Optimising Test Runner Performance for Chatbot Technology

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

ServisBOT which is based in ArcLabs in Waterford, specialises in chatbot technology and conversational AI. ServisBOT 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 primarily serverless, meaning services are provided over the internet rather than by using storage on a physical computer or server.

The company has a global customer base so the system needs to be consistently monitored and therefore tests are run continuously to ensure any problems are detected and resolved immediately.

ServisBOT's current approach to software testing involves a system named Frankenstein. However, there are numerous issues with Frankenstein, in particular the fact that the number of tests Frankenstein can handle is at its limit, so the test coverage of ServisBOT's platform is insufficient.

The objective of this project is to create and execute a test runner for ServisBOT that will replace Frankenstein. This will involve designing and implementing a system that can run each test independently without competing for memory. By doing so, the test suite could be scaled infinitely, perform faster, and potentially be more cost-effective.

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 typically independently deployable and loosely coupled. In essence its different parts can operate independently without being too dependent on each other.

The microservice architecture enables an organisation to deliver large, complex applications rapidly, frequently, reliably and sustainably - a necessity for competing and winning in today's world.[8]

Frankenstein runs tests against these micro services, testing the functions of the chatbot. They run several times an hour, every hour.

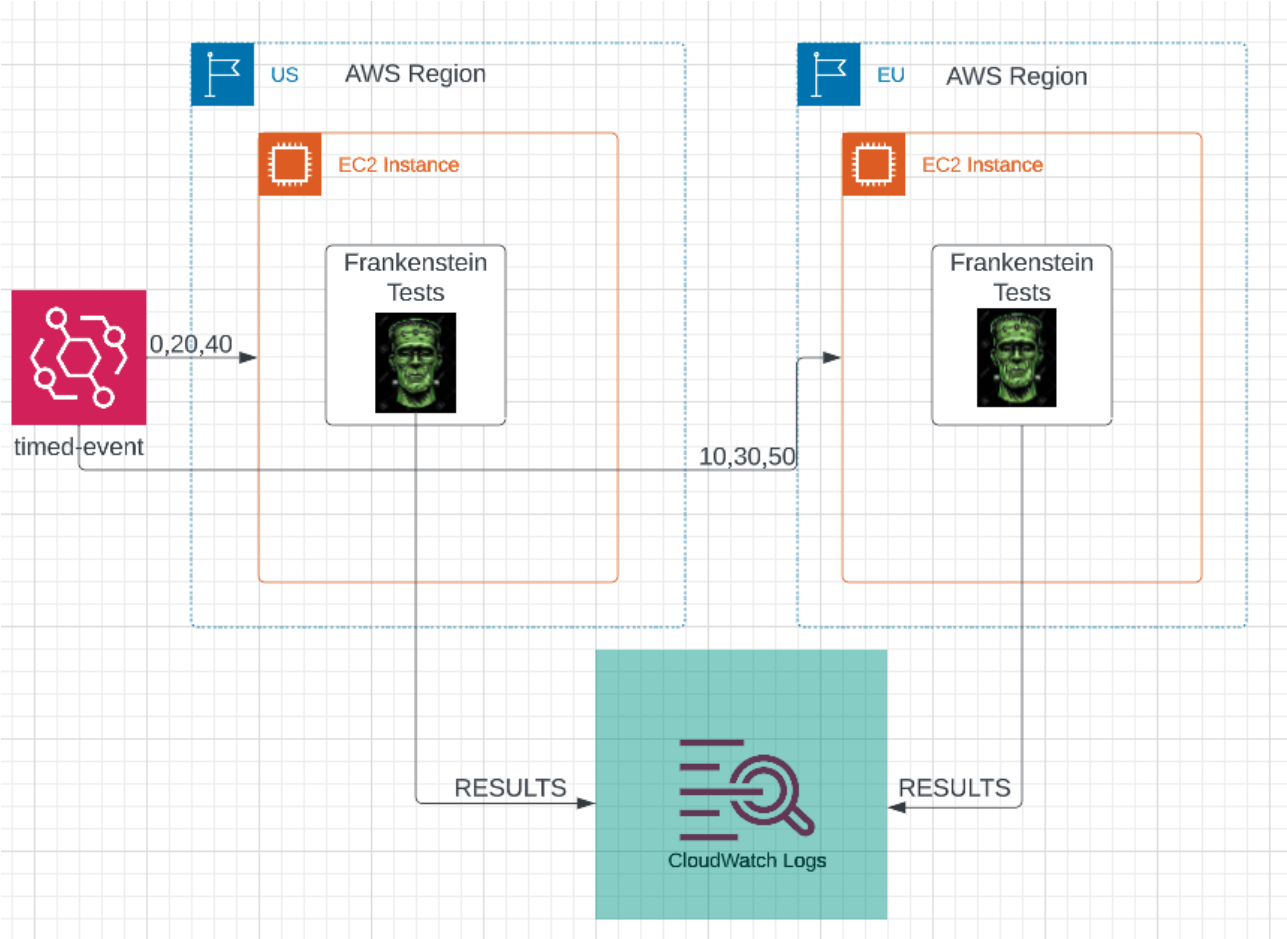


Figure 1.1 – High level view of Frankenstein

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 Amazon Web Services (AWS) regions, the US and the EU. AWS Elastic Compute Cloud instances (EC2) are virtual machines where developers can define the resources they need to run their programs. For example, what regions they are to be run in and how much memory they need.

