Frankenstein Test-Suite Re-Architecture

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

This project aims to design and implement a feature based test runner for ServisBOT which determines which features of a system are working and which are down. The proposed system has the working title Bosco.

ServisBOT Ltd runs an online platform that creates chatbots which provide customer service by allowing end users to communicate with a business or service through a pop up messenger on a website. The technology is cloud based and mostly serverless which means it is provided over the internet rather than using storage on a physical computer or server. It has to be consistently monitored and so tests are run continuously to ensure any problems are detected and resolved immediately. Testing the code is crucial to the service they provide. It ensures a robust platform and gives customers confidence that the system is dependable.

1.1.1 Frankenstein

The components of a chatbot and each of its functions are broken down into micro services. These are created independent of one another but combined are the building blocks for the chatbot. There is a suite of tests which ServisBOT have called Frankenstein tests which run against these micro services. They run several times an hour, every hour.

1.1.2 TestCafe

The tests use a software package called Testcafe which is an end to end testing service that uses messenger in a browser in order to run their tests. It simulates what a user would do by opening a browser and 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.

- Testcafe has proven to be 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.

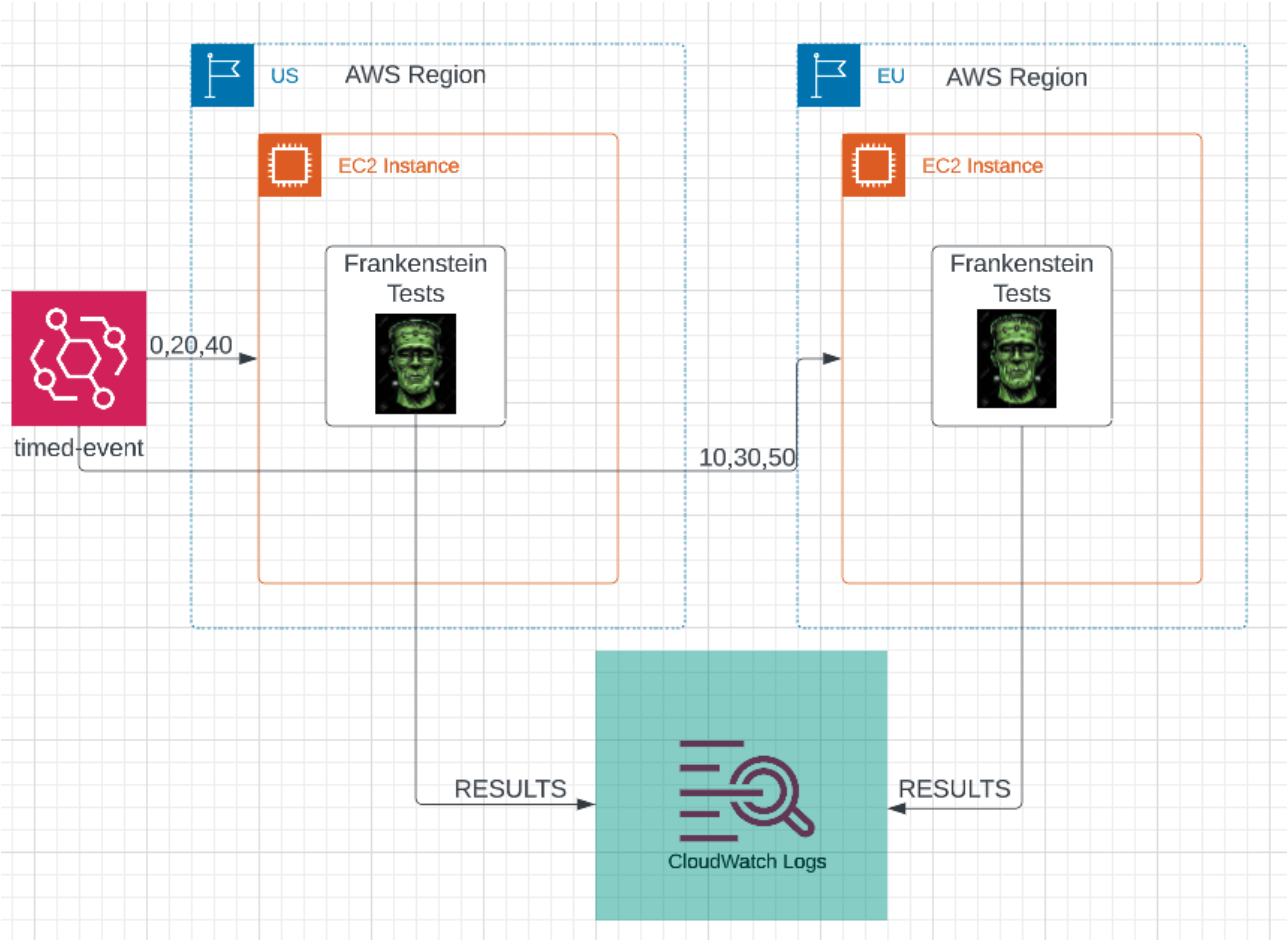


Figure 1.1: High level view of Frankenstein

1.3 Purpose and Requirements for Bosco

This project aims to completely overhaul the existing test suite by migrating the tests from the EC2 instances to AWS Lambda functions. This would ensure each lambda, whether it runs one test or multiple tests, is run independently and is not competing for memory. It would mean the test suite could be scaled infinitely, would run faster and as a result would most likely be a lot more cost efficient. This has yet to be proven.

