Optimising Test Runner Performance through Serverless Computing

Bosco

Project Report

Margaret McCarthy

ServisBOT Ltd.

20095610

Supervisor: David Power

Higher Diploma in Computer Science

South East Technological University

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

The objective of this project is to create and execute a test runner for ServisBOT that focuses on specific features of the system, identifying those that are operational and those that are not. The proposed system has the working title Bosco.

ServisBOT is a company which specialise in chatbot technology and conversational AI. They provide customer service by allowing end users to communicate with a business or service through a pop up messenger on an online platform. The technology is cloud based and mostly serverless, it is provided over the internet rather than using storage on a physical computer or server. ServisBOT has a global customer base so the system has to be consistently monitored and so tests are run continuously to ensure any problems are detected and resolved immediately.

Software testing is a critical element of software quality assurance and represents the ultimate review of specification, design and coding. Software testing is usually performed for one of two reasons: defect detection, and reliability estimation. [2]

1.1.1 Frankenstein

The components of a chatbot and each of its functions are created using microservice architecture. This is an architectural style that structures an application as a collection of services that are:

- Independently deployable
- Loosely coupled
- Organised around business capabilities
- Owned by a small team

The microservice architecture enables an organization to deliver large, complex applications rapidly, frequently, reliably and sustainably - a necessity for competing and winning in today's world. [10] There is a suite of tests which ServisBOT have named Frankenstein which run against these micro services testing every component of the chatbot. They run several times an hour, every hour.

1.1.2 TestCafe

The tests use a software package called Testcafe which is used for automated browser testing. By opening messenger in a browser in order to run the tests, it simulates what a user would do by interacting with a chatbot. If the chatbot reacts in the expected way the test passes, otherwise the issue is investigated and resolved.

1.1.3 EC2

The Frankenstein test suite is run on two EC2 instances in two different AWS regions. AWS Elastic Compute Cloud (EC2) is basically a virtual machine where developers can define the resources they need. For example, what regions the are to be run in.

1.2 The Issue

Running the tests causes a lot of contention for CPU and memory. When a test run is instigated all tests are competing for CPU usage because each EC2 instance is running thirty plus tests on one server which has limited memory.

Also there are numerous problems with Testcafe.

- ServisBOT, when running the Frankenstein test suite, find Testcafe resource heavy, expensive and inefficient, causing slow CPU performance.
- Because of the competing resources some tests affect the performance of others.
- Debugging and monitoring of the tests is complex.
- The test suite does not scale. Scalability is essential in order to increase the number of tests according to the number of user interactions and alternatively to reduce the number of tests when interactions drop which would not only prove to be more cost effective but would mostly solve all the issues mentioned.

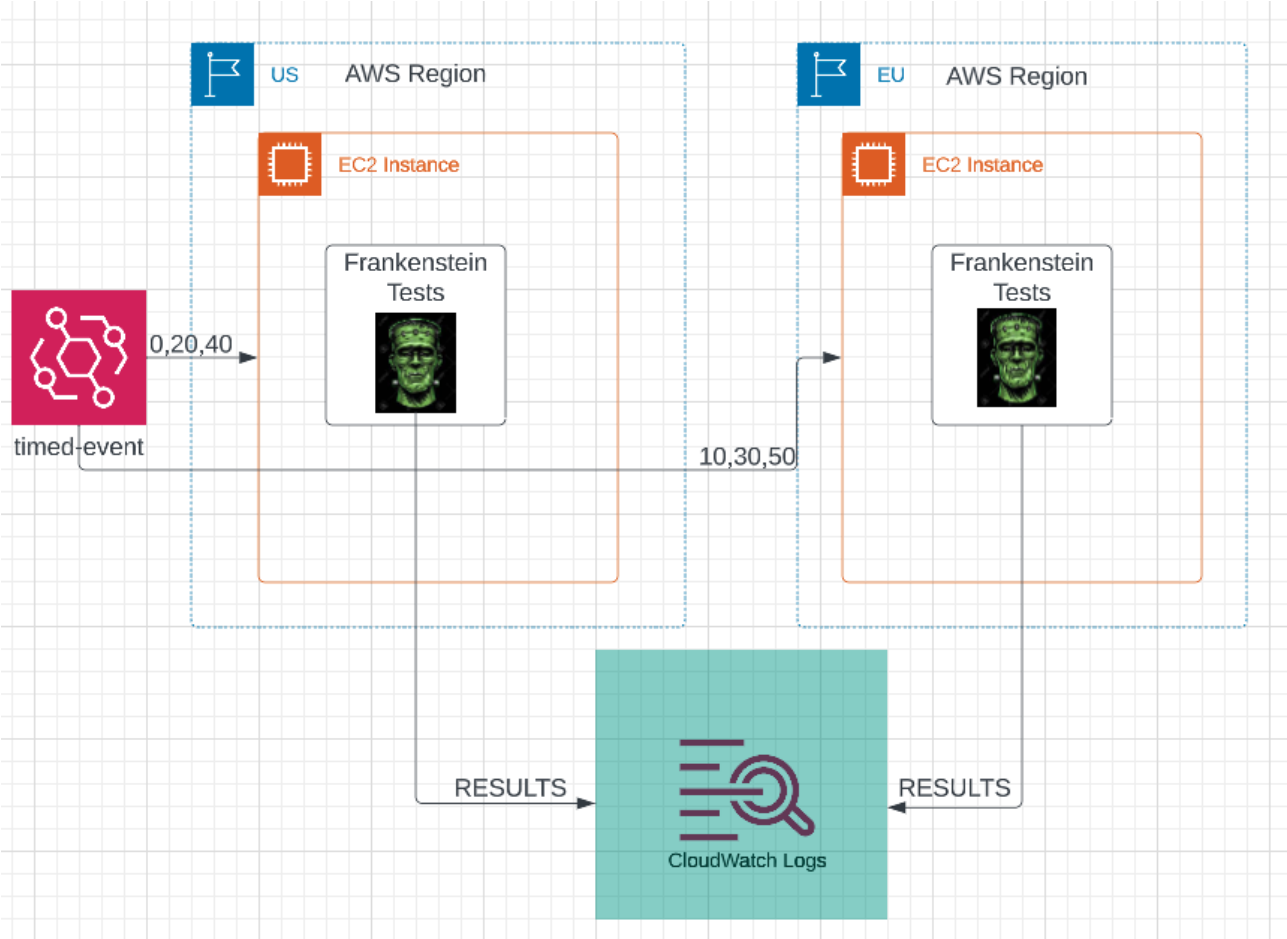


Figure 1.1 – High level view of Frankenstein

1.3 Purpose and Requirements for Bosco

The objective of this project is to create a new test runner, Bosco that is more efficient, scalable, optimised, and cost-effective than Frankenstein. The purpose is to completely revamp the existing test suite and develop AWS Lambda functions that can run each test independently without memory competition. By doing so, the test suite could be scaled infinitely, perform faster, and potentially be more cost-effective. Each test will be migrated from Frankenstein to Bosco once the infrastructure has been developed.

In particular the migration will focus on an automation tool called Puppeteer which has been proven by ServisBOT to be more performant than Testcafe. The Puppeteer package includes its own browser, Chromium, whereas Testcafe launches an external browser which is slow and cumbersome. With Puppeteer there is more control over what the developer can test, it is more efficient, easier to debug and has a lot more functionality. It is widely chosen by developers now. (4.6 million weekly downloads, 2023-03-02) [9]

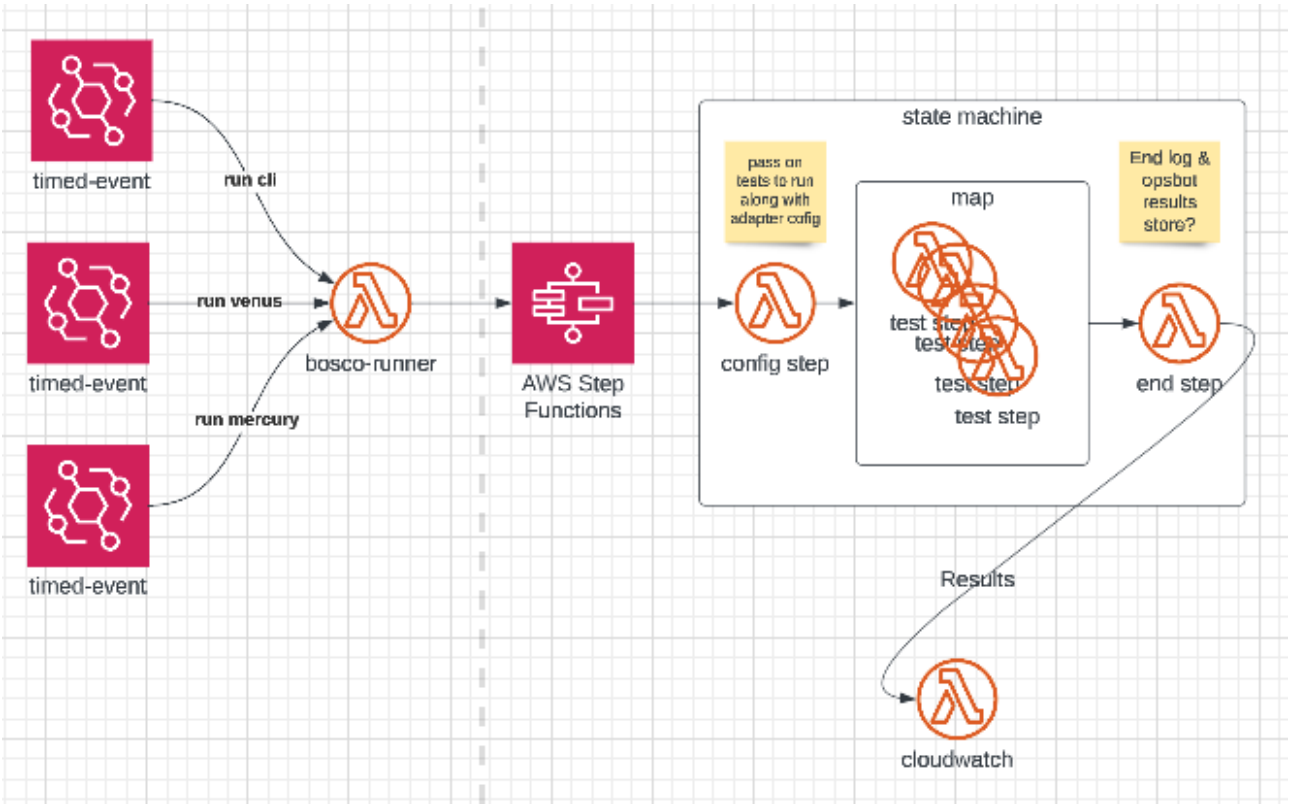


Figure 1.2 – High level view of Bosco

Chapter 2

Research and Development

In order to determine the best approach, technologies and methodology for Bosco, it was concluded that the research and analysis phase necessary for the implementation of this project was the following:

- Cost Analysis of Frankenstein
- Research into possible testing frameworks for Bosco
- Puppeteer comparison to Testcafe
- Lambda vs EC2 comparison
- Possible implementations and modelling of Bosco
- Additional requirements for implementing Bosco
- Scope of the project

2.1 Cost Analysis of Frankenstein Tests

The following cost analysis was carried out on Frankenstein. There is two EC2 instances running the tests in the EU and the US costing ServisBOT on average $20 per day for each region. This costs the company almost $15,000 a year to run tests.

