1. Instructions for the teaching assistant

Implemented optional features

None

Instructions for examiner to test the system.

I have tested the system myself by copying the given URL, and running it with the specified commands, and it has worked, nothing special needed. The only thing to note is that before tests are run, one should make sure that Docker is running, as the tests startup a Docker environment, and fail if Docker can't be called.

One thing to note is that since I couldn't find a way to close down RabbitMQ and Redis in any programmatic way, those containers aren't closed when the shutdown command is issued, but all the containers I implemented are closed. This can be verified by looking at the running containers via Docker.

Tests can be run by going to the tests folder, and running the commands "npm install" and "npm test". They are a bit slow, due to starting the Docker compose environment before running.

If running the system multiple times, Redis seems to not lose state history, so run-log endpoint might show state changes that happened in past runs of the system, but the timestamps should make it easy enough to tell which changes were made during the current run. As the specification for the task didn't specify whether or not this is allowed, and due to limited time, I didn't look into how this could be avoided.

2. Description of the CI/CD pipeline

As the assignment instructions specified, I used GitLab pipelines for the ci/cd pipeline. As I was working on the project solo, I didn't use any branches other than the project branch, and as the project branch is the only one to contain the pipeline definition, it is the only branch where pushes cause a pipeline run. Each push into the branch starts up a new run of the pipeline.

No special tools were used for building, the build phase is just "docker compose build", using the Yaml file found at the root of the repository.

For testing, I decided to use JavaScript, Vitest and Testcontainers. I made this decision partly because JavaScript testing is familiar to me, and because I will most likely be using these same tools in my work project quite soon, so I wanted to get experience with them. The tests work by first running the code in vitest.setup.js, which starts the docker compose environment defined at the root of the repository. Then the tests in gateway.test.js test the entire system by sending requests to the API gateway and evaluating the results. So, the tests are end-to-end tests on the entire running system.

The tests are divided into describe blocks defining which endpoint is called, and then test blocks describing what is happening in the test, and the actual test code. Tests are implemented for each endpoint of the gateway, though only one for most endpoints. The test cases described in simple words are as follows, further evaluation of them is in chapter 4 of the report.

- Get /messages: make a request, and check that it contains the start of one of the 3 possible log messages given by the system (Sending a message failed, a message was sent, or the request for logs was made before any logs have been recorder)

- Get /state: make a request, and check that the systems state is either INIT or RUNNING (could be either one depending on timing)
- PUT /state:
    - Set the state to PAUSED, make a request to the messages endpoint, wait 2 seconds, and make another request. If the system works correctly, the responses are the same, as nothing should be happening in the system while state is PAUSED
    - Set the state to PAUSED, request the messages endpoint, then set state to RUNNING, wait for 2 seconds, and make another request to the messages endpoint. Now if everything works correctly, service 1 has sent a new message due to state being RUNNING, and the logs differ
- GET /run-log
    - Make almost all of the possible state transitions (shutdown is omitted), and then check that the response gotten from the endpoint contains all of the state transitions made, and also the transition from INIT to RUNNING that should be made automatically

The pipeline runs the tests automatically, and then if they pass, it moves into deployment phase, but if they fail the pipeline run is aborted, and marked as failed. The running of the test simply consists of moving into their directory and running “npm install” and “npm test”.