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 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.

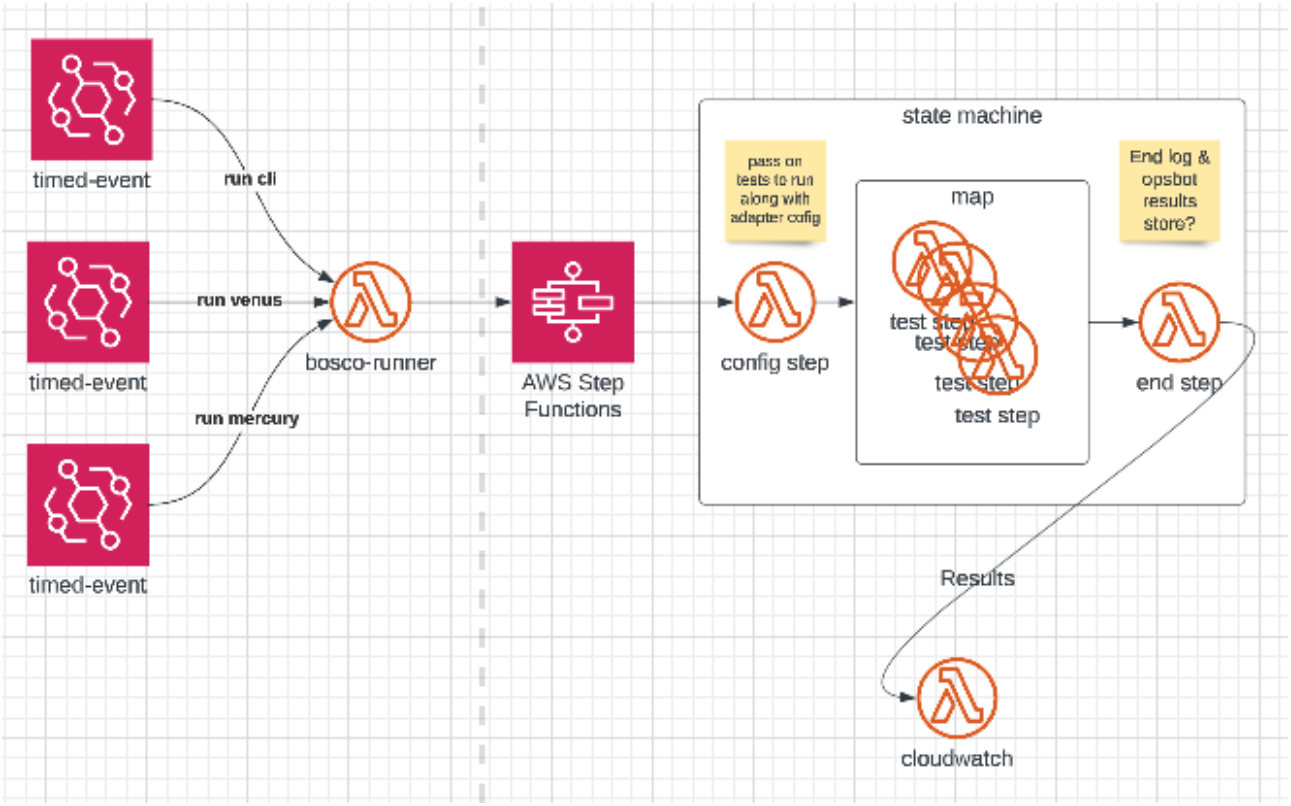


Figure 1.2: High level view of Bosco

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:

- Review of Frankenstein and Testcafe
- 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
- Proof of Concept
- Additional requirements for implementing Bosco
- Scope of the project

2.1 Review of Frankenstein and Testcafe

Frankenstein is a feature based test runner designed to determine which features of a system are up and working and which are down. The tests are run in AWS ECS and are run on a cron.

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	

Table 2.1: Pros and Cons of Testcafe

2.2 Cost Analysis of Frankenstein Tests

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

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

2.3 Research and Analysis of Possible Testing Frameworks for Bosco

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

2.3.1 Mocha

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. 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.

2.3.2 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.

