared data files to a temporary space `/dev/shm`. This way, we can be sure that nothing gets changed in our actual production container

```
service1:
  volumes:
    - /var/run/docker.sock:/var/run/docker.sock
    - /dev/shm:/shared-data
  # Define which docker file to use
  build: ./service1
  # Map inside port to outside port
  expose:
    - "8199"
  # Ensure Service2 is started before Service1
  depends_on:
    - service2
  networks:
    - app-network
```

4.3 Suggested Improvements

In terms of improvements, logging could have been done more thoroughly throughout the endpoints, such as notifying when a user is calling this endpoint. The test cases could be more extensively covered especially. While we did integration testing, we did not account for our ruby service and we did not do unit testing for functions used. Many things were also repeated in our API endpoint, and it would be better if we had converted it to utility functions, such as the opening of file and checking of state. We could then do unit testing for these functions separately to increase the maintainability and scalability of our system.

4.4 Effort Estimation (in Hours)

In total, it is estimated that I spent around 50 hours or less on the project together with the report.