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 finds Testcafe resource heavy, expensive and inefficient, causing slow CPU performance.
- Because of competing resources some tests affect the performance of others.
- Debugging and monitoring of Testcafe tests is complex, it uses a different syntax and approach to testing compared to other testing frameworks.

1.3 Requirements for Bosco

The primary requirements for Bosco are as follows:

- Create a new test runner, that is more optimised, scalable, and cost-efficient than Frankenstein.
- In particular, Bosco 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) [7]
- Implement an AWS Lambda, Step Function, State Machine which will run each test independently and can be scaled infinitely.
- Scheduled timed events will instigate test runs.
- Bosco will have test profiles which can determine which tests are run in which regions and on which adapters.
- Bosco will report the latest results of the test runs to Cloudwatch and store them in a DynamoDB table.

- Screenshots of failed test runs will be stored.

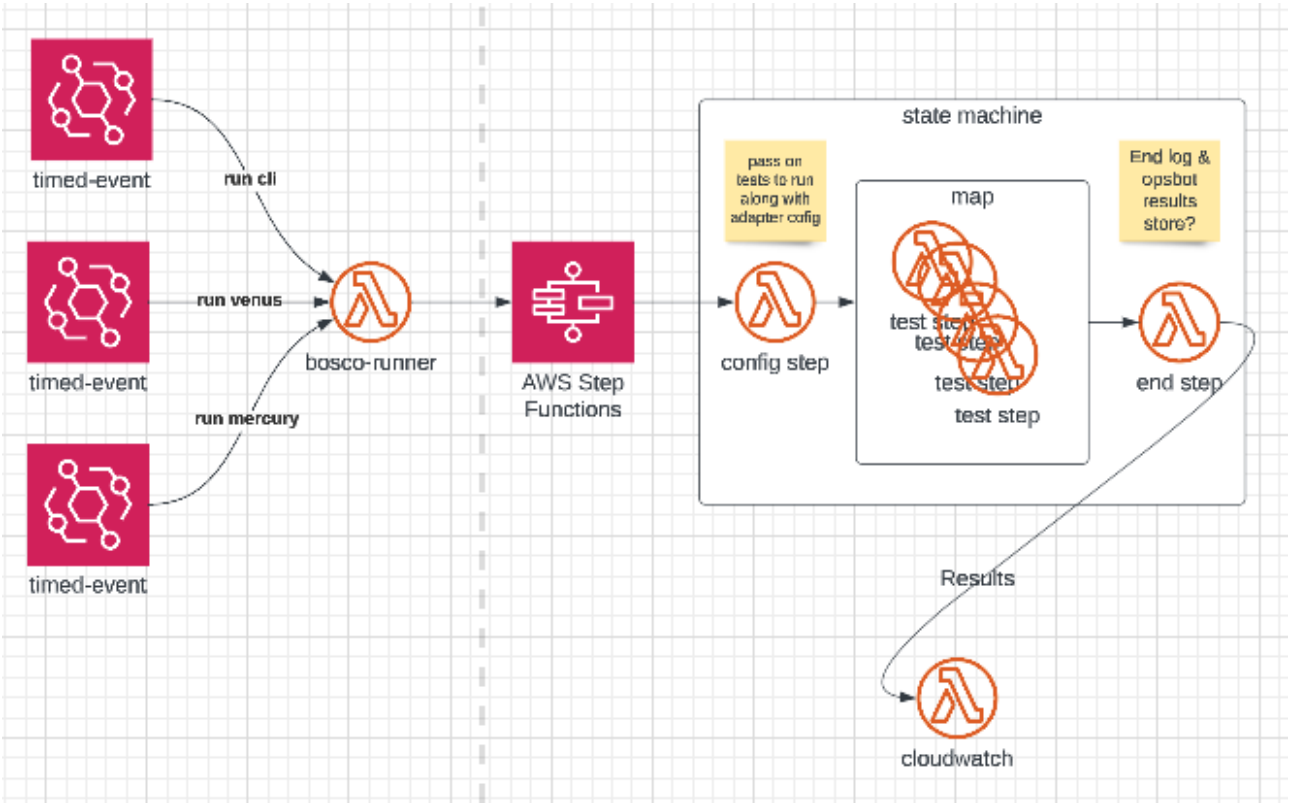


Figure 1.2 – High level view of Bosco

Chapter 2

Research and Development

In order to determine the best approach, technologies and methodology for Bosco, the research carried out 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

Cost analysis was carried out on Frankenstein which is shown in Figure 2.1. There is two EC2 instances running the tests in the EU and the US with two ServisBOT regions eu-private-1 and eu-1 in the EU and us-1 and usscif-1 in the US. All regions are in production except for eu-private-1 which is the testing region where most of the tests are run against before they are released to production. In summary the cost analysis concluded that Frankenstein is 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,*

Number of Tests per Region											
	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
EU	eu-private-1		61	72	4.8 sec	$ 7,745.76	$ 645.48	744	$ 20.82	$ 0.43	$ 0.14
	eu-private-1	20									
	eu-1		15					744			
	eu-1	9									
Total EU		29	76	72	0	$ 7,745.76	$ 645.48	1488	$ 20.82	$ 0.43	$ 0.14
US	us-1		16	72	5.48 sec	$ 7,197.72	$ 599.81	744	$ 19.35	$ 0.40	$ 0.13
	us-1	11									
	usscif-1		15					744			
	usscif-1	10									
Total US		21	31	72	0	$ 7,197.72	$ 599.81	1488	$ 19.35	$ 0.40	$ 0.13
TOTAL		50	107	144	0	$ 14,943.48	$ 1,245.29	2976	$ 40.17	$ 0.83	$ 0.28

Figure 2.1 – Frankenstein Cost Analysis

allowing for flexible and accurate reporting, while mapping uncaught exceptions to the correct test cases.[6]

Asynchronous testing is used to verify code behavior in software that relies on asynchronous programming. This programming allows tasks to be performed concurrently instead of waiting for each task to complete before moving on to the next one.