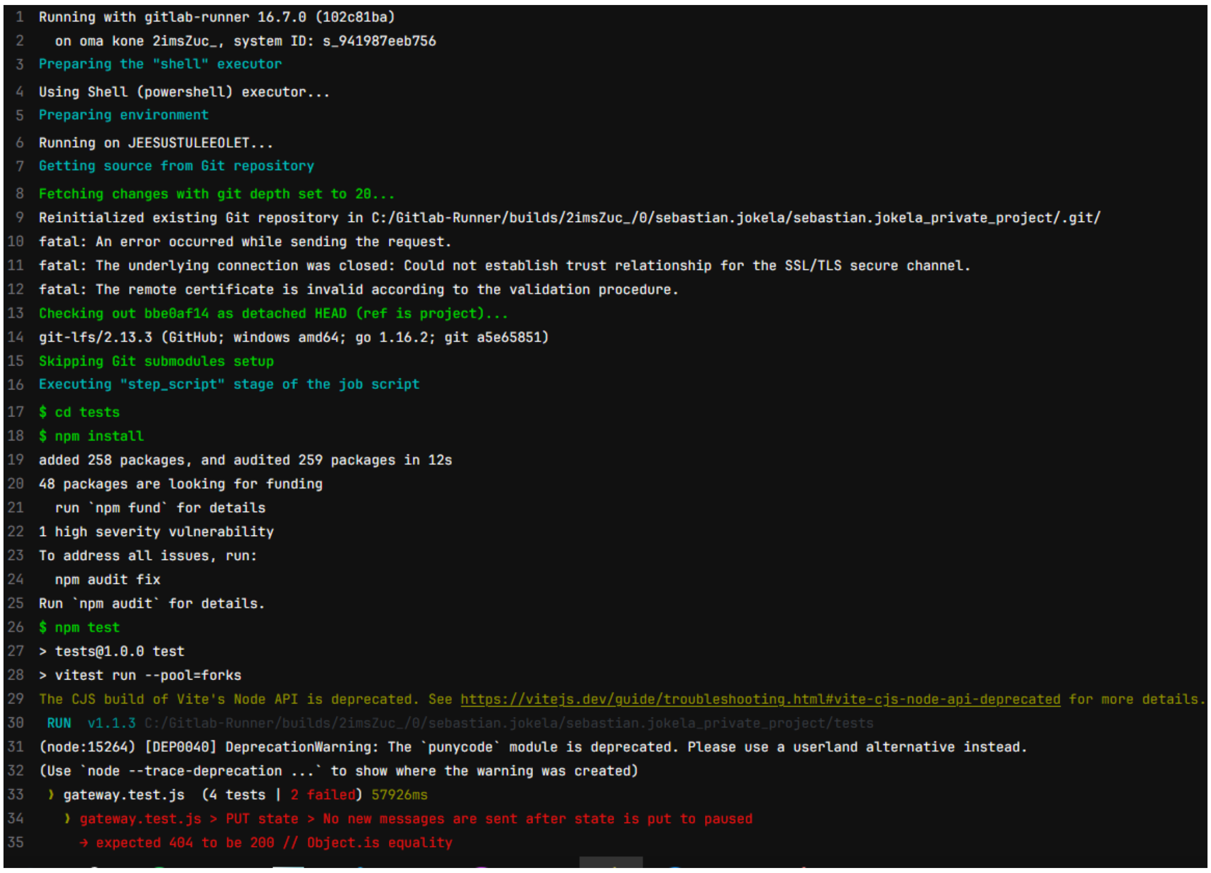
No special packing phase is in the pipeline.

The deployment phase is a simple “docker compose up --detach” command. The detach option is used to allow the pipeline run to end once the docker containers are running, so that it can be marked as passed. I would have liked to look into running the system in a cloud environment, but ran out of time.