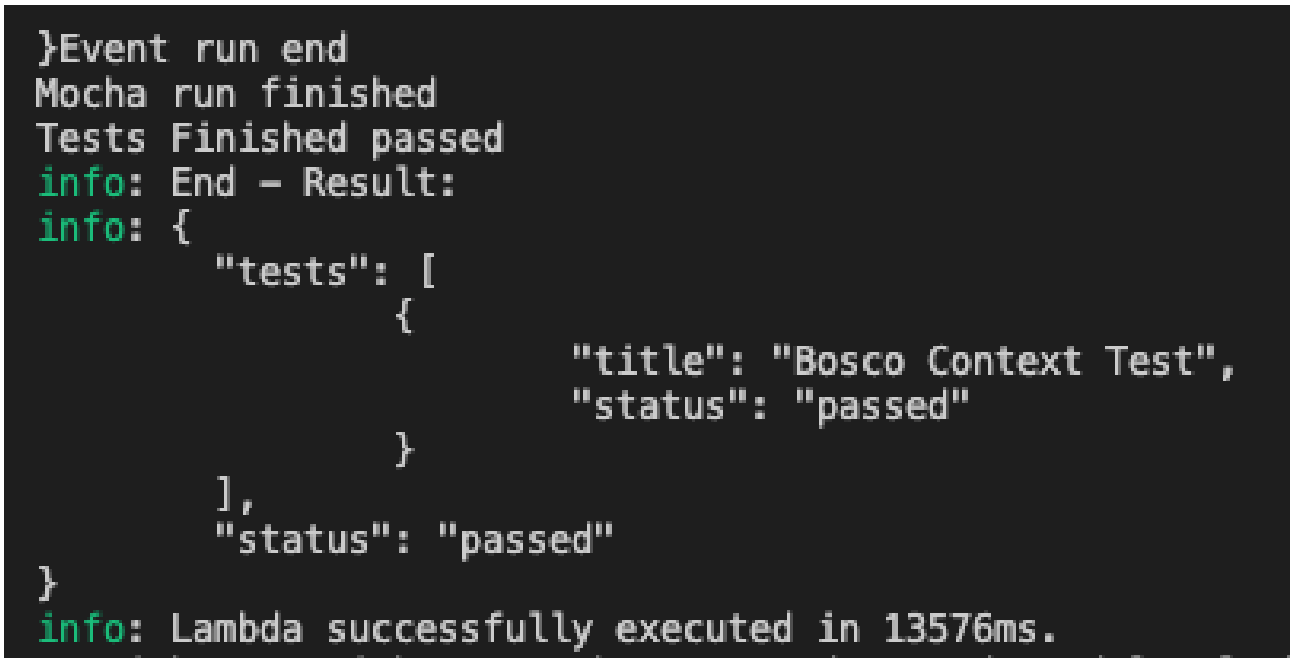


Figure 2.2: Mocha Test Result Output

2.3.3 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

Table 2.2: Mocha Vs Jest

2.4 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.

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: speed vs. reliability. Puppeteer aimed 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.

2.4.1 Pros of using Puppeteer

- Simple to set up.
- Good documentation. Small community but lots of tutorials at this point.
- Promise based.

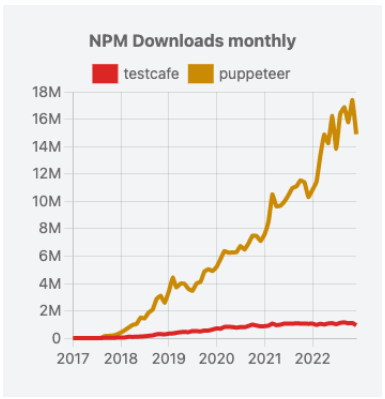
- Scriptable web browser
- Installs Chrome in a working version automatically
- Thin wrapper
- Bi-Directional (events) automating things like console logs is easy
- Maintained by Google.
- JavaScript first, so the code feels very natural
- Puppeteer also gives you direct access to the Chrome DevTools Protocol if you need it. Which can be very useful at times and in general it feels 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 agent which allows for user agent 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 wait for certain thing but has to waitFor manually for others.
- Debugging: Can write and debug Javascript from an IDE.

2.4.2 Cons of using Puppeteer

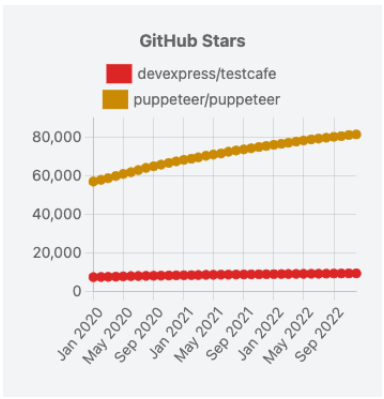
- Limited cross-browser support—only Chrome and Firefox
- Feels like an automation framework and not a test framework—you often have to re-implement testing-related tools
- Grids (running concurrently) in production are often a challenge
- The automatic browser set up downloads Chromium and not Chrome 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. Usually people build their own but this is due to change soon.
- Does not support self healing tests and automatically improving tests.
- Does not support Autonomous testing which is testing without code or user intervention.

2.4.3 Testcafe Vs Puppeteer

Puppeteer is a Node library which provides browser automation for chrome and chromium. TestCafe is a Node.js tool to automate end-to-end web testing. Puppeteer runs headless by default, but can be configured to run full (non-headless) Chrome or Chromium; It provides a high-level API to control Chromium or Chrome over the DevTools Protocol. TestCafe runs on Windows, MacOs, and Linux and supports mobile, remote and cloud browsers (UI or headless). It is also free and open source.

h 

(a) Npm Downloads



(b) GitHub stars

Figure 2.3: NPM statistics

2.5 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.

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.