Mocha, a testing framework, provides functions that execute in a specific order and logs results in the terminal. Mocha also cleans the software state to ensure that test cases run independently.

```
}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 widely used test automation tool maintained by Google. Although it is not a dedicated test tool, it is popular for automating tasks such as web scraping and generating PDFs. In testing user flows within web applications, headful or headless browsers are used. A headless browser is a browser that is working in the background without a user interface and a heaful browser will automate the actions while displaying the browser. Headful browsers like Firefox with Selenium are slower but more reliable, while headless browsers like Testcafe sacrifice some reliability for speed.

Puppeteer aims to eliminate the need for developers to choose between speed and reliability when testing web applications. It does this by providing access to the Chromium browser environment, allowing developers to use either headless or headful browsers that run on the same platform as their users.

2.3.1 Pros of using Puppeteer

- Simple to set up.
- Good documentation with lots of tutorials.
- Promise-based which is a software technique that is designed to manage tasks that do not occur in a specific order. It allows these tasks to happen simultaneously without any issues.
- 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 which are events by a user, like hovers.
- End to end tests are very fast.
- 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

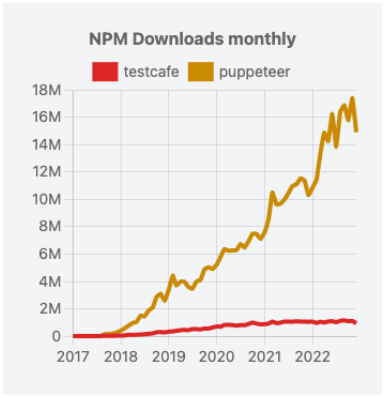
- Limited cross-browser support, it only supports Chrome and Firefox.
- Feels like an automation tool and not a test framework. Often developers have to re-implement testing-related tools.
- Grids (running concurrently) in production are often a challenge.
- The automatic browser set-up, downloads Chromium and not Chrome and there are subtle differences between the two.
- Smarter locators: No support for selecting elements in multiple ways.
- The software lacks the ability to run tests simultaneously on multiple computers, which can cause inefficiencies when testing large applications.
- The software cannot improve or fix test issues automatically, requiring manual intervention to correct problems.
- Does not support Autonomous testing which is testing without code or user intervention.

2.3.3 Testcafe

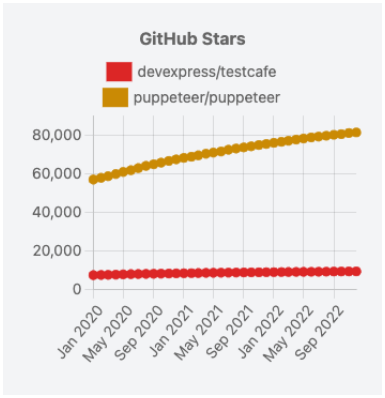
TestCafe is a Node.js tool to automate end-to-end web testing. It runs on Windows, MacOs, and Linux and supports mobile, remote and cloud browsers (UI or headless). It is also free and open source.

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	

h 

(a) Npm Downloads



(b) GitHub stars

Figure 2.3 – Puppeteer vs Testcafe NPM statistics

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.

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 [3]

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

After conducting research and analysis, it was affirmed that ServisBOT's decision to use Puppeteer in conjunction with Mocha, instead of using Testcafe will provide greater control, reliability, and ease of use for developers. Together with migrating the test runner from EC2 to Lambda, the test suite will become infinitely scalable and more cost efficient. Replacing Frankenstein with Bosco 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 was 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 was launched with the URL of the messenger page. After the page loaded, the messenger button was toggled and the chatbot displayed. The test passed if the response 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[5], developed 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 ensured the tests ran 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.

Regardless of 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 is a self-contained piece of software that packages up all the code and its dependencies needed to run an application. This ensures that the application runs reliably and consistently across different computing environments, even during different stages of development or on different computers with different settings. Using Docker containers instead of node modules can help to condense the code and simplify the application's setup.

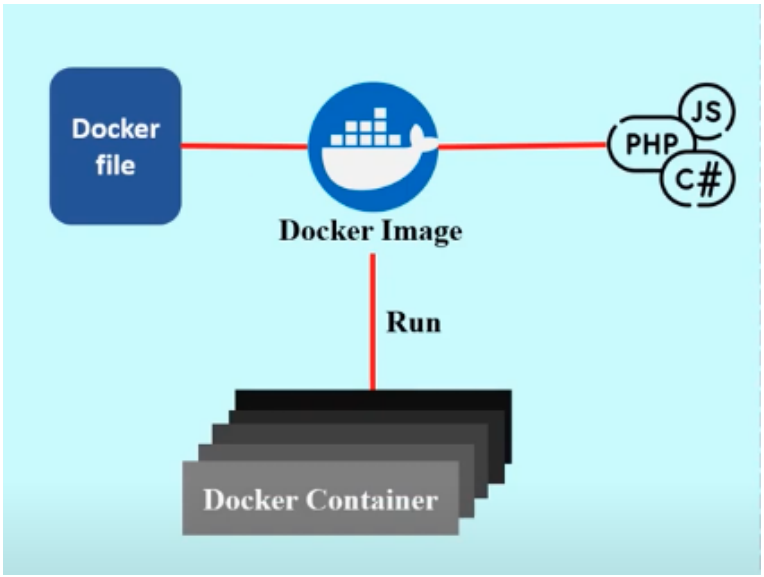


Figure 3.2 – Illustration of Docker

3.1.4 Conclusion

The puppeteer test was run locally in a Docker engine. When the test was functioning as expected the code was zipped up into docker image and got pushed to an AWS repository. The image was then deployed to the Bosco test-runner container where it was run and tested. This significantly sped up the testing process, running a test in six seconds. This process provides more control over the testing environment and can target specific tests through the event handler input parameters. The results validate the feasibility of running tests in a Lambda environment using Puppeteer.

3.2 Initial Stepfunction Research and Development

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

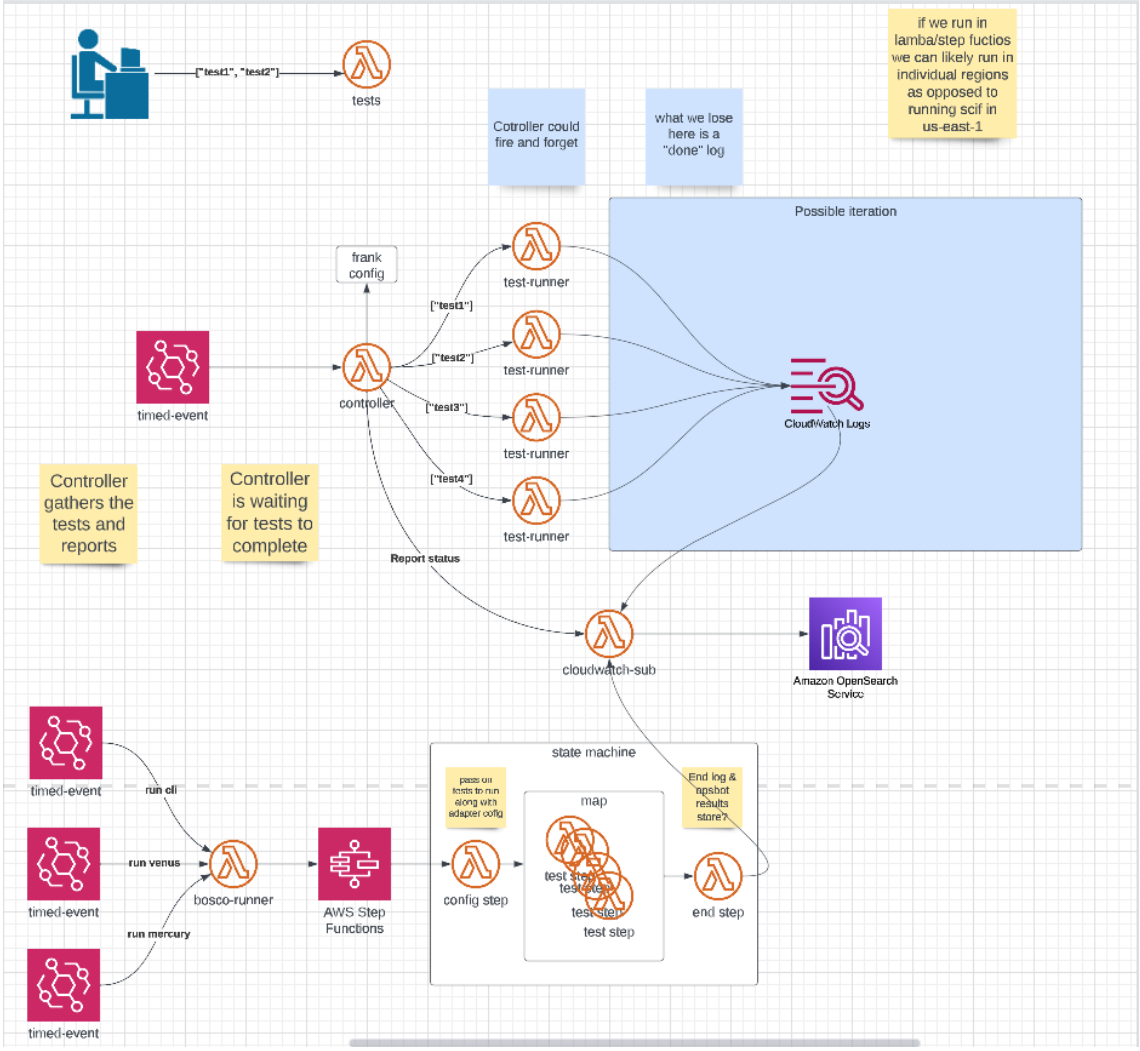


Figure 3.3 – Bosco Initial Possible Implementations

3.3 State Machine Transitions

The Bosco test suite runs through five state transitions.

- Start
- StartState
- Map
- Done
- End

Plus an additional transition per test that is run in the **Running** state. For example 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.

Bosco uses an Inline Map state which is suitable for fewer than forty parallel iterations which is appropriate for the number of tests that Bosco will run. The object structure passed to the **Running** lambda has been designed to enable the running of an array of tests or a single test, providing flexibility for the team to add tests as needed.

The lambdas run in parallel with each other with the results of the tests outputted to the **Done** lambda function which processes the results. The **End** state completes the workflow.

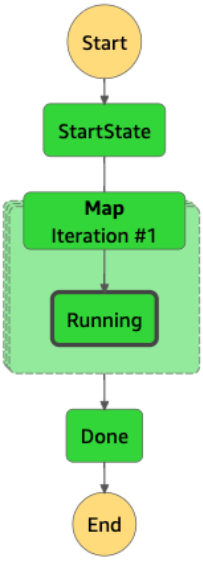


Figure 3.4 – Bosco State Machine

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 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.[2] 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 was based on the current number of Franenstein 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 began after finalising the proof of concept. The first step involved creating a Lightning page class which is the page the tests are running against. The class serves as the source for accessing the selectors and functions that have been added specifically for the Lightning page. Confining this functionality to a class reduces boilerplate code, a term for repetitious code. The tests were subsequently refactored to call the functions in the Lightning Page.