2.2 Research into Possible Testing Frameworks for Bosco

Frankenstein uses Mocha as its testing framework but there are multiple frameworks that ServisBOT could choose from. The following is an investigation into what else is available as well as a review of the current framework, Mocha.

2.2.1 Mocha

According to Mocha documentation, Mocha is a feature-rich JavaScript test framework running on Node.js and in the browser, simplifying asynchronous testing. Mocha tests run serially, allowing for flexible and accurate reporting, while mapping uncaught exceptions to the correct test cases. [8]

Asynchronous testing is a technique used in software testing to verify the behavior of code that relies on asynchronous programming or event driven architecture. Asynchronous programming

Number of Tests per Region											
Code	Region	Venus	Mercury	Runs / day	Ave Time per run	Cost per year	Cost per month	Hours per month	Cost per Day	Cost per Hour	Cost per Run
1	eu-private-1		61	72	4.8 sec	$ 7,745.76	$ 645.48	744	$ 20.82	$ 0.43	$ 0.14
2	eu-private-1	20									
3	eu-1		15					744			
4	eu-1	9									
5	us-1		16	72	5.48 sec	$ 7,197.72	$ 599.81	744	$ 19.35	$ 0.40	$ 0.13
6	us-1	11									
7	usscif-1		15					744			
8	usscif-1	10									
TOTAL		50	107			$ 14,943.48	$ 1,245.29	2976	$ 40.17	$ 0.42	$ 0.28

Figure 2.1 – Frankenstein Cost Analysis

allows the program to perform tasks concurrently or in parallel, rather than waiting for each task to complete before moving on to the next one.

Mocha provides functions that execute in a specific order, logging the results in the terminal window. Mocha also cleans the state of the software being tested to ensure that test cases run independently of each other.

```
}Event run end
Mocha run finished
Tests Finished passed
info: End - Result:
info: {
        "tests": [
                {
                        "title": "Bosco Context Test",
                        "status": "passed"
                }
        ],
        "status": "passed"
}
info: Lambda successfully executed in 13576ms.
```

Figure 2.2 – Mocha Test Result Output

2.2.2 Jest

Jest is a JavaScript testing framework created by Facebook. It is open source, well-documented and popular due to its high speed of test execution. It comes with a test runner, but also with its own assertion and mocking library unlike Mocha where you need to install an assertion library, there is no need to install and integrate additional libraries to be able to mock, spy or make assertions.

2.2.3 Mocha vs Jest

Table 2.1 – Mocha Vs Jest

Mocha	Jest
Flexible configuration options	High Speed of Test Execution
Good Documentation	Good Documentation
Ideal for back-end projects	Test Runner included
Ad-hoc library choice	Built in assertion and mocking library

2.3 Review of Puppeteer

Puppeteer is a popular test automation tool maintained by Google. It automates Chrome and Firefox and is relatively simple and stable to use. Fundamentally Puppeteer is an automation tool and not a test tool. This means it is incredibly popular for use cases such as web scraping, generating PDFs, etc. [1]

Testing user flows within web applications usually involves either using an automated headful browser (i.e. FireFox with Selenium) or a headless browser, one that presents no UI, that is built on top of its own unique JavaScript engine (i.e. Testcafe). This creates a situation where trade offs have to be made such as speed vs reliability. Puppeteer aims to remove this trade off by enabling developers to leverage the Chromium browser environment to run their tests and by giving them the flexibility to leverage headless or headful browsers that run on the same underlying platform as their users. [5]

2.3.1 Pros of using Puppeteer

- Simple to set up.
- Good documentation with lots of tutorials.
- Promise based which allows for management of asynchronous code.
- Programmable web browser.
- Installs Chrome in a working version automatically.
- Puppeteer has a thin wrapper which provides a simpler interface for an underlying complex system.
- Bi-Directional (events) automating console logs is easy.
- It uses JavaScript first, which is one of the most popular programming languages.
- Puppeteer also gives you direct access to the Chrome DevTools Protocol which allows for developers to feel like there are fewer moving parts.
- Works with multiple tabs and frames. It has an intuitive API
- Trusted Actions: This criterion means dispatching events by the user which allows for user behaviours like hovers.
- End to end tests are very fast in practice but people suffer misconceptions regarding the execution speed. Typically, it's the website or web-app that are slow and the tests end up waiting for the web app to be ready most of the time.

- Pro and Con: Stability, which means how often tests fail after being authored other than when detecting a real application bug. Puppeteer waits for certain things but has to wait manually for others.
- Debugging: Can write and debug Javascript from an IDE.

2.3.2 Cons of using Puppeteer

- Limited cross-browser support. Only Chrome and Firefox.
- Feels like an automation tool and not a test framework. Often developers have to re-implement testing-related tools.
- Grids (running concurrently) in production are often a challenge
- The automatic browser set-up, downloads Chromium and not Chrome and there are subtle differences between the two.
- Smarter Locators: No support for selecting elements in multiple ways.
- Does not support parallelism, grids and infrastructure.
- Does not support self healing tests and automatically improving tests.
- Does not support Autonomous testing which is testing without code or user intervention.

2.3.3 Testcafe

TestCafe is a Node.js tool to automate end-to-end web testing. It runs on Windows, MacOs, and Linux and supports mobile, remote and cloud browsers (UI or headless). It is also free and open source.

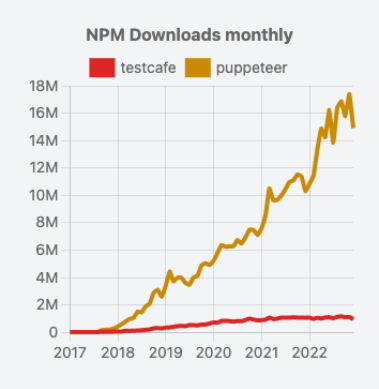
Table 2.2 – Pros and Cons of Testcafe

Pros	Cons
Cross Browser Testing	Expensive
Open Source	Slow
Easy Setup & Installation	Difficult to debug
Built-In Waits	No browser control
Supports devices without extra software package	Simulated events leads to false positives
UI End to End Testing	
Both client and server side debug	

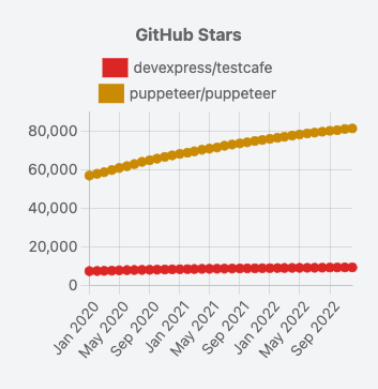
2.4 EC2 vs Lambda comparison

AWS Elastic Compute Cloud (EC2) is basically a virtual machine called an instance. Users can create an instance and define the resources neccessary for the task at hand. For example the type and number of CPU, memory, local storage and scalability. EC2 instances are intended for consistent, static operations and users pay a recurring monthly charge based on the size of the instance, the operating system and the region. The instances run until they are deliberately stopped.

h



(a) Npm Downloads



(b) GitHub stars

Figure 2.3 – Puppeteer vs Testcafe NPM statistics

Lambda on the other hand is billed per exectution and per ms in use with the amount of memory the user allocates to the function. When a lambda funciton is invoked, the code is run without the need to deploy or manage a VM. It is an event based service that is designed to deliver extremely short-term compute capability. AWS handles the back-end provisioning, loading, execution, scaling and unloading of the user's code. Lambdas only run when executed yet are always available. They scale dynamically in response to traffic.

Table 2.3 – Comparison of AWS Lambda and EC2 [4]

Feature	Lambda	EC2
Time to execute	milliseconds	seconds to minutes
Billing increments	100 milliseconds	1 second
Configuration &Function	Operating System	
Scaling Unit	Parallel function executions	Instances
Maintenance	AWS	AWS and User

2.5 Conclusion

Based on the research and analysis conducted, it can be affirmed that ServisBOT's decision to use Puppeteer in conjunction with Mocha instead of Testcafe is affirmed. Puppeteer will give the developers more control over the tests, testing will be more reliable and easier to use and debug. Together with migrating the test runner from EC2 to Lambda, the test suite will become infinitely scalable and more cost efficient. A revamp of the test runner will mean that issues encountered by Frankenstein can be resolved and unnecessary costs (for example running too many tests) can be eliminated.

Chapter 3

Initial Design & Implementation

3.1 Proof of Concept - Running Puppeteer on a Lambda

In order to prove it is possible to run Puppeteer tests on AWS Lambda, a basic end to end test was designed to run with Puppeteer. This automated a scenario whereby a browser, Chromium was launched with the url of the messenger page. After the page loaded, the messenger button was toggled and the chatbot would load. The test passed if the text from the messenger returned the expected output text.

3.1.1 Mocha

Mocha was incorporated as the testing framework. Assertions are made using the Assert library package and logged to the console but it is not in essence a testing tool. Once mocha was configured and run without errors the next step was to run the script in a lambda.

```
// basic-puppeteer.js
const puppeteer = require('puppeteer');
const url = 'https://www.google.com';
async run () => {
 const browser = await puppeteer.launch();
 const page = await browser.newPage();
 await page.goto(url);
 await page.screenshot({path: "screenshot.png"});
 browser.close();
}
run();
```

Figure 3.1 – Example of a function written with Puppeteer which takes a screenshot of the Google homepage and stores it locally

However, there are problems running Puppeteer in a lambda. Lambda has a 50 MB limit on the zip file you can upload. Due to the fact that it installs Chromium, the Puppeteer package is significantly larger than 50 MB. This limit does not apply when uploaded from an S3 bucket but there are other issues. The default setup of some Linux distributions, including the ones used by AWS Lambda do not include the necessary libraries required to allow Puppeteer to function.

3.1.2 Node Modules

A developer named Hansen[7], figured out a work-around for this in the form of the node modules, **puppeteer-core** and **chrome-aws-lambda**. Both these modules installed allow for a version of Chromium that is built to run for AWS Lambda. Incorporating these ensures the tests run successfully. Unfortunately these modules need to be in parody with each other.