2.6 Conclusion

// TODO

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

The mocha test framework was added to the code as in reality Puppeteer is not a test framework but an automation tool. 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.

```
// simple-test.js
const puppeteer = require("puppeteer");
const url = process.argv[2];
if (!url) {
    throw "Please provide URL as a first argument";
}
async function 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: Simple test written with Puppeteer

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 MBṪhis limit does not apply when uploaded from an S3 bucket but

there are other issues. Linux including AWS Lambda does not include the necessary libraries required to allow Puppeteer to function.

3.1.2 Node Modules

There are work-arounds for this in the form of node modules, ṕuppeteer-coreánd ćhrome-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 is at present at version 19.3.0. When the version numbers are synced the lambda function passes. Obviously this is not ideal as there is a vast difference between versions. Whatever the reason for these modules not being updated, relying on them is impractical and will eventually lead to our tests being broken.

3.1.3 Docker

The process uses a Docker container 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. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging.

The puppeteer test runs locally using a Docker engine, then gets tagged and pushed to a container on a Lambda function named ṕuppeteer-container. 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 tests the use of Puppeteer and Mocha as an automation tool in a Lambda container and finds 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 recorded in Cloudwatch logs for monitoring and assessment.

3.2 Initial Stepfunction Research and Development

One possible infrastructure that is important to explore is the use of of step functions for the test suite. 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.

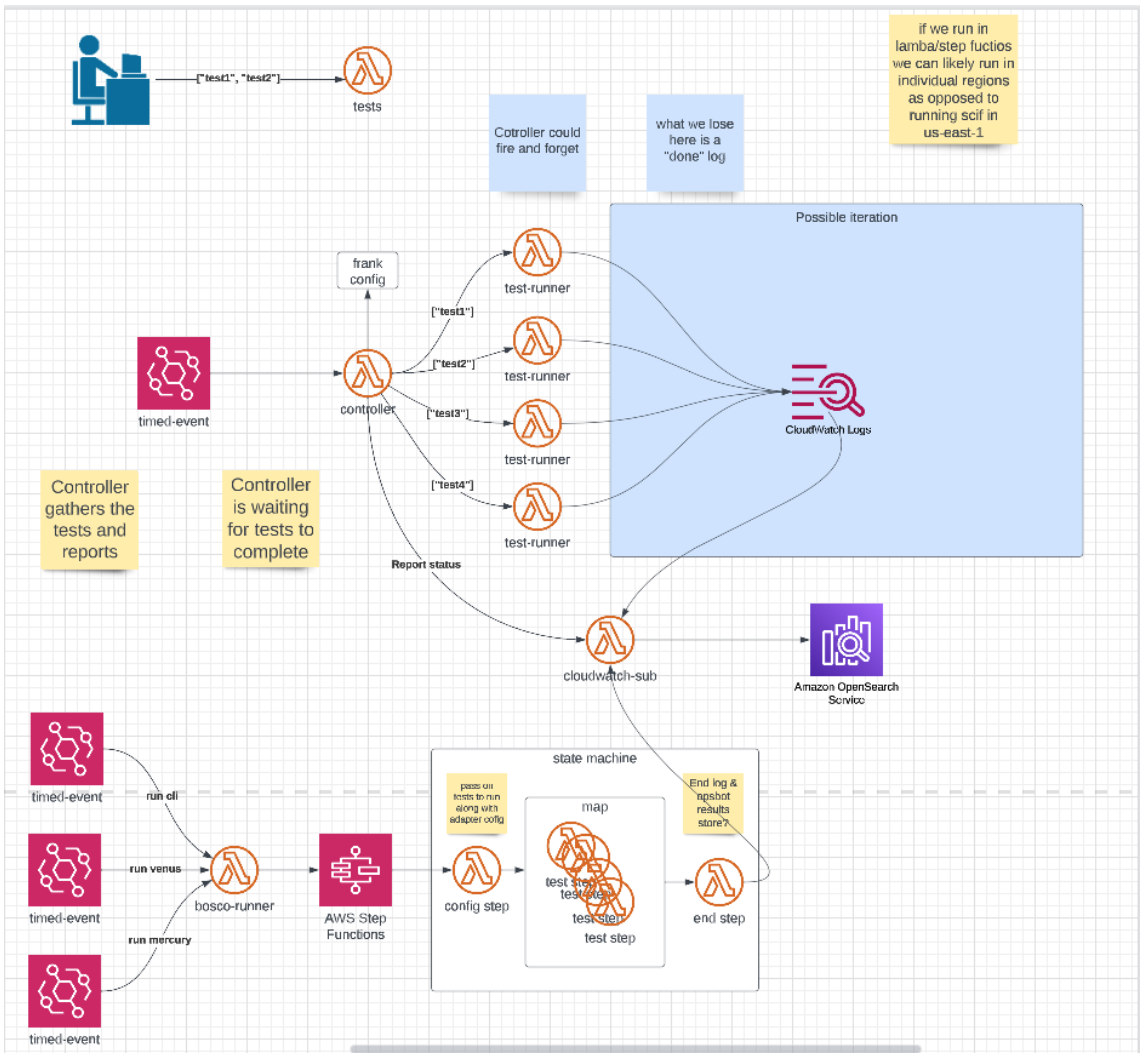


Figure 3.2: Bosco Planned Implementation

3.3 State 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 8 states as the R̈unningstate is executed three times. The first state is the S̈tartstate which initiates the run and then the S̈tartStateïs invoked, creating an array of tests and passing the payload to the M̈apstate.