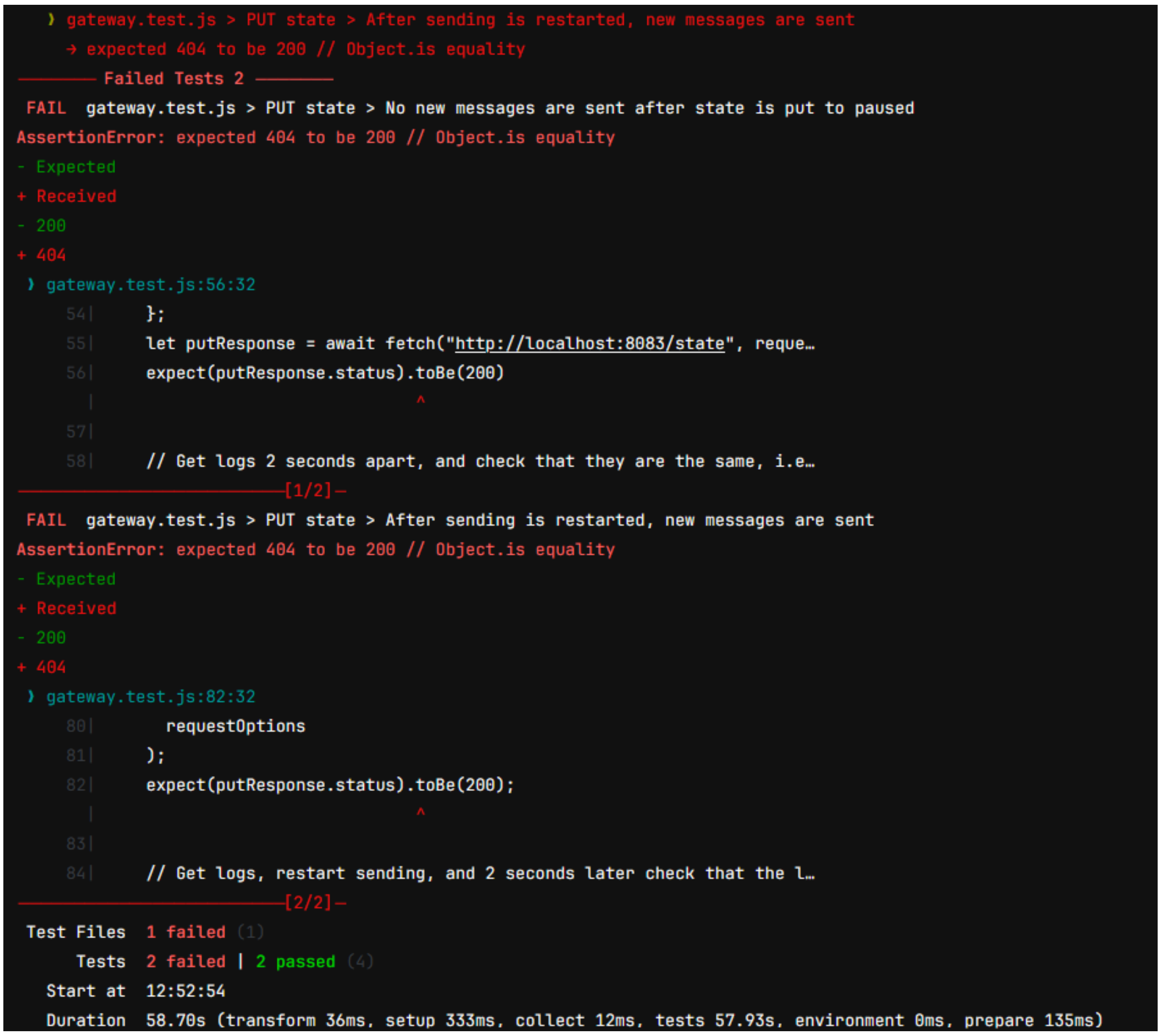
The only ways to operate or monitor the system are the features implemented in the API gateway, no additional ways exist.