The aws-chrome-lambda module has not been updated since June 2021 so its latest version is 10.1.0 whereas puppeteer-core has been updated regularly and at time of writing (26/03/23) is at version 19.8.0. When the version numbers are synced the lambda function passes. Therefore using the latest version of both modules would mean they are incompatible with each other. Whatever the reason for these modules not being updated, relying on them is impractical and will eventually lead to the tests being broken.

3.1.3 Docker

A Docker container was used instead of node modules to condense the code. A Docker container is a standalone piece of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to the next. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. Container images become containers at runtime. When you use a Docker container, it creates an isolated environment that shields the application from other parts of the computer it is running on. This means that the application can work smoothly and consistently, even if it is being used in different stages of development or on different computers with different settings.

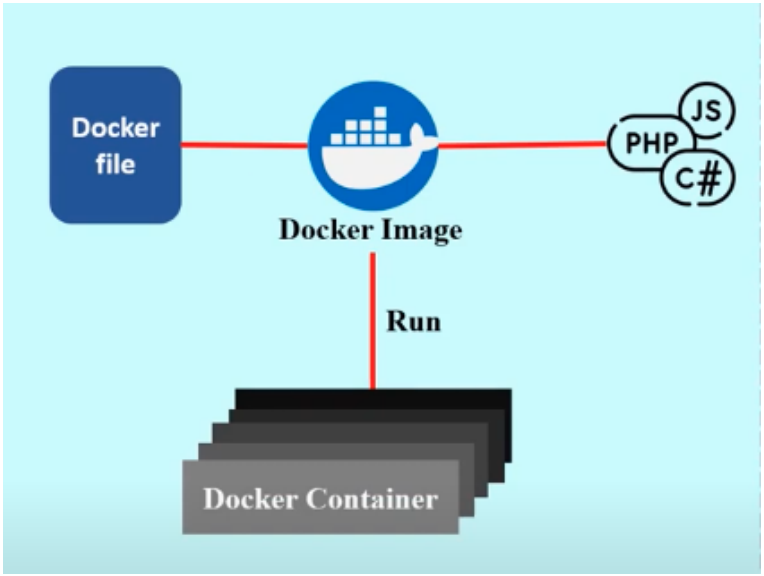


Figure 3.2 – Illustration of Docker

The puppeteer test can be run locally using a Docker engine. When the test was functioning as expected the docker image gets built, tagged and pushed to an AWS repository. The latest Docker container image is deployed to the Bosco test-runner Lambda container where it can be run and tested in a Lambda environment. This significantly speeds up the testing process, automating a browser in milliseconds compared to the previous method's average time of six seconds. The process provides more control over the testing environment and successfully runs the tests by targeting a number of tests through event handler parameters instead of hard coding. The results validate the feasibility of running tests in a Lambda using Puppeteer.

3.1.4 Conclusion

The process tested the use of Puppeteer and Mocha as an automation tool in a Lambda container and found that deploying an image through Docker is a more reliable solution. The tests can be run locally with Docker and then uploaded to the Lambda container. The results are automatically recorded in Cloudwatch logs for monitoring and assessment.

3.2 Initial Stepfunction Research and Development

The use of a Step Function in order to run the tests had been considered by ServisBOT to be a viable option for Bosco. With step functions a workflow can be created through a series of lambda functions with each step being a state within the workflow. They are based on a state machine and tasks, where a state machine is a workflow and a task is a state in that workflow that another AWS service performs.

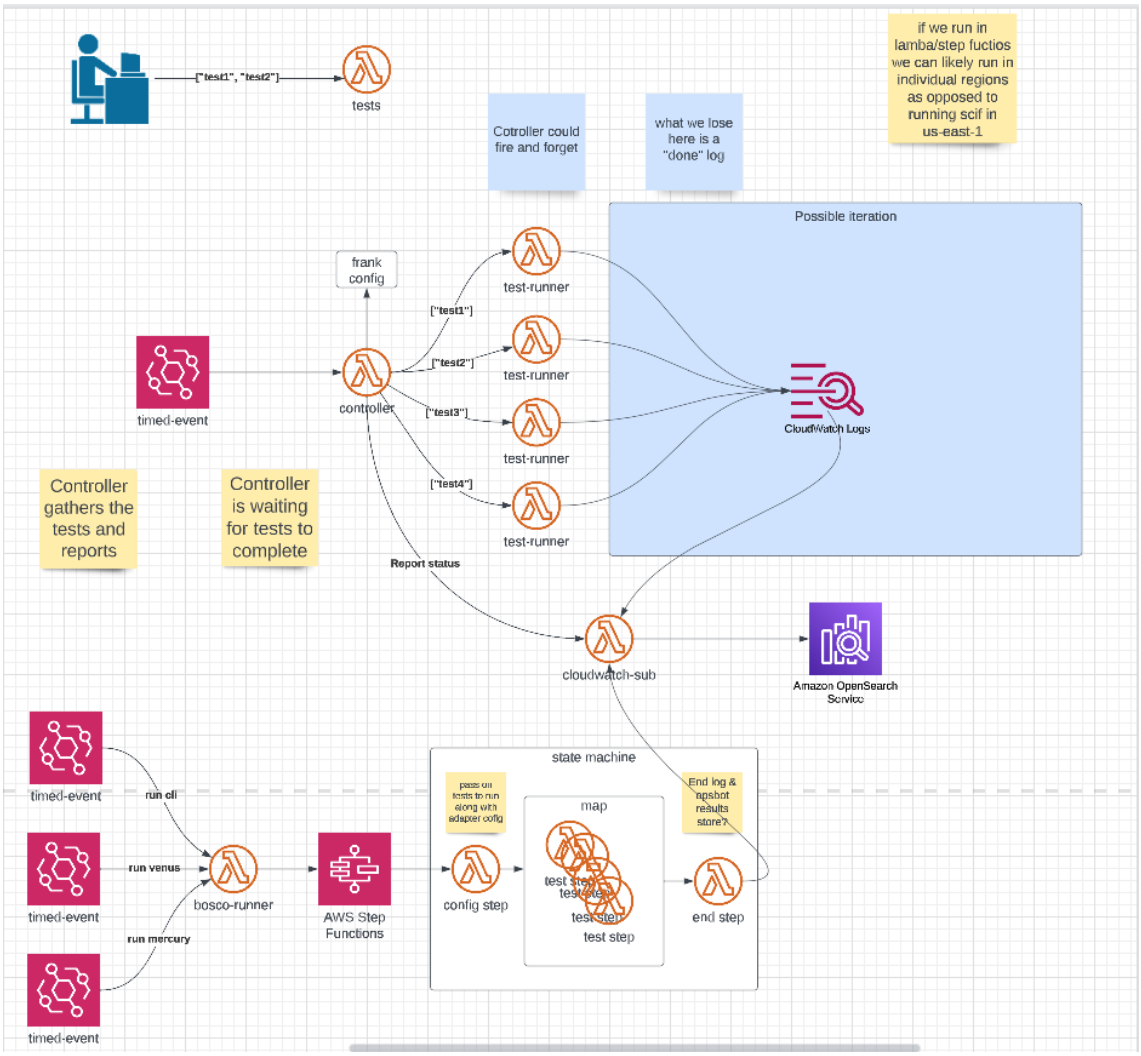


Figure 3.3 – Bosco Initial Possible Implementations

3.3 State Machine Transitions

The Bosco test suite runs through 5 state transitions (Start, StartState, Map, Done, End) plus an additional transition for each test run (Running). With three tests, there are eight states as

the **Running** state is executed three times. The first state is the **Start** state which initiates the run and then the **StartState** is invoked, creating an array of tests and passing the payload to the **Map** state.

This experiment focuses on using the Inline Map state instead of the Distributed Map state. The Inline Map state is used for fewer than forty parallel iterations which is appropriate for the number of tests that Bosco will run, while the Distributed Map state is used for larger workloads. The map state can run a lambda for each test, or it can run an array of tests. The lambdas run in parallel with each other with the results of the tests outputted to the **Done** lambda, the next state, which processes and runs the tests. Finally, a state prints out the results and the **End** state completes the workflow.

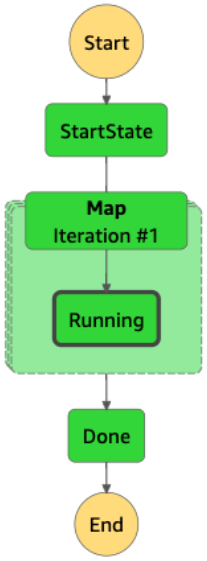


Figure 3.4 – Bosco State Machine

Name	Type	Status	Resource	Duration	Timeline	Started After
StartState	Task	Succeeded	Lambda | ...	00:00:00.198		00:00:00.38
Map	Map	Succeeded	-	00:00:21.336		00:00:00.236
#0	MapIteration	Succeeded	-	00:00:17.912		00:00:00.236
#1	MapIteration	Succeeded	-	00:00:21.336		00:00:00.236
#2	MapIteration	Succeeded	-	00:00:12.619		00:00:00.236
Done	Task	Succeeded	Lambda | ...	00:00:00.220		00:00:21.572

Figure 3.5 – Results of Execution of State Machine

3.3.1 Cold vs Warm Lambdas

Comparisons were made between running the tests on cold lambdas as opposed to warm lambdas and the results were significantly different. The cold lambda run ran in 40 seconds and the warm lambda run ran in 14 seconds total which is more than half the duration. Later in the project imiplementation, an issue occured with Lambda, where the caching of files was occuring

which is why a warm run is quicker than a cold run. This was not the intention and it caused the tests to fail. This means testing the code locally is not always compatible with testing it in a lambda environment. When the code passed through the CICD process it fell over in the Lambda environment and the code needed to be refactored to take this into consideration.

3.3.2 Initial Cost Analysis of Bosco Step Function State Machine

The cost of running the test suite in a state machine using AWS Step Functions was based on the number of state transitions.[3] This cost analysis does not account for error handling, which may increase the cost due to retries. The cost per transition is $0.000025 according to the AWS pricing calculator. The analysis is based on the current number of tests and the frequency they are run (three times an hour), which at the time of conducting the cost analysis may vary for Bosco tests.