This experiment focuses on using the Inline Map state instead of the Distributed Map state. The Inline Map state is used for fewer than 40 parallel iterations while the Distributed Map state is used for larger workloads. The map state runs a lambda for each test, which runs in parallel with each other. The results of the tests are then outputted to the D̈onelambda, the next state, which processes and runs the tests. Finally, a state prints out the results and the End state completes the workflow.

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. Further investigation will take place when actual Frankenstein tests can be added to our state machine.

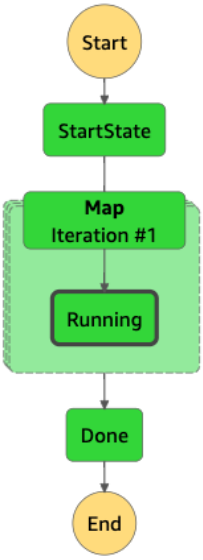


Figure 3.3: Bosco State Machine

3.3.2 Cost Analysis of Step Function State Machine

The cost of running the test suite in a state machine using AWS Step Functions is based on the number of state transitions. 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 may vary for Bosco tests.

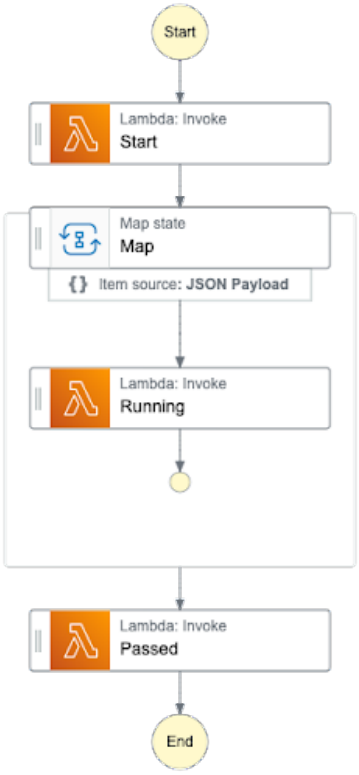


Figure 3.4: Inline Map State for Bosco Step Function

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
	usscil-1		15	15	20	60	43800	$ 1.10	$ 13.14
	usscil-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 function state machine there is 5 state transitions plus a transition for each test excluding any error handling

Figure 3.5: 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 I was able to finish writing the test and ensure it was working correctly. I made a PR, Dean reviewed it. I made the changes Dean suggested which included tidying up the code.

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 Cloud Formation

The Bosco project will be deployed using Cloud Formation, a deployment tool. The Cloud Formation 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 Cloud Formation and have the state machine reference these lambdas. Once this is complete, the state machine will invoke the lambdas.

The cloud formation 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:

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

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

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