```
async getMessage(index) {
  const allMessages = await this.getAllMessages();
  return allMessages[index];
}
```

Figure 4.1 – Example of getMessage Lightning Page function

4.2 Tests

4.2.1 Context Test

The initial Frankenstein test that was migrated to Bosco was the Context test. This test establishes the context from the URL and verifies whether it is displayed along with the rest of the context in the messenger.

4.2.2 Conversation Engaged Test

The next test was 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.

4.2.3 Interaction Test

This is a test designed to check the functionality of a multi-select list node in a chatbot.

4.2.4 NLP Worker Lex V2 Intent Publish Test

The NLP Worker Lex V2 test uses a secret. 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.

A **Lex** worker is a remotely managed NLP worker which 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 remotely 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 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 in JSON format.

This test creates a long-living bot which is a bot that is created the first time the test is run and then reused for every test. The bot has a variable intent which is updated with a timestamp every time to ensure the bot is publishing successfully. When the bot is publishing its status is polled every 5 seconds until it succeeds and then the test starts.

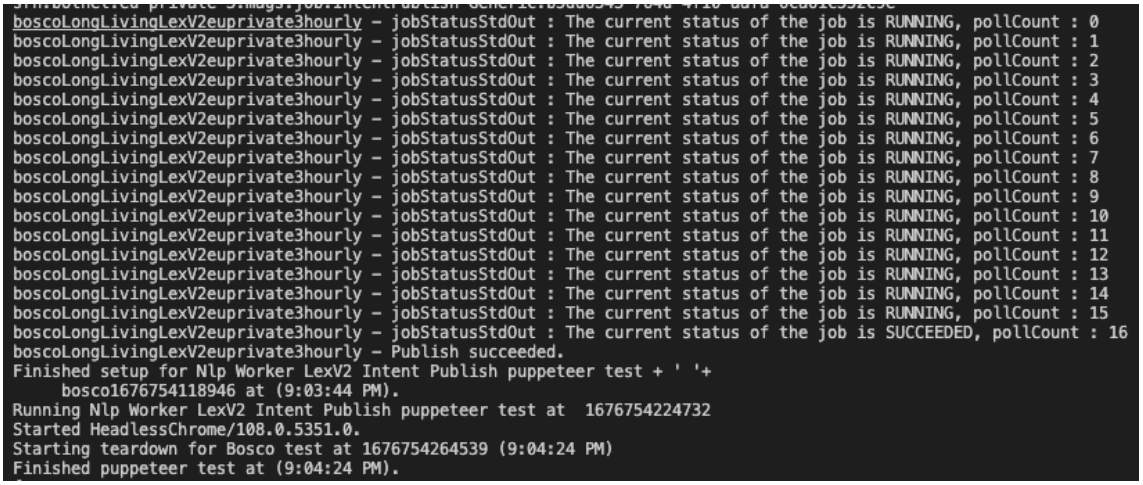


Figure 4.2 – NLP worker test with polling

4.2.5 CLI Test: Content

Frankenstein also has a number of tests that test the functionality of the ServisBOT 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 successfully.

The CLI test chosen was the Content, create-describe-update-delete-list test. It was a basic test that tests a Content node by creating a content node object and then performing an update, describe and delete on it.

When put to the team 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, the UI that the users would be using first hand. CLI tests are basically only for developers and there is 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. The CLI test was removed from Bosco.

4.3 CloudFormation

When deploying Bosco, a CloudFormation template in the form of YAML is run. CloudFormation is an AWS service that allows developers to define and deploy infrastructure as code. The Cloudformation template, which can be in either YAML or JSON format, is used to define and provision the AWS resources. When the CloudFormation template is uploaded to AWS it creates a CloudFormation stack which is a collection of AWS resources that can be managed as one unit. The template specifies the necessary environment variables, parameters, conditionals and other resources such as IAM roles and policies.

CloudFormation saves time and reduces errors as the same resources can be created multiple times once the template is defined. A YAML formatting extension from Redhat to format the YAML file was used for Bosco as it is not possible to deploy the YAML unless the formatting is accurate.

Bosco's CloudFormation Template consists of the following resources:

LambdaRole : IAM (Identity and Access Management) role in AWS that is used to grant permissions to AWS Lambda functions.

ScheduledEventIAMRole : IAM role used to grant permissions to the cron in Cloudwatch Events that will run the state machine.

BoscoIAMManagedPolicy : AWS managed policy that defines the permissions for the roles and users attached to the IAM role.

ApiLambda : The Lambda function that serves as an API endpoint for retrieving test run results from DynmaoDB.

TestOrchestrationLambda : The Lambda function that configures the input to the state machine based on the profile provided in the event.

TestRunnerLambda : The Lambda function that runs the suite of tests and returns the results.

TestResultsLambda : The Lambda function that processes, stores and logs the tests results.

StateMachine : The AWS Step Function State Machine that is used to manage the orchestrator, runner and results Lambdas in a workflow based on a map inline state.

ApiGateway : The API Gateway created to access the DynamoDB results table.

ResultsDynamoTable : The DynamoDB table that the results are written to.

S3BucketScreenshots : The S3 bucket that screenshots of failed tests are stored in.

4.4 Test Profiles

One significant way in which Bosco differs to Frankenstein is in the capacity 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 which stores all the profiles defined by the ServisBOT region and the adapters Venus and Mercury in the Dev environment.