Cost Analysis of Bosco Step Function State Machine (without error handling)

	Region	Venus	Mercury	No of Tests	Total No of StateTransitions*	Transitions/Hour	Transitions/ Month	COST / MONTH	Cost / Year
EU	eu-private-1		61	61	66	198	144540	$ 3.61	$ 43.36
	eu-private-1	20		20	25	75	54750	$ 1.37	$ 16.43
	eu-1		15	15	20	60	43800	$ 1.10	$ 13.14
	eu-1	9		9	14	42	30660	$ 0.77	$ 9.20
TOTAL EU		**29**	**76**	**105**	**125**	**375**	**273750**	**$ 6.84**	**$ 82.13**
US	us-1		16	16	21	63	45990	$ 1.15	$ 13.80
	us-1	11		11	16	48	35040	$ 0.88	$ 10.51
	usscif-1		15	15	20	60	43800	$ 1.10	$ 13.14
	usscif-1	10		10	15	45	32850	$ 0.82	$ 9.86
TOTAL US		**21**	**31**	**52**	**72**	**216**	**157680**	**$ 3.94**	**$ 47.30**
TOTAL		**50**	**107**	**157**	**197**	**591**	**431430**	**$ 10.79**	**$ 129.43**

* Based on our Bosco step funciton state machine there is 5 state transitions plus a transition for each test excluding any error handling
** Also does not include the free tier of 4000 free state transitions per month

Figure 3.6 – Bosco Step Function Cost Analysis

Chapter 4

Bosco Implementation

4.1 Lightning Page

The implementation of Bosco has begun after finalizing the proof of concept. The first step involves refactoring the Frankenstein Lightning page, which serves as the source for all selectors and functions related to the Messenger page. To avoid duplication, a constructor was created and a function was added to launch the browser and open the Messenger page.

```
async goToPage() {
 await this.page.goto(this.url,
 { waitUntil: 'networkidle2' });
}
```

Figure 4.1 – Example of a Lightning function

With this starting point the interaction node test was refactored to use the Lightning page. Two functions were created, one which clicks through the selectable list and another function which types into messenger instead of clicking the list. A BrowserFactory class was created which has a function to create the browser.

4.2 Developer Dependencies

ESLint was also added to the project which is a linting tool that ensures all the code is formatted consistently by anyone who makes changes to the code.

loglevel was added to replace the console.logs and a dependency check was added which checks for unused dependencies before commits. Once the test was running without errors the work was committed, the code reviewed and the POC branch was merged into the master branch on the Bosco repository.

lambda-local, a node module was used for testing purposes. This allowed for the running of the lamdas locally rather than having to deploy each time they were to be tested.

4.3 Context Test

The first actual Frankenstein test to be converted to Bosco was the Context test. This test sets the context from the url and tests if it is outputted with the rest of the context. This was interesting to work on as it exposed a bug in the Frankenstein test had a bug in it so it was not testing what it was supposed to be testing even though the test was passing. Once the error was highlighted and fixed by the developer team it was now possible to finish writing the test and ensure it was working correctly.

4.4 Conversation Engaged Test

The next task at hand is to convert the Conversation Engaged test. Whenever a user interacts with a bot, a goal is generated, which is a default or custom event that's tracked to measure the success of the bot's interactions. At first, it was believed that this test would be simple, but changes were requested after a pull request was submitted for review, including removing nested if statements and improving the code's formatting.

This created a need to parse the exported goal JSON result. However, an error was encountered in the Servisbot CLI proxy, where the outputted JSON was not formatted correctly and could not be parsed. To work around this, the CSV output option was used instead of JSON and the csv-parse node module was imported. To test this, a temporary script named p̈roxy.mjsẅas created to run the goals part of the test independently. It was discovered that csv-parse/sync needed to be imported instead of just csv-parse and that VSCode needed to be restarted as it was not running the code. After discussing the issue with colleagues and having them run the code on their machines, it was concluded that the problem was with VSCode and simply restarting it solved the issue.

4.5 Cloudformation

The Bosco project will be deployed using Cloudformation, a deployment tool. The Cloudformation template, which can be in either yaml or json format, specifies the necessary environment variables and resources. The aim is to upload the start, done, and test runner lambda functions through Cloudformation and have the state machine reference these lambdas. Once this is complete, the state machine will invoke the lambdas.

The cloudformation template will have three lambda functions. The start function, the test runner which points to the image of the container and the done function. The template will also contain the state machine definition. A YAML formatting extension from Redhat to format the YAML file is also necessary as it is not possible to deploy the YAML unless the format is accurate.

To log into the ServisBOT cli a ėnv file containing authentication details can be added as a temporary solution. The environment variables will be read through AWS SSM eventually.

4.6 Environment Variables

The next objective is to supply the handler with the environment variables through the orchestration lambda. The final result is to have an array called TestSuite, consisting of elements, each of which contains two arrays: one for tests and the other for environment

variables. The tests array can contain either a single test or an array of tests, but it is structured in a way that each element in the TestSuite array is also an array, which leaves room for running multiple tests in one lambda or running each test separately in different lambda functions.

Once it is possible to provide the environment variables to the handler via the orchestration lambda via an object containing the test file path and the environment variables the next step is to provide a shared environment between all lambda functions in the map. This can be achieved by providing the environment variables to the handler at the same level as the array of tests and refactoring the state machine to use the environment variables for each test. This was necessary to implement as environments can grow and therefore can be resource heavy on the state machine. By providing one instance of environment variables rather than 40 test objects containing environment variables and a test file path.

It is essential at this stage to ensure it is possible to flip between environments by either providing the JSON through the orchestrator lambda function or by providing it to the State machine initiation step. If the JSON is provided by both, then the tests should run twice.

By providing the state machine with the environment variables and testSuite there is more control over the state machine. What tests are run with what environment variables can be determined at any stage.

4.7 Refactor of State Machine to use a Shared Environment on Individual Test Instances

The state machine was then refactored to use a shared environment instead of passing the environment variables in with each test object. In the state machine definition there is an option to use an ItemSelector which allows the map to iterate over the ItemsPath but use the ItemSelector to add additional parameters.

The updated definition of the state machine included the following extra parameters:

```
 "Next": "Done",
 "ItemsPath": "\$.testSuite",
 "ItemSelector": {
  "testSuite.\$": "\$\$.Map.Item.Value",
  "environment.\$": "\$.environment",
  "profile.\$": "\$.profile",
  "testConfig.\$": "\$.testConfig"

}
```

Figure 4.2 – Definition of Map State including Test Profile

The input to the map state became the following:

In addition the input to each iteration of the running state became the following:

```
{
 "tests": [
  {
   "path": "goals/puppeteer-conversation-engaged.js"
  }
 ]
}
```

Figure 4.3 – Input to one Iteration of Map State with just the Test Path

```
{
 "testConfig": {
  "endpoint": {
   "create": {
    "AutoFailover": true
   }
  }
 },
 "environment": [
  {
   "name": "ORGANIZATION",
   "value": "some-organization"
  },
  {
   "name": "USERNAME",
   "value": "some-username"
  },
  {
   "name": "PASSWORD",
   "value": "some-password"
  },
  {
   "name": "SERVISBOT_REGION",
   "value": "eu-private-3"
  },
  {
   "name": "LOG_LEVEL",
   "value": "INFO"
  }
 ],
 "testSuite": {
  "tests": [
   {
    "path": "goals/puppeteer-conversation-engaged.js"
   }
  ]
 },
 "profile": "eu-private-3-venus"
}
```

Figure 4.4 – Running State Input including Shared Environment

4.8 Test Profiles

One significant way in which Bosco will differ to Frankenstein is in the capability to run test profiles rather than running all the tests, all the time, in all the regions. By creating a test profile it is possible to specify which environment, region and frequency of the tests which Frankenstein does not have the ability to do.

A profiles folder was created with two different profiles for the adapters Venus and Mercury in the dev environment. Each profile contained the test suite so the orchestrator needed to be updated in order to read the file paths from the JSON provided in the event profile. The node module **file system&(fs)** was used to read the contents of the profile, this was parsed to a javascript object and a spreader operator was used to combine the contents of the profile and the contents of the even inputted to the orchestrator and output these to the handler.

```
const readProfile = fs.readFileSync(testProfile, (err, data) => {
if (err) throw err;
 logger.info(data);
});
const profile = JSON.parse(readProfile);
const output = { ...profile, ...event };
```

Figure 4.5 – Parsing a JSON string to a javascript object and use of the spreader (...) operator

Next the global environment variables were removed so instead of reading them from the input to the orchestrator they would be passed from the profile to the tests. There was quite a lot of code build needed for this as Mocha is limiting in that it doesn't allow the passing of parameters to the tests.

4.8.1 Singleton Class: TestRunnerConfig

A singleton class was created to overcome this. A siingleton class is a class that can only have one instance of itself. It is used when there is going to be no change or update to the object for the duration it is used. This would be the case with our tests as we would just be using the configuration for accessing the lightning page to run the tests.

Initially the TestRunnerConfig took in the runnerEvent and created an instance of itself. It had getters to get different variables from the event which at the time were just the profile, testConfig, testSuite and environment. The tests ran but now the aim was to remove process.env to set the variables and use an Environment class that instead reads the environment variables from the TestRunnerConfig.

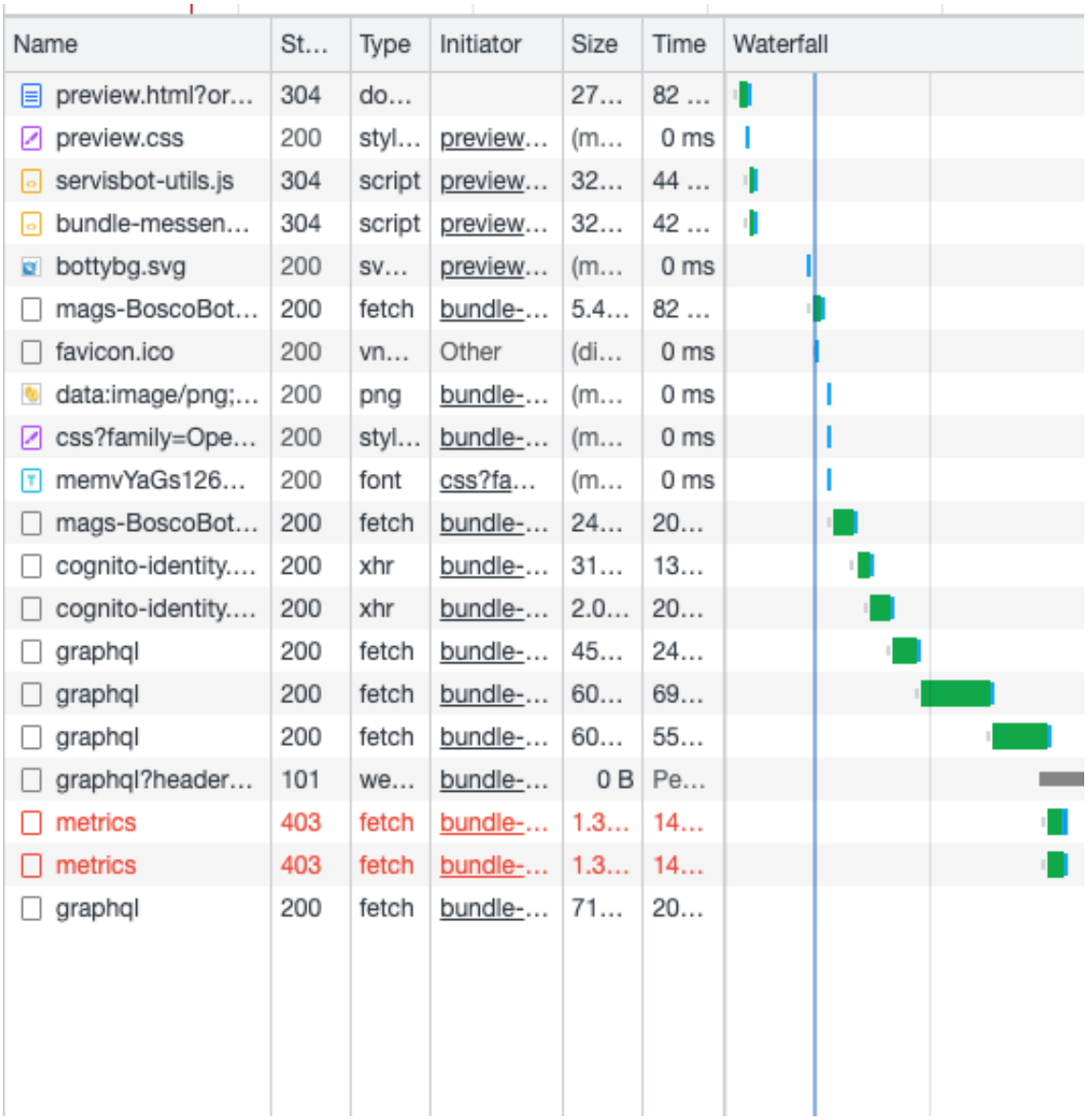
An Environment object was created using the environment from the event. This could now be accessed in the tests by calling the getters from the Environment object.

```
const testRunnerConfig = TestRunnerConfig.getInstance();
const environment = testRunnerConfig.getEnvironment();
const params = {
organization: environment.getOrganization(),
username: environment.getUsername(),
password: environment.getPassword(),
region: environment.getSBRegion()
};
```