```
{
  "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.

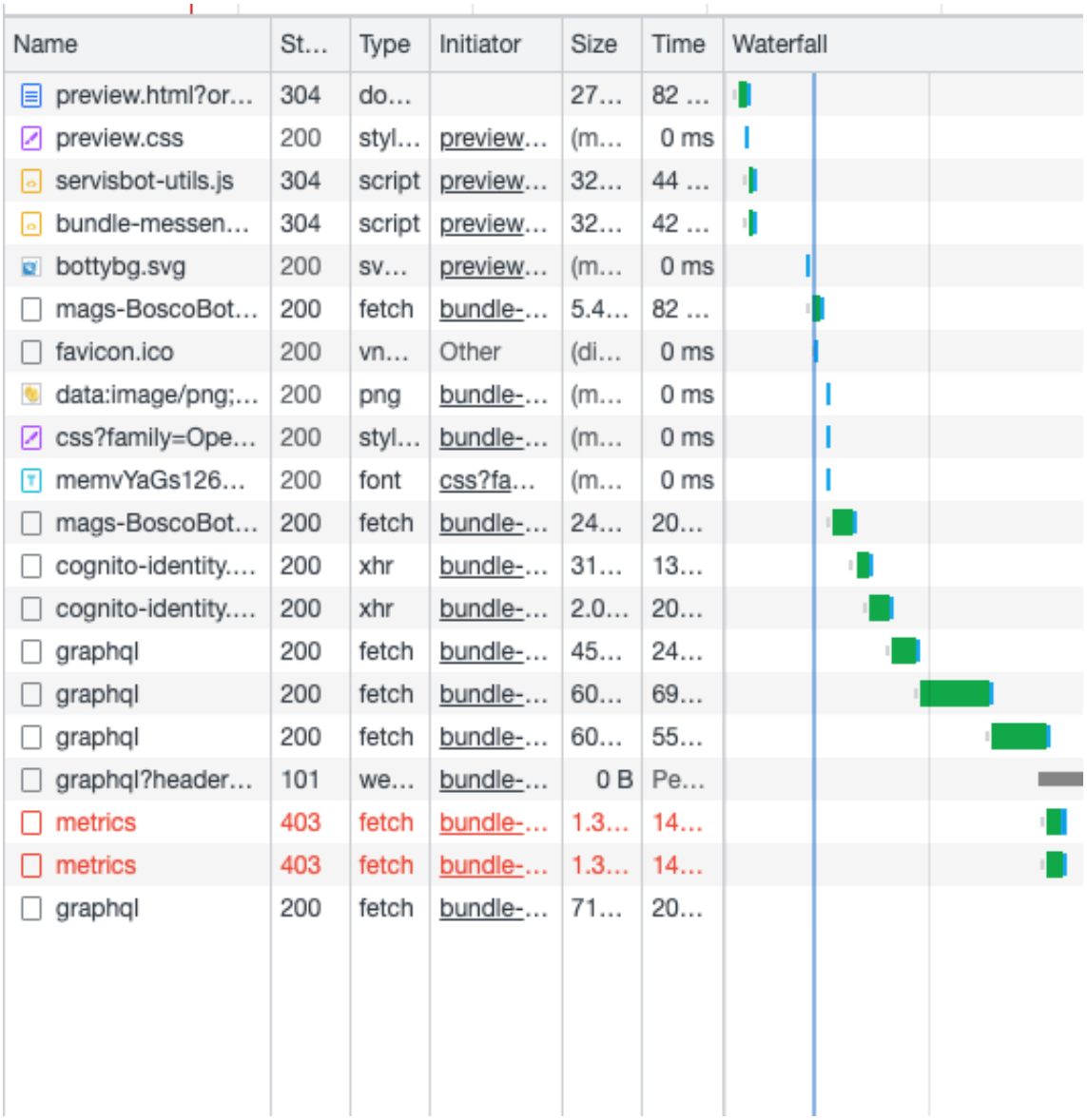
An Environment object was created using the environment from the event. This could now be accessed in the tests by calling the getters from the Environment object.

```
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.

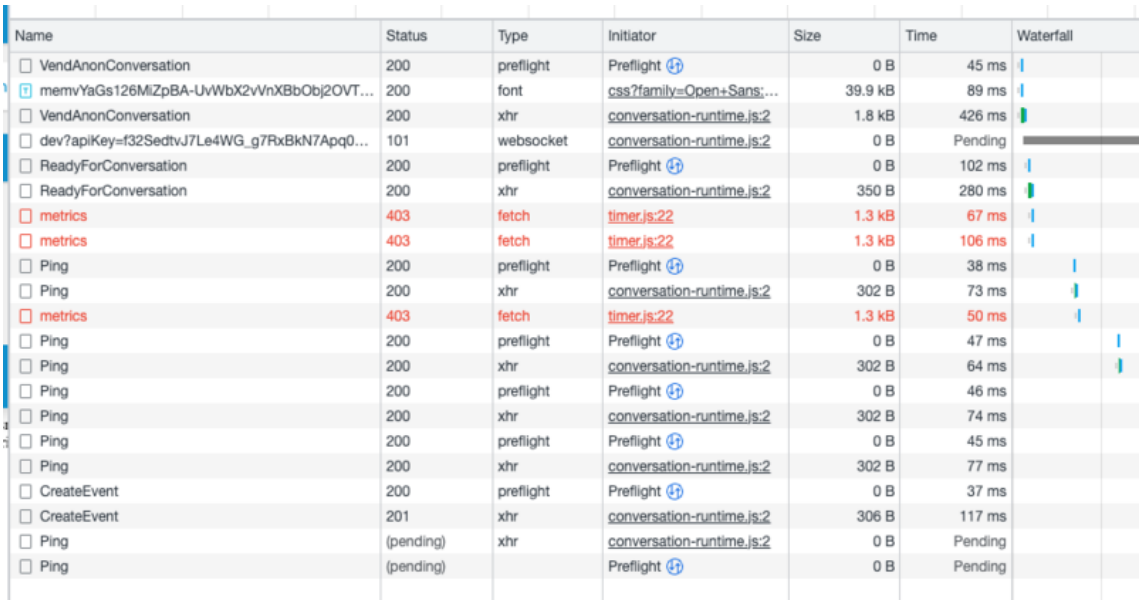


Name	St...	Type	Initiator	Size	Time	Waterfall
preview.html?or...	304	do...		27...	82 ...	
preview.css	200	styl...	preview...	(m...	0 ms	
servisbot-utils.js	304	script	preview...	32...	44 ...	
bundle-messen...	304	script	preview...	32...	42 ...	
bottybg.svg	200	sv...	preview...	(m...	0 ms	
mags-BoscoBot...	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

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



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

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.

4.11 SSM Parameters

The next task for Bosco was to build the environment in the orchestrator using SSM parameters. SSM parameters are credentials and secrets that are stored in AWS Systems Manager parameter store. 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