```
{
  "profile": "eu-private-3-venus",
  "organization": "bosco",
  "sbRegion": "eu-private-3",
  "awsRegion": "eu-west-1",
  "messengerUrl": "some-url",
  "testConfig": {
    "..."
  },
  "testSuite": [
    "..."
  ]
}
```

Figure 4.3 – Sample Profile: eu-private-3-venus

Each profile contains the testSuite 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 event inputted to the orchestrator and output these to the handler.

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

Figure 4.4 – Parsing a JSON string to a javascript object and use of the spreader (...) operator

4.4.1 Singleton Class: TestRunnerConfig

It was a requirement for Bosco that the test configuration 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.

A singleton class was created to overcome this. A singleton 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 runner event 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 class 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 4.5 – Replacing proccess.env with getters to an Environment class invoked by a singleton class, TestRunnerConfig

4.5 Environment Variables

The environment variables are made up of the credentials needed to log into the ServisBOT CLI, the Log Level and the SSM parameters for the Lex V2 test which uses an external service. Storing the log level in the environment variables allows for consistency across the application, better security, easier debugging and flexibility. The log level can be changed without changing the code.

Initially, the environment variables were passed to the Orchestrator through the event. Once this was proven to work, the environment variables became part of the profile. However, a more secure option was neccessary so they were subsequently removed from the profile and stored in AWS Systems Manager Parameter Store, an AWS service for storage, configuration, secrets and data management. More information on SSM parameter store in Bosco is provided at a later stage in this report.

Two more classes were constructed in addition to the Environment class in order to resolve the environment variables.

EnvironmentResolver: A class with three functions: resolveEnvironment; resolveEnvironmentVariable; retrieveSSMParam

resolveEnvironmentVariables: A class which takes in the test profile and uses this to return an array of environment variables.