Figure 4.6 – Replacing proccess.env with getters to an Environment class invoked by a singleton class, TestRunnerConfig

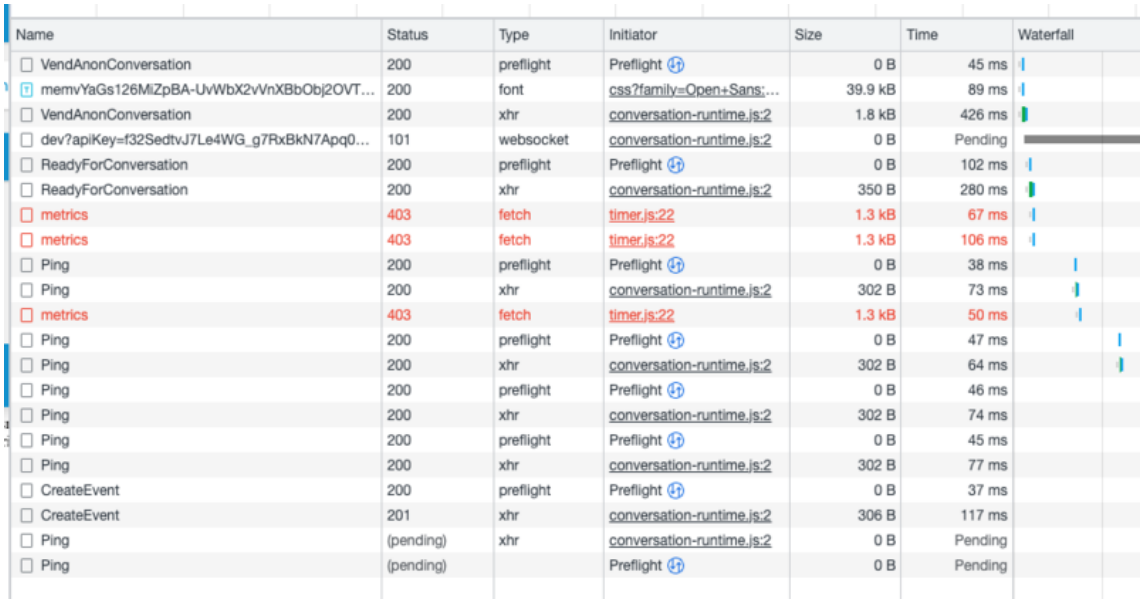
4.9 Endpoint Proxy

Bosco needed to support both Mercury and Venus engagement adapters similar to Frankenstein. This is achieved through the endpoints whereby each endpoint is suffixed by the test profile name. When the endpoint is created instead of creating it uniquely with the timestamp, middleware in the form of an Endpoint proxy would instead suffix the endpoint with the profile and thereby point the tests to the url containing the endpoint proxy. To prove the tests were making network calls to either Mercury or Venus, the tests were slowed down, run in headful mode and the network tab examined. When the browser was talking to Venus there calls to **VendAnonConversation** and **ReadyForConversation** and when there were calls to Mercury, **graphql** calls were evident.



Name	St...	Type	Initiator	Size	Time	Waterfall
preview.html?or...	304	do...		27...	82 ...	
preview.css	200	styl...	preview...	(m...	0 ms	
servisbot-utils.js	304	script	preview...	32...	44 ...	
bundle-messen...	304	script	preview...	32...	42 ...	
bottybg.svg	200	sv...	preview...	(m...	0 ms	
mags-BoscoBot...	200	fetch	bundle-...	5.4...	82 ...	
favicon.ico	200	vn...	Other	(di...	0 ms	
data:image/png;...	200	png	bundle-...	(m...	0 ms	
css?family=Ope...	200	styl...	bundle-...	(m...	0 ms	
memvYaGs126...	200	font	css?fa...	(m...	0 ms	
mags-BoscoBot...	200	fetch	bundle-...	24...	20...	
cognito-identity....	200	xhr	bundle-...	31...	13...	
cognito-identity....	200	xhr	bundle-...	2.0...	20...	
graphql	200	fetch	bundle-...	45...	24...	
graphql	200	fetch	bundle-...	60...	69...	
graphql	200	fetch	bundle-...	60...	55...	
graphql?header...	101	we...	bundle-...	0 B	Pe...	
metrics	403	fetch	bundle-...	1.3...	14...	
metrics	403	fetch	bundle-...	1.3...	14...	
graphql	200	fetch	bundle-...	71...	20...	

Figure 4.7 – Mercury Network Calls



Name	Status	Type	Initiator	Size	Time	Waterfall
VendAnonConversation	200	preflight	Preflight	0 B	45 ms	
memvYaGs126MiZpBA-UvWbX2vVnXBbObj2OVT...	200	font	css?family=Open+Sans:...	39.9 kB	89 ms	
VendAnonConversation	200	xhr	conversation-runtime.js:2	1.8 kB	426 ms	
dev?apiKey=f32SedtvJ7Le4WG_g7RxBkN7Apq0...	101	websocket	conversation-runtime.js:2	0 B	Pending	
ReadyForConversation	200	preflight	Preflight	0 B	102 ms	
ReadyForConversation	200	xhr	conversation-runtime.js:2	350 B	280 ms	
metrics	403	fetch	timer.js:22	1.3 kB	67 ms	
metrics	403	fetch	timer.js:22	1.3 kB	106 ms	
Ping	200	preflight	Preflight	0 B	38 ms	
Ping	200	xhr	conversation-runtime.js:2	302 B	73 ms	
metrics	403	fetch	timer.js:22	1.3 kB	50 ms	
Ping	200	preflight	Preflight	0 B	47 ms	
Ping	200	xhr	conversation-runtime.js:2	302 B	64 ms	
Ping	200	preflight	Preflight	0 B	46 ms	
Ping	200	xhr	conversation-runtime.js:2	302 B	74 ms	
Ping	200	preflight	Preflight	0 B	45 ms	
Ping	200	xhr	conversation-runtime.js:2	302 B	77 ms	
CreateEvent	200	preflight	Preflight	0 B	37 ms	
CreateEvent	201	xhr	conversation-runtime.js:2	306 B	117 ms	
Ping	(pending)	xhr	conversation-runtime.js:2	0 B	Pending	
Ping	(pending)		Preflight	0 B	Pending	

Figure 4.8 – Venus Network Calls

4.10 Creating a Test with a Secret

Many of the Frankenstein tests use secrets in order to run the tests. Secrets can be defined as secure documents containing access keys, api keys, api secrets or IDs to external systems. They can be created and stored in the ServisBOT system and then referenced by bots securely. This is how ServisBOT manage API keys for AWS.

The test that was chosen was the NLP worker, Lex V2, intent-publish test. A **Lex** worker is a remotely managed NLP worker who takes input and sends it to a Lex bot for Natural Language Processing. A Lex bot is an Amazon bot, users use ServisBOT to access, manage and run their Lexbots. **NLP** is a form of Artificial Intelligence that transforms text that a user submits into language that the computer programme can understand. In order for ServisBOT to classify utterances using a remote managed NLP, the correct API keys, access tokens or cross account roles need to be configured in the ServisBOT Secrets Vault.

An AWS cross account role, ie a secret, is required in order to access Lex. An environment variable was created that the test could access called LEX_V2_CROSS_ACCOUNT_ROLE_SECRET and to this was assigned the secret definition for dev in JSON format.

This test creates a long living bot which is a bot that is not deleted after the test but is created the first time the test is run and then reused for every test. The bot's variable intent is updated with the timestamp every time to ensure the bot is publishing successfully. When the bot is publishing its status is polled every 5 seconds and when published its status is SUCCEEDED and so the test starts.

A bot class was created so that a publish function which polled the status of the publishing and outputted the status to the console could be created. This was so the test behaved the same way as Frankenstein but also because using the ServisBOT cli proxy does not have this functionality so in calling sbCli.publish we would not have the polling outputted to the console.