```
boscoLongLivingLexV2euprivate3hourly - jobStatusStdOut : The current status of the job is RUNNING, pollCount : 0
boscoLongLivingLexV2euprivate3hourly - jobStatusStdOut : The current status of the job is RUNNING, pollCount : 1
boscoLongLivingLexV2euprivate3hourly - jobStatusStdOut : The current status of the job is RUNNING, pollCount : 2
boscoLongLivingLexV2euprivate3hourly - jobStatusStdOut : The current status of the job is RUNNING, pollCount : 3
boscoLongLivingLexV2euprivate3hourly - jobStatusStdOut : The current status of the job is RUNNING, pollCount : 4
boscoLongLivingLexV2euprivate3hourly - jobStatusStdOut : The current status of the job is RUNNING, pollCount : 5
boscoLongLivingLexV2euprivate3hourly - jobStatusStdOut : The current status of the job is RUNNING, pollCount : 6
boscoLongLivingLexV2euprivate3hourly - jobStatusStdOut : The current status of the job is RUNNING, pollCount : 7
boscoLongLivingLexV2euprivate3hourly - jobStatusStdOut : The current status of the job is RUNNING, pollCount : 8
boscoLongLivingLexV2euprivate3hourly - jobStatusStdOut : The current status of the job is RUNNING, pollCount : 9
boscoLongLivingLexV2euprivate3hourly - jobStatusStdOut : The current status of the job is RUNNING, pollCount : 10
boscoLongLivingLexV2euprivate3hourly - jobStatusStdOut : The current status of the job is RUNNING, pollCount : 11
boscoLongLivingLexV2euprivate3hourly - jobStatusStdOut : The current status of the job is RUNNING, pollCount : 12
boscoLongLivingLexV2euprivate3hourly - jobStatusStdOut : The current status of the job is RUNNING, pollCount : 13
boscoLongLivingLexV2euprivate3hourly - jobStatusStdOut : The current status of the job is RUNNING, pollCount : 14
boscoLongLivingLexV2euprivate3hourly - jobStatusStdOut : The current status of the job is RUNNING, pollCount : 15
boscoLongLivingLexV2euprivate3hourly - jobStatusStdOut : The current status of the job is SUCCEEDED, pollCount : 16
boscoLongLivingLexV2euprivate3hourly - Publish succeeded.
Finished setup for Nlp Worker LexV2 Intent Publish puppeteer test + ' '+
     bosco1676754118946 at (9:03:44 PM).
Running Nlp Worker LexV2 Intent Publish puppeteer test at  1676754224732
Started HeadlessChrome/108.0.5351.0.
Starting teardown for Bosco test at 1676754264539 (9:04:24 PM)
Finished puppeteer test at (9:04:24 PM).
```