```
"environment": [
  {
    "name": "SB_CLI_CREDENTIALS",
    "value": {
      "username": "some-username",
      "password": "some-password"
    }
  },
  {
    "name": "LEX_V2_CROSS_ACCOUNT_ROLE_SECRET",
    "value": {
      "externalId": "some-externalId",
      "crossAccountRole": "some-cross-account-role"
    }
  },
  {
    "name": "LOG_LEVEL",
    "value": "INFO"
  }
],
```

Figure 4.6 – Environment Variables Inputted to Handler

The environment variables are provided to the handler at the same level as the testSuite and the testProfileConfig. By fetching and resolving the environment variables in the Orchestrator, this allows for a shared environment between the tests rather than providing the environment to each test which would be inefficient and resource heavy. Environments can grow and so this allows for flexibility in the development of Bosco.

It was essential to prove it was possible to flip between environments by either providing the environment to the Orchestrator Lambda or by inputting it to the State machine as an event. This was important to prove because providing the state machine with the environment variables and testSuite configuration there is more control over the state machine. What tests are run with what environment variables can be determined at any stage.

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:

```
{
  "ItemsPath":"$.testSuite",
  "ItemSelector": {
    "testSuite.$": "$$.Map.Item.Value",
    "environment.$": "$.environment",
    "testProfileConfig.$": "$.testProfileConfig",
    "executionId.$": "$$.Execution.Id"
  },
}
```

Figure 4.7 – State Machine Definition passing the environment to the Handler

The input to the Handler became the following:

```
{
  "testProfileConfig": {
    "..."
  },
  "executionId": "...",
  "environment": [
    "..."
  ],
  "testSuite": {
    "tests": [
      {
        "path": "goals/puppeteer-conversation-engaged.js"
      }
    ]
  }
}
```

Figure 4.8 – Input to one Iteration of Map State with just the Test Path

4.6 Endpoint Proxy

Bosco needed to support both Mercury and Venus engagement adapters similar to Frankenstein. This was 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.

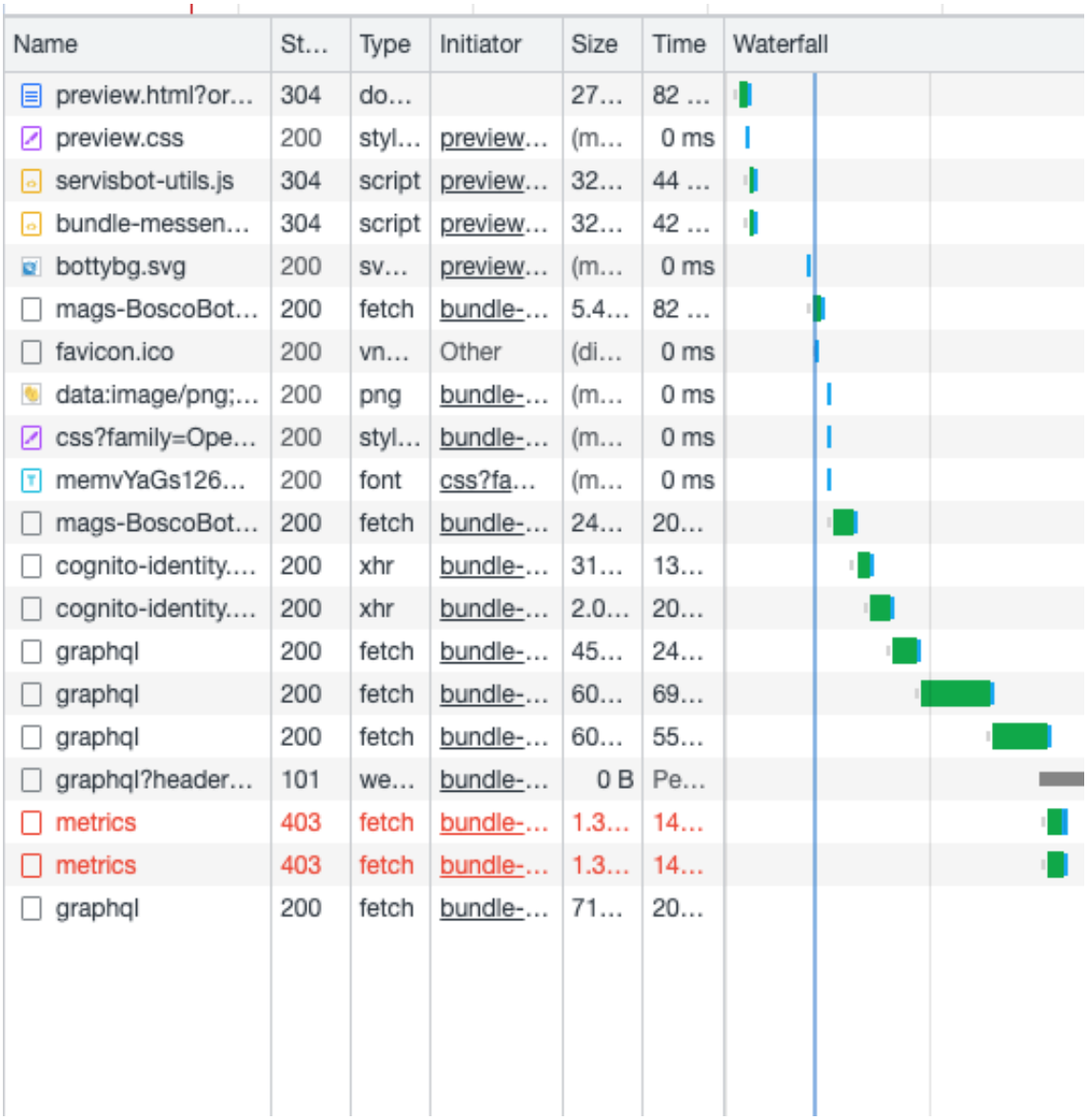
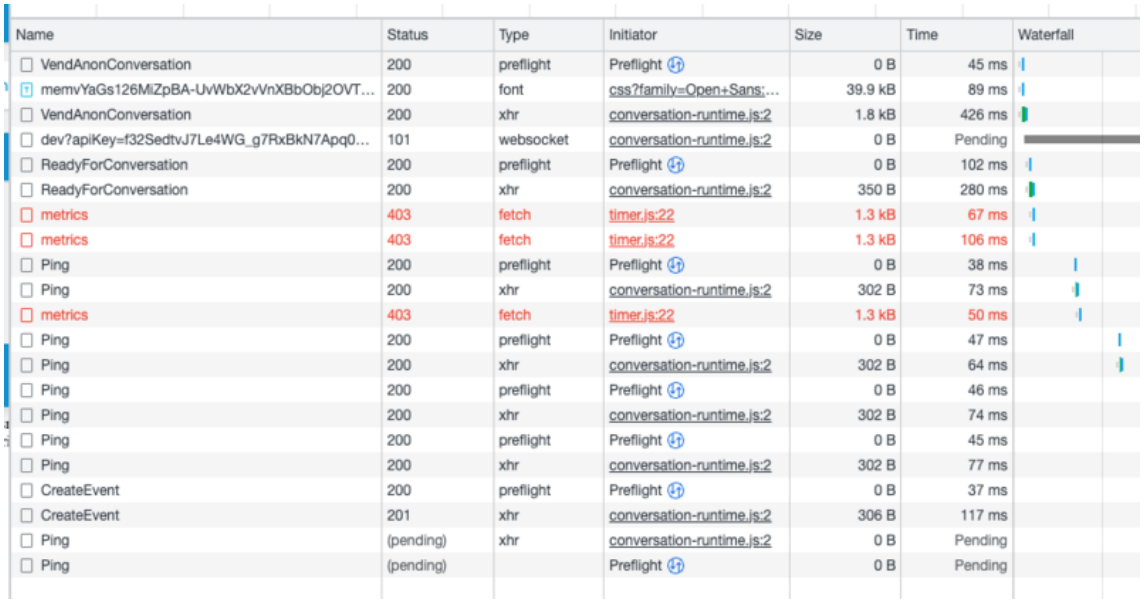


Figure 4.9 – Mercury Network Calls



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 4.10 – Venus Network Calls

4.7 SSM Parameters

In order for Bosco to build the environment in the orchestrator, SSM parameter store was utilised. *SSM (Systems Manager) is a service provided by AWS that allows you to securely store and retrieve data for your application.*[4]. 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 AWS SDK (Software Development Package) **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 environment object and returning this in 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.
- The README was updated to reflect the changes for future development from the ServisBOT team.

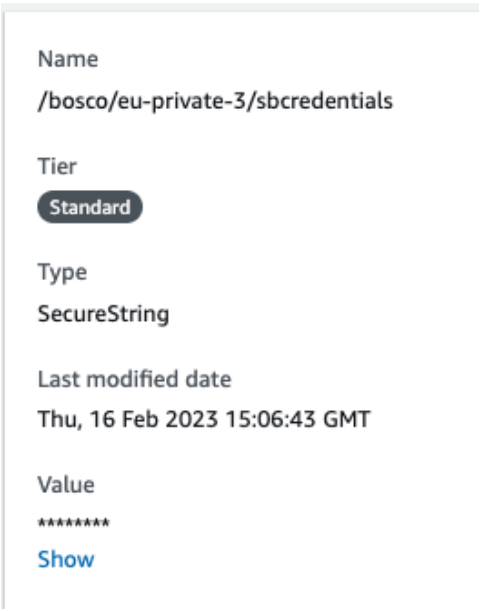


Figure 4.11 – SSM Parameters created in AWS Systems Manager

4.8 Eventbridge Rules (CRON)

Like Frankenstein, Bosco needs to run on a scheduled event. AWS Eventbridge allows developers to trigger events 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 updated.

- Scheduled Rules were added as resources to the CloudFormation template.