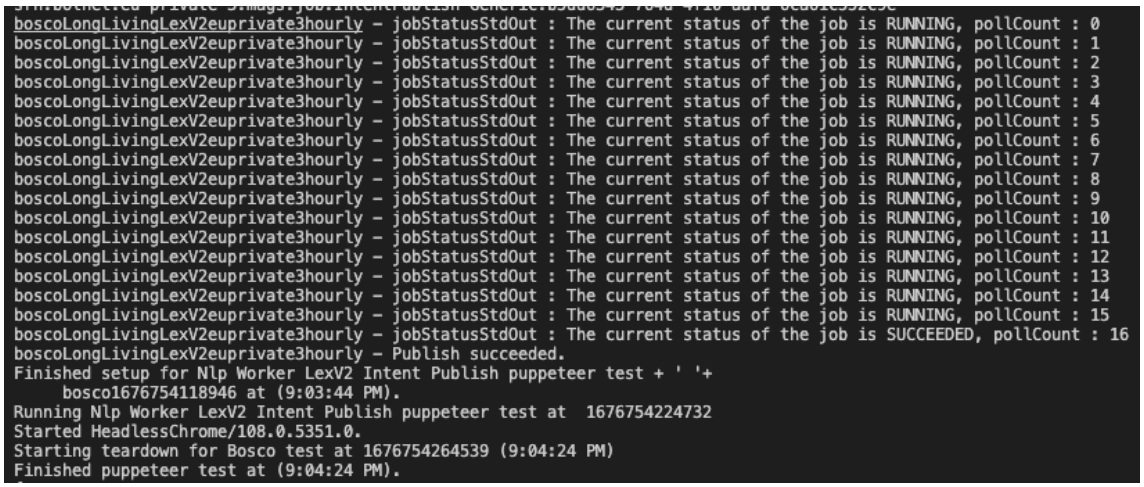


Figure 4.9 – NLP worker test with polling

4.11 SSM Parameters

The next task for Bosco was to build the environment in the orchestrator using SSM parameters. SSM (Systems Manager) is a service provided by AWS that allows you to securely store and retrieve data for your application [6]. They can be encrypted and the service is easy and free to use. A few lines of code can retrieve the parameters from the store.

The steps to this task were as follows:

- The parameters were created in the SSM parameter store in the same format as the Frankenstein parameters.
- Code was added to the orchestrator to read the SSM parameters using the node module **aws-sdk**.
- A new policy was added to the cloudformation template in order to give IAM permissions to read from the parameter store
- The environment was built in the orchestrator by retrieving the ssm parameters in the orchestrator, building the envirnoment object and returning this to the output.
- A new class, Secrets was added in order to retrieve the SSM parameters from the parameter store to extract that work from the orchestrator. Later to be renamed EnvironmentResolver. This was done by passing the ssm through to the Secrets constructor.
- The README was updated to reflect the changes.

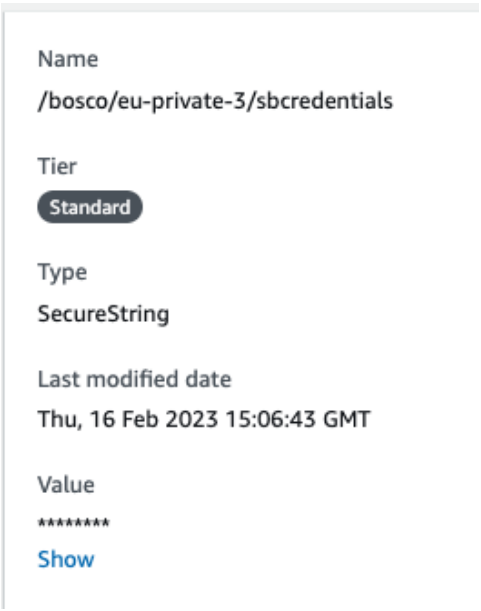


Figure 4.10 – SSM Parameters created in AWS Systems Manager

4.12 Eventbridge Rules (CRON)

Like Frankenstein, Bosco needs to run on a scheduled event. WS Eventbridge allows developers to trigger evAents on a CRON which is 'a command to an operating system or server for a job that is to be executed at a specified time'. To trigger the state macine to run different profiles at different times, in different regions the cloudformation file needed to be updates.

- First the rule defaults were added to the cloudformation
- The CRON rules were then added

```
  ScheduleMercuryExpression: { Type: String, Default: "cron(0,20,40 * * * ? *)"
↪  } # run at 0,20,40 past the hour
  ScheduleVenusExpression: { Type: String, Default: "cron(10,30,50 * * * ? *)"
↪  } # run at 10,30,50 past the hour
  ScheduleHourlyExpression: { Type: String, Default: "cron(0 * * * ? *)" } #
↪  run at 0 past the hour
```

- The triggers were added.

```
  EuMercuryTriggerEnabled: !Equals [ !Ref MercuryEuRunnerEnabled, "true" ]
  EuVenusTriggerEnabled: !Equals [ !Ref VenusEuRunnerEnabled, "true" ]
  EuHourlyTriggerEnabled: !Equals [ !Ref HourlyEuRunnerEnabled, "true" ]
```

- The schedule rules were added
- A new Sceduled Event IAM role to execute the state machine was added with a special policy that allows the execution of the state machine.

```
ScheduledEventIAMRole:
    Policies:
      - PolicyName: StateMachineExecutionPolicy
        PolicyDocument:
          Version: "2012-10-17"
          Statement:
            - Effect: "Allow"
              Action: "states:StartExecution"
              Resource:
                - !Ref ImportBoscoStateMachine
```

Once the cloudformation file was deployed successfully, it was possible to observe the timed events executing successfully. You can see from the following table that each execution is now given a unique ID and runs at regular intervals past the hour. The working CRON instigated time machine was showcased to the team and they were queried on whether they would liked the introdcution of a profile that ran hourly as opposed to every twenty minutes. The team agreed that the tests were already running too frequently and costing a lot. Bosco will have more control over what tests are run, in what regions and how frequently through the test profiles, therefore giving ServisBOT much more control over their testing suite and providing flexibility and control that Frankenstein seriously lacks in.

256c7f8c-8dd0-f4eb-300f-f2c673a98d1b_b0a4c8c7-3fa5-f450-21c0-b04cb9f5562a	Succeeded	Feb 20, 2023 10:20:20.192 AM	Feb 20, 2023 10:20:54.247 AM
ddb30f4c-71a5-4fbc-14b5-21495d29e4b5_cf5eddd6-7b06-4e9b-881d-497f1aaf1565	Succeeded	Feb 20, 2023 10:15:33.297 AM	Feb 20, 2023 10:17:35.007 AM
99ee6095-c74f-4087-5cf3-50f5ccccffa6_4cadea85-c221-5dde-891e-d2c784dce24e	Succeeded	Feb 20, 2023 10:10:10.408 AM	Feb 20, 2023 10:10:48.721 AM
fa6aab61-2abe-8873-2330-1e704b80cc10_6a47ad6d-ab4d-3fcb-8887-a8274809039f	Succeeded	Feb 20, 2023 10:00:20.308 AM	Feb 20, 2023 10:00:58.032 AM
120c878d-5297-5c06-94b4-1a8e94e9e168_d084cfbf-1d09-b26a-eefb-f2fb98b3a75c	Succeeded	Feb 20, 2023 09:50:10.268 AM	Feb 20, 2023 09:50:50.959 AM
6449e094-5499-9982-1692-7226dd0f0d30_2b6cff19-094d-6d71-51d1-a524453204ed	Succeeded	Feb 20, 2023 09:40:20.343 AM	Feb 20, 2023 09:41:24.648 AM
8497585c-c160-f8a0-616f-84ee86adfde2_0c5204ff-64c3-d462-3199-686fbfdb8549	Succeeded	Feb 17, 2023 03:20:27.477 PM	Feb 17, 2023 03:21:00.094 PM
616643b6-d190-aeaf-4559-73d62664ccbd_5c783440-fed9-5798-ae34-554c2d5b125f	Succeeded	Feb 17, 2023 03:15:32.213 PM	Feb 17, 2023 03:17:54.455 PM
cfda32f0-2e56-eea6-28b6-42ecf176c67c_d31ca8d1-447f-449d-a189-4237c660cc30	Succeeded	Feb 17, 2023 03:10:24.509 PM	Feb 17, 2023 03:11:01.972 PM
e94a653a-9d03-12cd-1004-82ca8e12baa3_045378e0-79a0-8f64-ebed-672a59746f32	Succeeded	Feb 17, 2023 03:00:27.399 PM	Feb 17, 2023 03:01:23.119 PM
3ee3966e-00c5-8c17-681c-7c48c90d6778_4133d88a-fc7e-9c25-9ff2-45116471cdf9	Succeeded	Feb 17, 2023 02:50:24.253 PM	Feb 17, 2023 02:51:03.456 PM

Figure 4.11 – State Machine Executions on a CRON

4.13 Converting a CLI Test

Frankenstein also has a number of tests that test the functionality of the cli. The cli has most of the functionality if not more of the ServisBOT UI. The tests do not use a web scraper like Testcafe as it is not needed, instead the tests just run the cli commands and use an assertion to ensure they were carried out sufficiently. As a result the tests are completed much faster than the Testcafe tests.

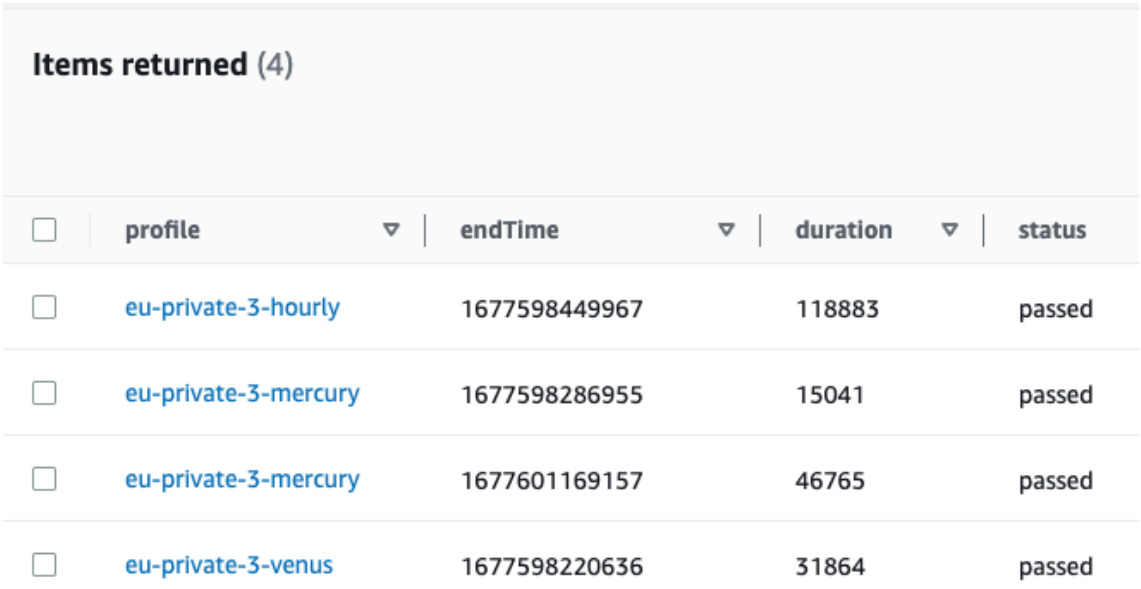
The CLI test chosen was the Content, create-describe-update-delete-list test. It was a simple test that tests a Content node. It takes a content definition in JSON format and creates a content node object. Then it describes it, updates it, lists the contents and deletes the content. When deployed the test took 18 seconds as opposed to a Puppeteer test taking an average of 55 seconds.