3. Example runs of the pipeline

Log of failing test for PUT /state when the PUT method wasn’t implemented for the endpoint (Pipeline #2191):



```
Running with gitlab-runner 16.7.0 (102c81ba)
  on oma kone 2imsZuc_, system ID: s_941987eeb756
Preparing the "shell" executor
Using Shell (powershell) executor...
Preparing environment
Running on JEESUSTULEEOLET...
Getting source from Git repository
Fetching changes with git depth set to 20...
Reinitialized existing Git repository in C:/Gitlab-Runner/builds/2imsZuc_/0/sebastian.jokela/sebastian.jokela_private_project/.git/
fatal: An error occurred while sending the request.
fatal: The underlying connection was closed: Could not establish trust relationship for the SSL/TLS secure channel.
fatal: The remote certificate is invalid according to the validation procedure.
Checking out bbe0af14 as detached HEAD (ref is project)...
git-lfs/2.13.3 (GitHub; windows amd64; go 1.16.2; git a5e65851)
Skipping Git submodules setup
Executing "step_script" stage of the job script
$ cd tests
$ npm install
added 258 packages, and audited 259 packages in 12s
48 packages are looking for funding
  run `npm fund` for details
1 high severity vulnerability
To address all issues, run:
  npm audit fix
Run `npm audit` for details.
$ npm test
> tests@1.0.0 test
> vitest run --pool=forks
The CJS build of Vite's Node API is deprecated. See https://vitejs.dev/guide/troubleshooting.html#vite-cjs-node-api-deprecated for more details.
 RUN  v1.1.3 C:/Gitlab-Runner/builds/2imsZuc_/0/sebastian.jokela/sebastian.jokela_private_project/tests
(node:15264) [DEP0040] DeprecationWarning: The `punycode` module is deprecated. Please use a userland alternative instead.
(Use `node --trace-deprecation ...` to show where the warning was created)
 ❯ gateway.test.js  (4 tests | 2 failed) 57926ms
   ❯ gateway.test.js > PUT state > No new messages are sent after state is put to paused
     → expected 404 to be 200 // Object.is equality
```

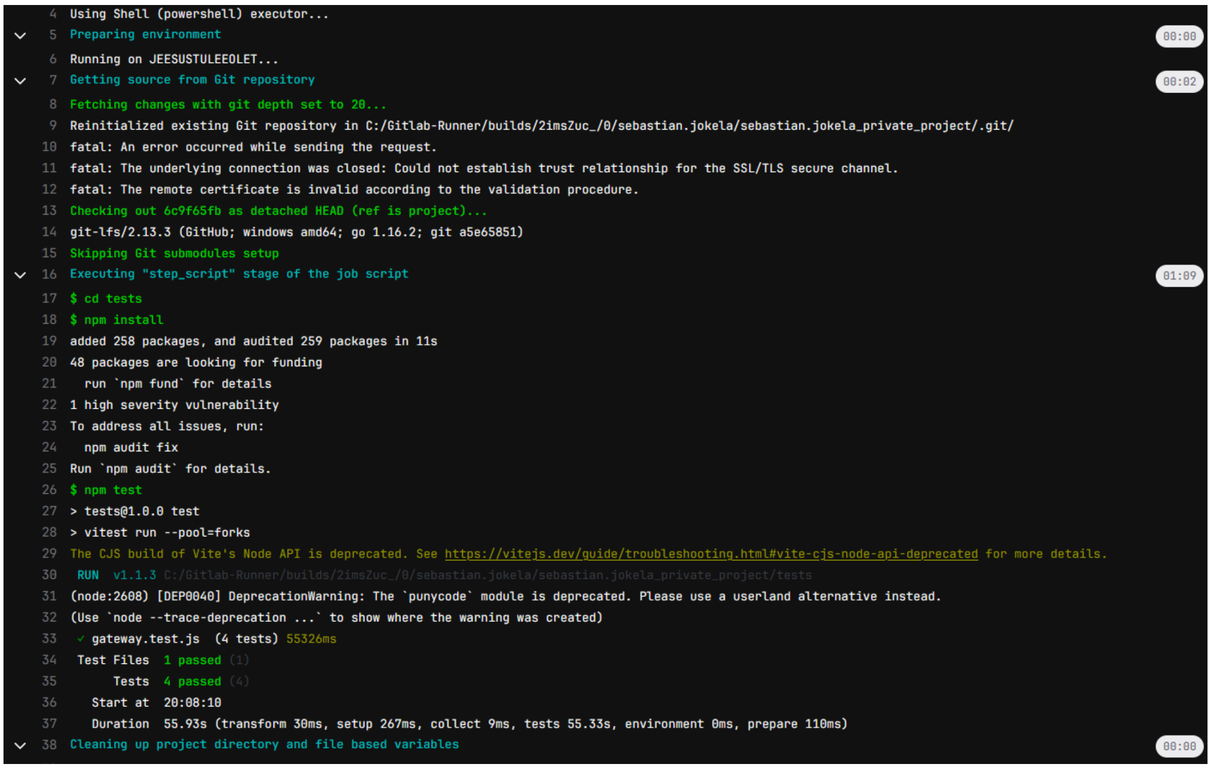


```
  ❯ gateway.test.js > PUT state > After sending is restarted, new messages are sent
    → expected 404 to be 200 // Object.is equality
──────── Failed Tests 2 ────────
 FAIL  gateway.test.js > PUT state > No new messages are sent after state is put to paused
AssertionError: expected 404 to be 200 // Object.is equality
- Expected
+ Received
- 200
+ 404
 ❯ gateway.test.js:56:32
         };
         let putResponse = await fetch("http://localhost:8083/state", reque…
         expect(putResponse.status).toBe(200)
                                   ^

         // Get logs 2 seconds apart, and check that they are the same, i.e…
──────────────────────────[1/2]─
 FAIL  gateway.test.js > PUT state > After sending is restarted, new messages are sent
AssertionError: expected 404 to be 200 // Object.is equality
- Expected
+ Received
- 200
+ 404
 ❯ gateway.test.js:82:32
           requestOptions
         );
         expect(putResponse.status).toBe(200);
                                   ^

         // Get logs, restart sending, and 2 seconds later check that the l…
──────────────────────────[2/2]─
 Test Files  1 failed (1)
      Tests  2 failed | 2 passed (4)
   Start at  12:52:54
   Duration  58.70s (transform 36ms, setup 333ms, collect 12ms, tests 57.93s, environment 0ms, prepare 135ms)
```