Figure 4.9: NLP worker test with polling

aws-sdk.

- A new policy was added to the cloud formation template in order to give IAM permissions to read from the parameter store
- The environment was built in the orchestrator.
- The logger was added as a parameter to cloud formation instead of being inputted in the test profile environment
- A new class Secrets was added in order to retrieve the SSM parameters from the parameter store. Later to be renamed EnvironmentResolver
- The documentation was updated

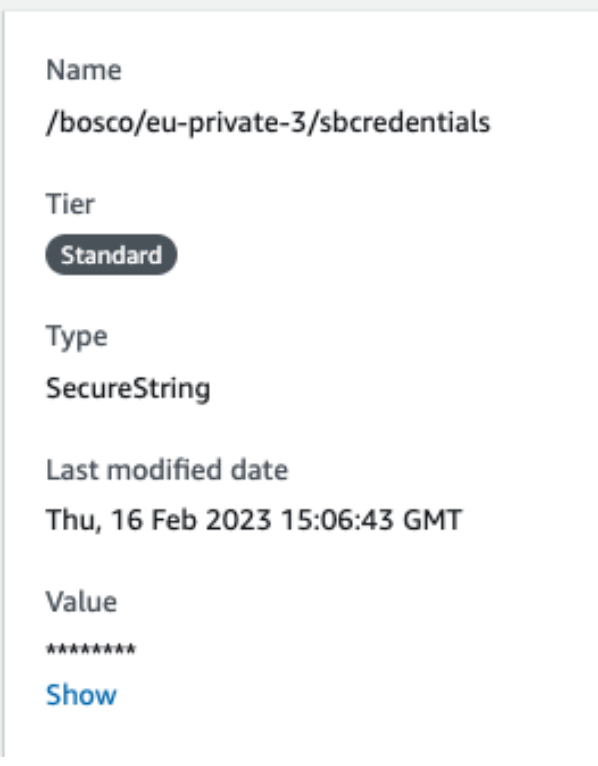


Figure 4.10: SSM Parameters created in AWS Systems Manager

Appendix A

Methodology

I am currently using Jira to track my sprints on the Bosco project. I am working in week long sprints for the implementation phase as this is better suited to the type of projec this is as this will be a live running system when it's complete and the urgency for it to be complete is high.

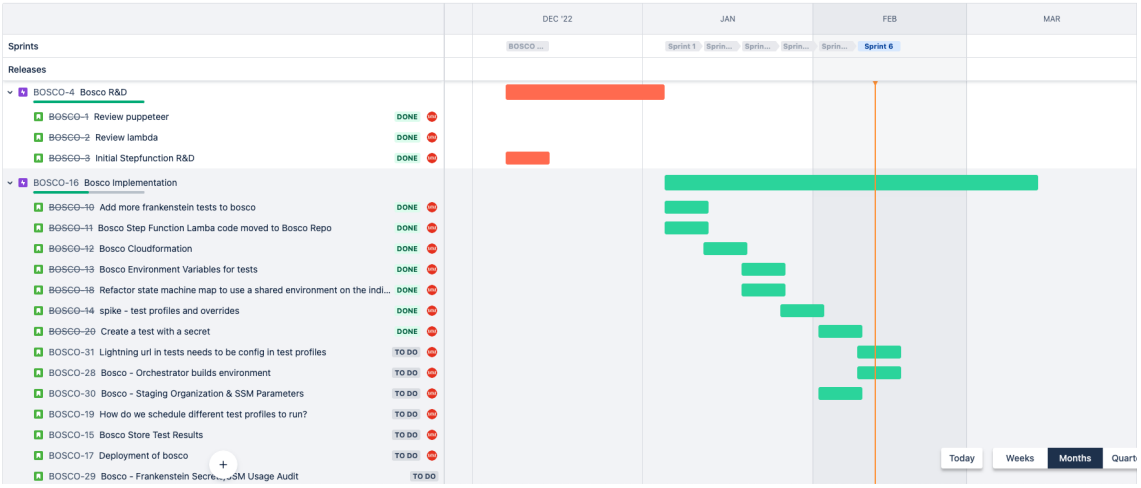


Figure A.1: R&D Phase and Bosco Implementation Phase

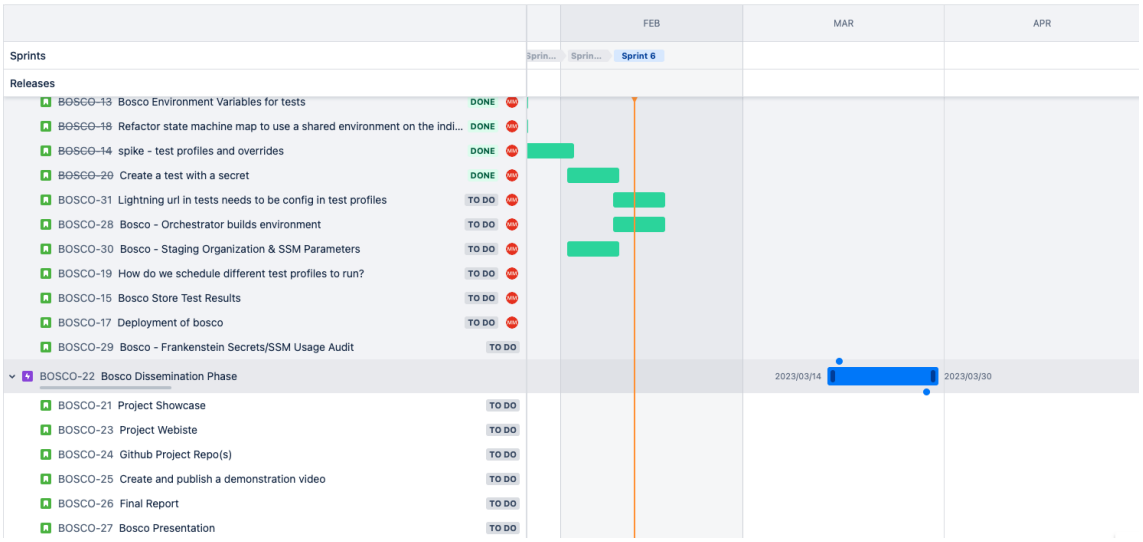


Figure A.2: Bosco Dissemination Phase

Appendix B

Bibliography

https://www.opsramp.com/guides/aws-monitoring-tool/cloudwatch-synthetics/ https://www.functionize.co
testing/assertion https://jestjs.io https://www.browserstack.com/guide/unit-testing-for-nodejs-
using-mocha-and-chai https://www.tricentis.com/blog/bdd-behavior-driven-development https://www.plura
development/tdd-vs-bdd https://www.testim.io/blog/puppeteer-selenium-playwright-cypress-
how-to-choose/ https://www.testim.io/blog/webinar-summary-is-ai-taking-over-front-end-testing/
https://aws.amazon.com/blogs/architecture/field-notes-scaling-browser-automation-with-puppeteer-
on-aws-lambda-with-container-image-support/ https://moiva.io/ https://docs.aws.amazon.com/AmazonClo
https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html https://oxylabs.io/blog/puppet
on-aws-lambda https://aws.amazon.com/blogs/aws/new-for-aws-lambda-container-image-support/
https://www.npmjs.com/package/chrome-aws-lambda https://www.docker.com/resources/what-
container/ https://blog.logrocket.com/testing-node-js-mocha-chai/ https://www.ponicode.com/shift-
left/ https://blog.shikisoft.com/3-ways-to-schedule-aws-lambda-and-step-functions-state-machines/
https://evanhalley.dev/post/aws-ssm-node/

Appendix C

References

//TODO