When put to the team however, whether Bosco was to include CLI tests or not, it was agreed that it was unnecessary to include the CLI tests. Bosco was to test portal specifically where the users would be using the UI first hand. CLI tests were basically only for developers and there was less of a need to test them. The goal is to have full Frankenstein coverage with Bosco and

testing the ServisBOT CLI proxy has 95% coverage. It was then decided to remove the CLI test from Bosco.

4.14 Dynamo DB

The next task for Bosco was to store the test results in a global Dynamo DB table. Each test run would have a row in the table with details on the results of the test. Initially the following attributes were chosen. The primary key was set to the test profile with the sort key which is unique set to the endTime of the test run. The other values passed to the dynamo table would be the duration of the test run and the status of the test run. Once the step function would write to the dynamo table, additional attributes would be added.



Items returned (4)

profile	endTime	duration	status
eu-private-3-hourly	1677598449967	118883	passed
eu-private-3-mercury	1677598286955	15041	passed
eu-private-3-mercury	1677601169157	46765	passed
eu-private-3-venus	1677598220636	31864	passed

Figure 4.12 – Dynamo DB Global Table Results

The learning path to achieving this began with:

- Manually creating a dynamo table in dev and deciding the attributes that would be written to it.
- The handler was then refactored in order to pass the profile, the endTime and the duration to the results lambda.
- The results lambda then used the aws-sdk to write to the Dynamo DB. It filtered through the status of all the tests and if there was a failed test then the overall test run was marked as a fail.
- Once the step function was writing to the table which was created manually, a new resource, a Global Dynamo table was added to the cloudformation template.
- A condition was added to the Dynamo table so that it was possible to interchange between deploying the table or not.

Chapter 5

Deployment

The deployment process was undertaken with help from the team as the initial plan needed to be altered. The triggers were removed from the cloudformation script and instead were put in a script specific to the AWS Region they would be deployed in.

5.1 CICD Process

The steps to the CICD process for Bosco are as follows:

- Developer puts in a PR (pull request) with the changes they made to the project code.
- Senior developer reviews the code and when ok, approves the merge.
- Developer merges branch which kicks off the CICD process.
- Unit tests are run, artefacts are built and the project code is deployed on lower/staging/testing.
- If successful Slack will report on deployment, otherwise error needs to be located and debugged.
- Developer tests the code changes on lower/staging/testing.
- When the changes are verified developer tags a release through deploybot on Slack.
- This initiates the CICD process again which builds artifacts, runs unit tests, integration tests and deploys to lower/staging/testing again as a tagged release.
- Developer tests on staging/lower/testing again.
- Once verified developer requests a deploy to production/testing from deploybot on Slack.
- CICD process is initiated once more with the project being deployed on production/upper.
- Developer tests the code on upper and verifies.

5.2 Handling Multiple ServisBOT Regions

It was required that Bosco would run multiple ServisBOT region test profiles per deploy. For example deploying an instance of Bosco in the AWS region, eu-west-1 should be capable of creating CRON schedules for ServisBOT regions, eu-1 and eu-private-1.

5.2.1 Triggers

Triggers are what initiates the test runs. They can be defined for each region and each environment. The decision was taken to reduce the number of test runs that Frankenstein runs. Frankenstein runs 24/7 and this was deemed unnecessary. This will have a huge cost reduction effect on the company. The following times were decided on for the test runs.

Table 5.1 – Scheduled Triggers (CRON)s for Running Bosco

	Staging/Lower	**Prod/Upper**	**Prod/Upper**	**Prod/Upper**
ServisBOT Region	**eu-private-1**	**eu-1**	**us1**	**usscif1**
Timeframe	Mon-Fri, 8am-5pm	24/7	24/7	24/7
Mercury	0,20,40	0,20,40	0,20,40	N/A
Venus	10,30,50	10,30,50	10,30,50	10,30,50
Hourly	hourly	hourly	hourly	hourly

A new folder src/triggers was created which contatins cloudformation templates for each AWS region and ServisBOT region pair. For example: src/triggers/eu-west-1/eu-1-triggers.yaml src/triggers/eu-west-1/eu-private-1-triggers.yaml

Within each of these template files is the infrastructure for the eu-1 CRON and eu-private-1 respectively. A new lambda was created in Bosco which does the following: * When invoked, checks the AWS region the lambda is running in. * Reads the src/triggers/¡aws-region¿ folder, and uses the AWS SDK to deploy each of the templates found within this folder. * The lambda function has the ability to poll to check for new or updated cloudformation stacks. * The lambda will fail if any of the stacks fail to create/update. A post-build code build script invokes the lambda. This now only runs on deployment to an account (not the CICD account).

5.3 Additional Features Post-Deployment

5.3.1 Logging Test Results

A new class was added, LogReporter in order to report the test results to Cloudwatch. Each test reports on itself plus each test run will log a completion report. These reports can be viewed in Cloudwatch in the following format.


```
    {
      "SbRegion": "eu-private-3",
      "TestProfile": "eu-private-3-venus",
      "TestTitle": "Bosco Interaction Test",
      "DurationInMs": 42268,
      "Result": "passed",
      "Type": "test",
      "TestRunnerIdentifier": "1b3bc8b5-7db5-48ae-83a4-3ad40a6bfa92"
}
```


Figure 5.1 – Cloudwatch Log of Bosco Interacton Test

```
    {
      "SbRegion": "eu-private-3",
      "TestProfile": "eu-private-3-venus",
      "TotalDurationInMs": 120218,
      "Result": "passed",
      "Type": "completion",
      "TestRunnerIdentifier": "1b3bc8b5-7db5-48ae-83a4-3ad40a6bfa92"
  }
```


Figure 5.2 – Cloudwatch Log of Complete Test Run

5.3.2 Cloudwatch Insights

A query was constructed and added to the READMe to make it easier for developers to be able to view failed test results. The following figure is the result of the query.

#	@timestamp	SbRegion	TestTitle	DurationInMs	TestProfile	Type	TestRunnerIdentifier	Result
▼ 1	2023-03-15T06:02:21.…	usscif1	Bosco NLP…	41237	usccif1-hou…	test	4c1de8b7-d76e-f545-ffa9-c…	failed

Field	Value
@ingestionTime	1678860150895
@log	178866867700:/aws/lambda/prod-bosco-test-runner
@logStream	2023/03/15/[$LATEST]ed38f21934ec42c7be27b51a0671b273
@message	2023-03-15T06:02:21.927Z 722653d5-5612-47e3-9e4c-6abbb6a34d8f INFO {"SbRegion":"usscif1","TestProfi
@requestId	722653d5-5612-47e3-9e4c-6abbb6a34d8f
@timestamp	1678860141927
DurationInMs	41237
Result	failed
SbRegion	usscif1
TestProfile	usccif1-hourly
TestRunnerIdentifier	4c1de8b7-d76e-f545-ffa9-c47c723e534a_9456bda5-69fc-a9ea-b456-c5799d0cf6c6
TestTitle	Bosco NLP Worker LexV2 Intent Publish Test
Type	test

Figure 5.3 – Cloudwatch Log Insight of Failed Tests for 1 week on Production

5.3.3 Failed Test Screenshots stored in S3

On failure of a test, a screenshot is taken using Puppeteer and stored locally in a /tmp/screenshots folder with the execution ID of the state machine run. On completion of the test run, the handler checks for the existance of a directory with the execution ID and if it exists then it runs through each file in the directory which will be a screenshot of the moment the test failed and it uploads these screenshots to an S3 bucket. The /tmp directroy is the only directory that lambda can access. Any attempts to read/write to other named directories will cause an error. Once the files have been uploaded the /tmp/screenshots directory is deleted.

The S3 bucket is deployed using cloudformation. There is a condition on the bucket to deploy only in a staging environment and otherwise only if specified. Cloudformation permissions were added to allow for upload to the specific bucket.

The process to achieving this feature began initially with each test storing the screenshot on a failure. A new function was written in the Lightning Page class

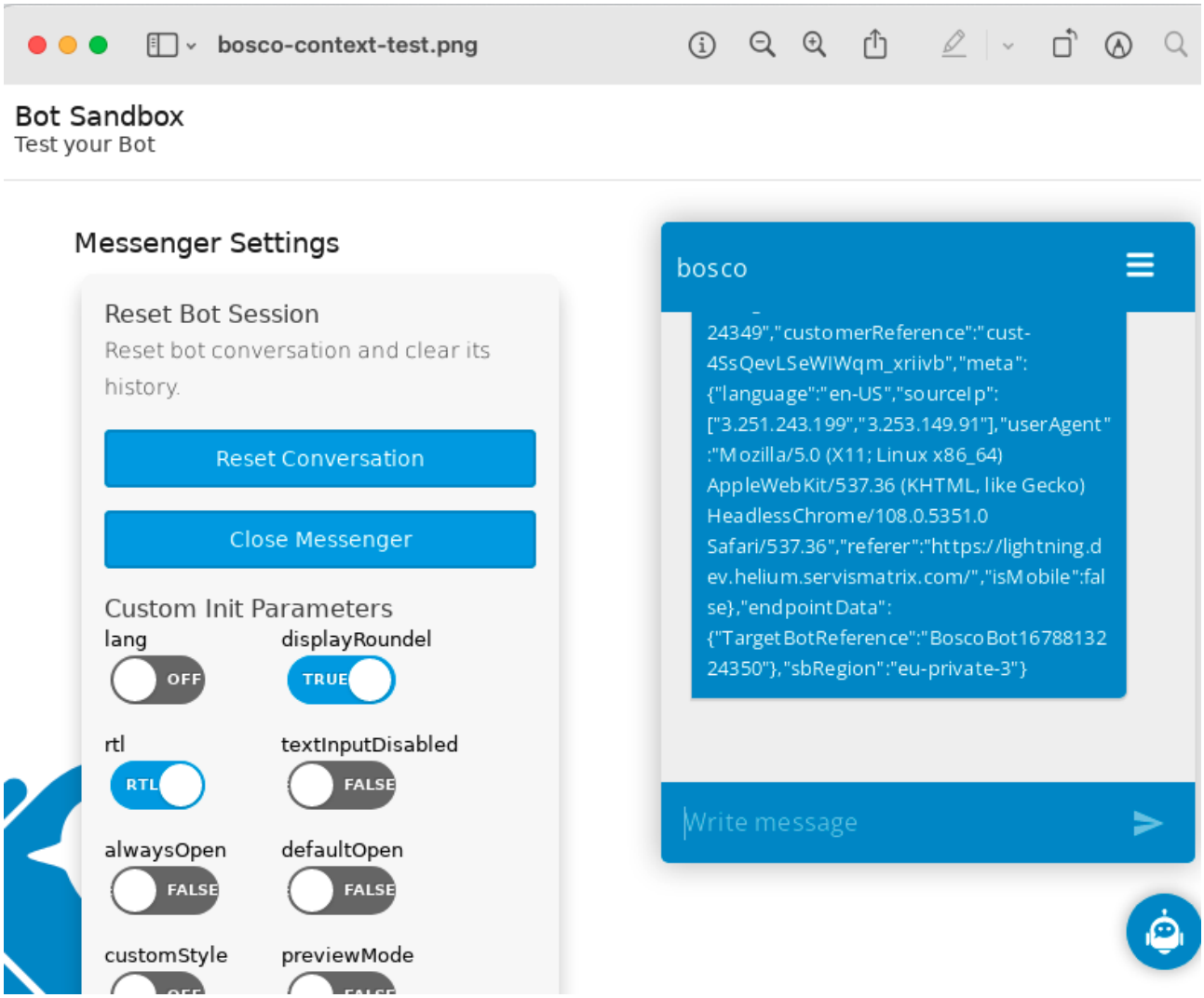


Figure 5.4 – Screenshot of failed test

Chapter 6

Conclusion

6.1 Reflection

6.2 Key skills

- Sprint planning.
- Amazon Web services, in particular Lambda, Step Functions, §Cloud Formation, S3, DynmaoDB, AWS regions, SSM Parameters,
- Showcasing progress of the project to the team
- LateX
- CICD process
- Git
- Javascript fundamentals
- Project design and infrastructure
- Puppeteer and Mocha
- Docker

6.3 Challenges

Fortunately the team at ServisBOT were very supportive and as a result, working on this project posed very few challenges. However, there were a few which may be common to work based projects for students completing theirs in the workplace.