Log of passing test run after the method was implemented:



```
Using Shell (powershell) executor...
Preparing environment                                                    00:00
Running on JEESUSTULEEOLET...
Getting source from Git repository                                       00:02
Fetching changes with git depth set to 20...
Reinitialized existing Git repository in C:/Gitlab-Runner/builds/2imsZuc_/0/sebastian.jokela/sebastian.jokela_private_project/.git/
fatal: An error occurred while sending the request.
fatal: The underlying connection was closed: Could not establish trust relationship for the SSL/TLS secure channel.
fatal: The remote certificate is invalid according to the validation procedure.
Checking out 6c9f65fb as detached HEAD (ref is project)...
git-lfs/2.13.3 (GitHub; windows amd64; go 1.16.2; git a5e65851)
Skipping Git submodules setup
Executing "step_script" stage of the job script                          01:09
$ cd tests
$ npm install
added 258 packages, and audited 259 packages in 11s
48 packages are looking for funding
  run `npm fund` for details
1 high severity vulnerability
To address all issues, run:
  npm audit fix
Run `npm audit` for details.
$ npm test
> tests@1.0.0 test
> vitest run --pool=forks
The CJS build of Vite's Node API is deprecated. See https://vitejs.dev/guide/troubleshooting.html#vite-cjs-node-api-deprecated for more details.
 RUN  v1.1.3 C:/Gitlab-Runner/builds/2imsZuc_/0/sebastian.jokela/sebastian.jokela_private_project/tests
(node:2608) [DEP0040] DeprecationWarning: The `punycode` module is deprecated. Please use a userland alternative instead.
(Use `node --trace-deprecation ...` to show where the warning was created)
 ✓ gateway.test.js  (4 tests) 55326ms
 Test Files  1 passed (1)
      Tests  4 passed (4)
   Start at  20:08:10
   Duration  55.93s (transform 30ms, setup 267ms, collect 9ms, tests 55.33s, environment 0ms, prepare 110ms)
Cleaning up project directory and file based variables                   00:00
```

4. Reflections

Main learnings and worst difficulties

I learned a lot during this assignment. I learned how to use new testing libraries, Redis, and I learned new and relearned many things about programming in many languages. One of the big takeaways for me from this assignment was how easy setting up Testcontainers was, and how using it in my tests gave me fairly good certainty that the things I was testing actually worked, since I was testing the entire system. I was surprised how much certainty in my systems functioning I got from the tests, though given their quality and coverage, some of that most likely was false. Still it was nice to get a reason to use it and experiment with it.