- It became obvious early in the process that the fundamentals of Javascript and object orientated programming was still challenging and not fully comprehended by myself. However, from being tasked with new problems every day and week, which explored varioius components, AWS services, features and technologies, an immense learning curve was accomplished and skills achieved at a pace which inevitably sped up the process by the week.

- As this was a work based project which was completed during an internship, the code had to be reviewed by a team member before being merged into the main branch. If the team was under pressure this could take a few days which disrupted the workflow.
- Learning how to use LateX for the project report initially was very confusing and stunted the process. It was almost as challenging as learning a new programming language yet the benefits from using it outweighed the frustrations.

6.4 Future Work

Incredibly at the end of the project, Bosco is at a state where the team can now take over and add the remainder of the tests. There are about forty tests that are to be transitioned from Frankenstein to Bosco which any team member once lightly versed in Puppeteer and Mocha can work on easily.

Appendix A

Methodology

Jira was used to keep track the sprints on the Bosco project. Week long sprints were the chosen method for the implementation phase as this was better suited to the type of project as the urgency to have it complete is high.

The projcet was broken into three Epics:

- Bosco Research and Design
- Bosco Implementation
- Bosco Dissemination Phase

A.1 Research and Design

Research and Design was carried out over a number of weeks starting in December with the main focus on the following:

- Review Puppeteer
- Review Lambda
- Initial Step Function R&D
- Cost Analysis

A.2 Implementation

The implementation of Bosco was carried out in one week sprints as was required by the project supervisor in ServisBOT.

Sprint 1

- Add two Frankenstein tests to Bosco
- Step Function Lambda code moved to Bosco

Sprint 2

- Cloudformation

Sprint 3

- Bosco environment variables for tests
- Refactor state machine to use a shared environment

Sprint 4

- Test profiles and overrides

Sprint 5

- Create a test with a secret

Sprint 6

- Lightning url in tests needs to be config in test profiles

Sprint 7

- READMe update for Developer Experience
- Schedule different test profiles to run on a CRON
- Orchestrator builds the Environment

Sprint 8

- Create a CLI test in Bosco
- Staging organization and SSM Parameters

Sprint 9

- Deployment of Bosco
- Storing Test Results in Dynamo DB
- Logging Test Results and Completion
- Bosco infrastructure to handle multiple ServisBOT regions with one deploy

Sprint 10

- Screenshots of failed tests upload to S3
- Alert of test failures and failed test runs

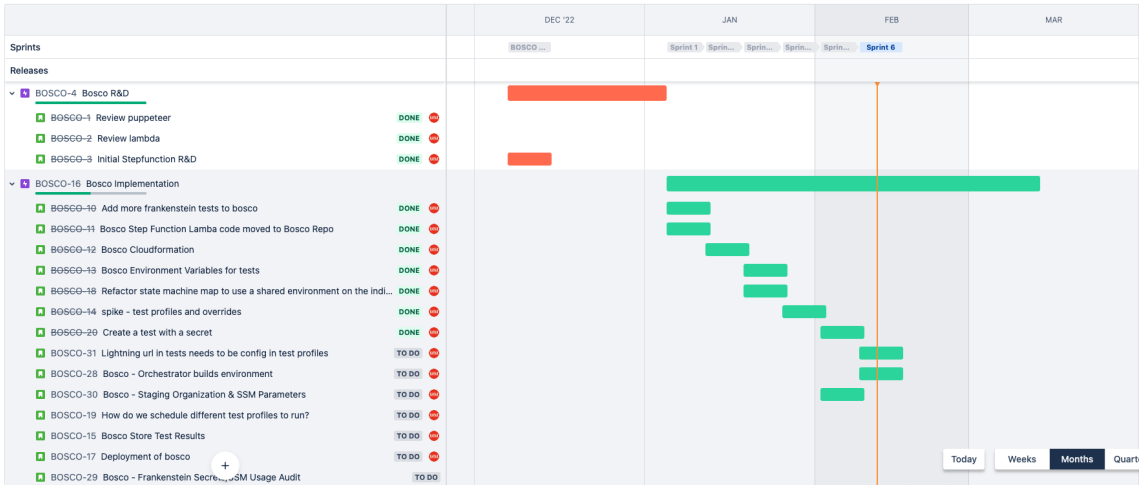


Figure A.1 – R&D Phase and Bosco Implementation Phase

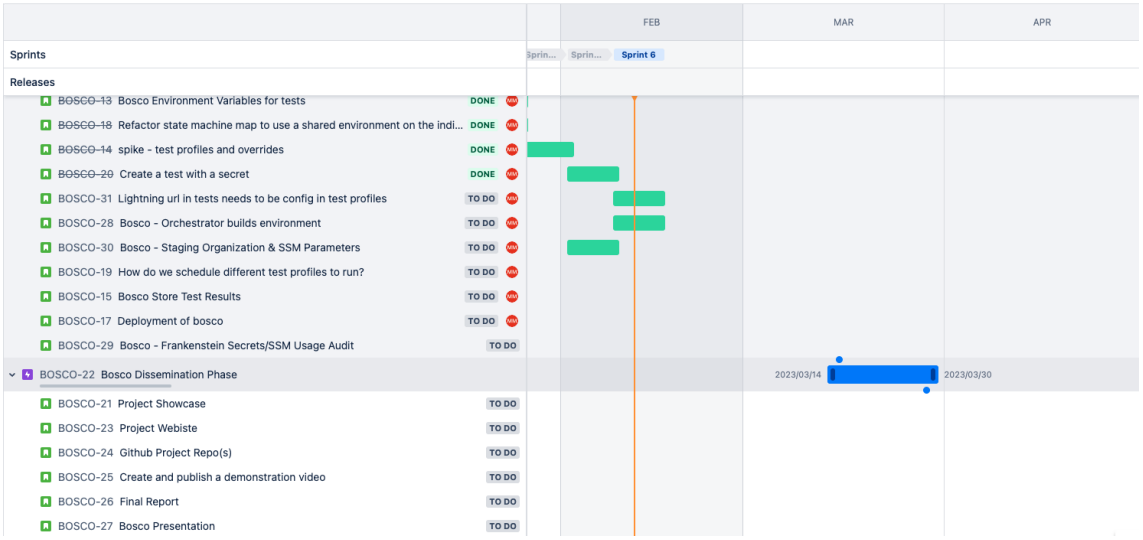


Figure A.2 – Bosco Dissemination Phase

References

[1] BrowserStack. (n.d.) *Puppeteer vs Selenium: Core Differences*. 2023.

[2] Ahamed. *Studying the Feasibility and Importance of Software Testing: An Analysis*. 2010. URL: https://arxiv.org/abs/1001.4193 (visited on 01/14/2023).

[3] Inc. (n.d.) Amazon Web Services. *AWS Step Functions Pricing — Serverless Microservice Orchestration — Amazon Web Services*. 2023.

[4] Brijesh Choudhary et al. "Case Study: Use of AWS Lambda for Building a Serverless Chat Application". In: Jan. 2020, pp. 237–244. ISBN: 978-981-15-0789-2. DOI: 10.1007/978-981-15-0790-8_24.

[5] S Crowther. *What is Puppeteer? Why Developers and Fraudsters Love it*. 2021.

[6] Evan Halley. *Retrieving AWS SSM Parameters with Node*. 2021. URL: https://evanhalley.dev/post/aws-ssm-node (visited on 02/12/2023).

[7] Jordan Hansen. *Puppeteer on AWS Lambda*. 2021. URL: https://oxylabs.io/blog/puppeteer-on-aws-lambda (visited on 11/04/2022).

[8] Mochajs.org. *Mocha - the fun, simple, flexible JavaScript test framework*. 2019.

[9] npm. *Puppeteer*. 2023.

[10] C Richardson. *Microservices*. 2017.

Appendix B

Bibliography

1. https://www.opsramp.com/guides/aws-monitoring-tool/cloudwatch-synthetics/
2. https://www.functionize.com/automated-testing/assertion
3. https://jestjs.io
4. https://www.browserstack.com/guide/unit-testing-for-nodejs-using-mocha-and-chai
5. https://www.tricentis.com/blog/bdd-behavior-driven-development
6. https://www.pluralsight.com/blog/software-development/tdd-vs-bdd
7. https://www.testim.io/blog/puppeteer-selenium-playwright-cypress-how-to-choose/
8. https://www.testim.io/blog/webinar-summary-is-ai-taking-over-front-end-testing/
9. https://aws.amazon.com/blogs/architecture/field-notes-scaling-browser-automation-with-puppeteer-on-aws-lambda-with-container-image-support/
10. https://moiva.io/
11. https://docs.aws.amazon.com/AmazonCloudWatch/
12. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html
13. https://oxylabs.io/blog/puppeteer-on-aws-lambda
14. https://aws.amazon.com/blogs/aws/new-for-aws-lambda-container-image-support/
15. https://www.npmjs.com/package/chrome-aws-lambda
16. https://www.docker.com/resources/what-container/
17. https://blog.logrocket.com/testing-node-js-mocha-chai/
18. https://www.ponicode.com/shift-left/
19. https://blog.shikisoft.com/3-ways-to-schedule-aws-lambda-and-step-functions-state-machines/
20. https://evanhalley.dev/post/aws-ssm-node/
21. https://dockerlabs.collabnix.com/beginners/components/what-is-container.html
22. https://www.youtube.com/watch?v=rQijrDj1wCQ