Overall using test driven development was a nice experience. I was also somewhat surprised at how easy making the pipeline was given my previous experience with GitLab pipelines, though this assignment required only a very minimal pipeline (which seems somewhat weird, given the subject of the course).

Overall this assignment, along with the exercises gave me quite good confidence in working with Docker and compose, which I had some previous experience with, but not much. Though running docker compose up in the pipeline, while having tests that set up the same Docker environment caused me some headaches when I forgot to shut down the environment set up by the pipeline, and caused me to need to add a docker compose down command in to the pipeline to allow sequential runs. Though these problems would have been avoided if I had the time to set up a cloud environment.

Programming wise I was surprised how easy setting up a web server using JavaScript was. The exercises and this assignment also gave me a chance to try out C#, which was nice to experiment with, though the code I made is at points quite atrocious. Overall I wish I had the time to clean up the codebase a bit, because I am aware there are quite a few places where the could be cleaned up and split up a bit more nicely. And my code mostly just hopes there aren't any exceptions, so adding code for cases where everything doesn't go as planned would be a must if this system was actually meant to be running for a long time reliably.

I think the biggest difficulties I had were with the shutdown process, as most web servers aren't exactly meant to be shutdown based on a request, so this assignment forced me to do things with the frameworks that one isn't necessarily meant to do. This means that the shutdown process does not work optimally, for example the API gateway does not seem to respond to the shutdown command, due to shutting down before it has the time to respond. Due to limited time, I didn't have much interest in making this process work optimally, especially since it didn't seem to be possible to stop the RabbitMQ and Redis containers anyways, so optimal functioning of the feature seems to be impossible anyways.

A big thing that I could have made better is the testing. The end-to-end tests worked pretty well I feel, especially for the state management. But there definitely could have been more tests, and the tests for the GET calls are quite bad. I probably could have at least employed more timeouts there to maybe be surer of the state, and maybe looked somewhat into whether checking the timestamps in some way would have been possible. Also, while the end-to-end tests worked nicely for many parts, in some cases unit/integration tests could have been better, or at least a good addition. For example, they would have been better for testing the messages endpoint, as I could have controlled the monitor responses better, so I could have asserted more precisely what should be returned. And of course, in an actual project unit and integration tests for the other services would have been important as well. On the other hand, for only having 5 tests, I do feel my tests did cover the functionality of the system rather well, and with the run time only being around a minute, they are relatively quick as well for how much work they do setting up the environment and all.

Architecture wise, I have mixed feelings about my use of Redis. On one hand, it is much cleaner than storing the run-logs and state information in the API gateway, and having a store for them could be a good thing. And getting to experiment with how Redis works was nice. On the other hand, it might have been cleaner if service 1 could receive changes to the state via a http server or RabbitMQ, rather than reading the state every 2 seconds, which is somewhat unnecessary, when the state might not change that often. I looked into a http server or a RabbitMQ client running in a background thread as an option, but chose to use Redis instead, as I have no experience with asynchronous programming in Python, and Redis seemed an easier solution that was more interesting to experiment with. And in a way, it is nice not to have to use the API gateway to store any information, as that would violate the single responsibility principle. Also, while writing this report, I realized that the logs stored by monitor most likely should have been moved in to Redis as well, since I opted to use it.

Amount effort (hours) used

Roughly 60 hours I would guess. All in January, as I didn't have time in the 2nd period to look into this, and as I wanted to keep a small Christmas holiday. This was a very laborious assignment, and most of the effort I felt went into things that had quite little to do with DevOps. But overall it was a nice assignment, that thought me a lot and was interesting.