```
  MercuryProfileScheduledRule:
    Type: AWS::Events::Rule
    Properties:
      Name: !Sub "${CoreName}-Mercury-Profile"
      State: "ENABLED"
      ScheduleExpression: ScheduleVenusExpression
      Targets:
        - Arn:
            Fn::ImportValue:
              Fn::Sub: "${Environment}-bosco-StateMachineArn-${AWS::Region}"
          Id: !Sub "${CoreName}-Mercury"
          Input: !Sub '{"profile": "${ServisBotRegion}-mercury"}'
          RoleArn:
            Fn::ImportValue:
              Fn::Sub:
↪   ${Environment}-bosco-ScheduledEventIAMRoleArn-${AWS::Region}
```

- 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" ]
```

- 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. The following table displays each execution, which is given a unique ID and runs at regular intervals past the hour.

The working CRON instigated state machine was showcased to the team and they were queried on whether they would like the introdcution of a profile that runs 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-4087-5cf3-50f3cccccffa6_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 4.12 – State Machine Executions on a CRON

4.9 DynamoDB

Bosco stores the latest test results of each profile in a global DynamoDB table. Each profile has a row in the table with details on the results of the latest test run. The attributes stored in the dynamo table are the duration, the end time and the status of the test run. When the Bosco state machine is run, the test results are stored in this table overwriting the previous test run as the team are only interested in the latest result.

The process to completing this task was the following:

profile	duration	endTime	status
eu-private-3-mercury	20667	167965395...	passed
eu-private-3-venus	39432	167992591...	passed

Figure 4.13 – Dynamo DB Global Table Results

- Manually creating a dynamo table in dev and deciding the attributes that would be written to it, namely **profile** would be the primary key.
- The handler was then refactored in order to pass the profile, the endTime and the duration to the results lambda. The status was already being passed through.
- 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 is possible to choose what profiles to deploy the table in.

Chapter 5

Deployment

The deployment process was undertaken with help from the team as the initial plan needed to be altered. The CRON triggers which were initially in the CloudFormation template were removed and instead were put in a script specific to the AWS Region they would be deployed in which allowed for more control over what tests to run in what region.

5.1 CICD Process

CI/CD falls under DevOps (the joining of development and operations teams) and combines the practices of continuous integration and continuous delivery. CI/CD automates much or all of the manual human intervention traditionally needed to get new code from a commit into production, encompassing the build, test (including integration tests, unit tests, and regression tests), and deploy phases, as well as infrastructure provisioning. With a CI/CD pipeline, development teams can make changes to code that are then automatically tested and pushed out for delivery and deployment.[1]

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

- The developer puts in a PR (pull request) with the changes they made to the project code.
- A second developer reviews the code and when content with the changes, 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 in testing.
- If successful Slack will report on staging releases channel, otherwise the error needs to be located and debugged.
- The developer tests the code changes on testing.
- When the changes are verified, the 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 the testing account again as a tagged release.
- The developer tests on testing again.
- Once verified, the developer requests a deploy to production/testing from deploybot on Slack.

- Operations trigger the deployment, the CICD process is initiated once more with the project being deployed on production/upper.
- The 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 trigger was created in cloud formation 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.
- 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.

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 uploads each screenshot in the directory 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 only deployed in the testing environment but production and dev will all use this bucket. This is to save multiple buckets being deployed unnecessarily.

#	@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

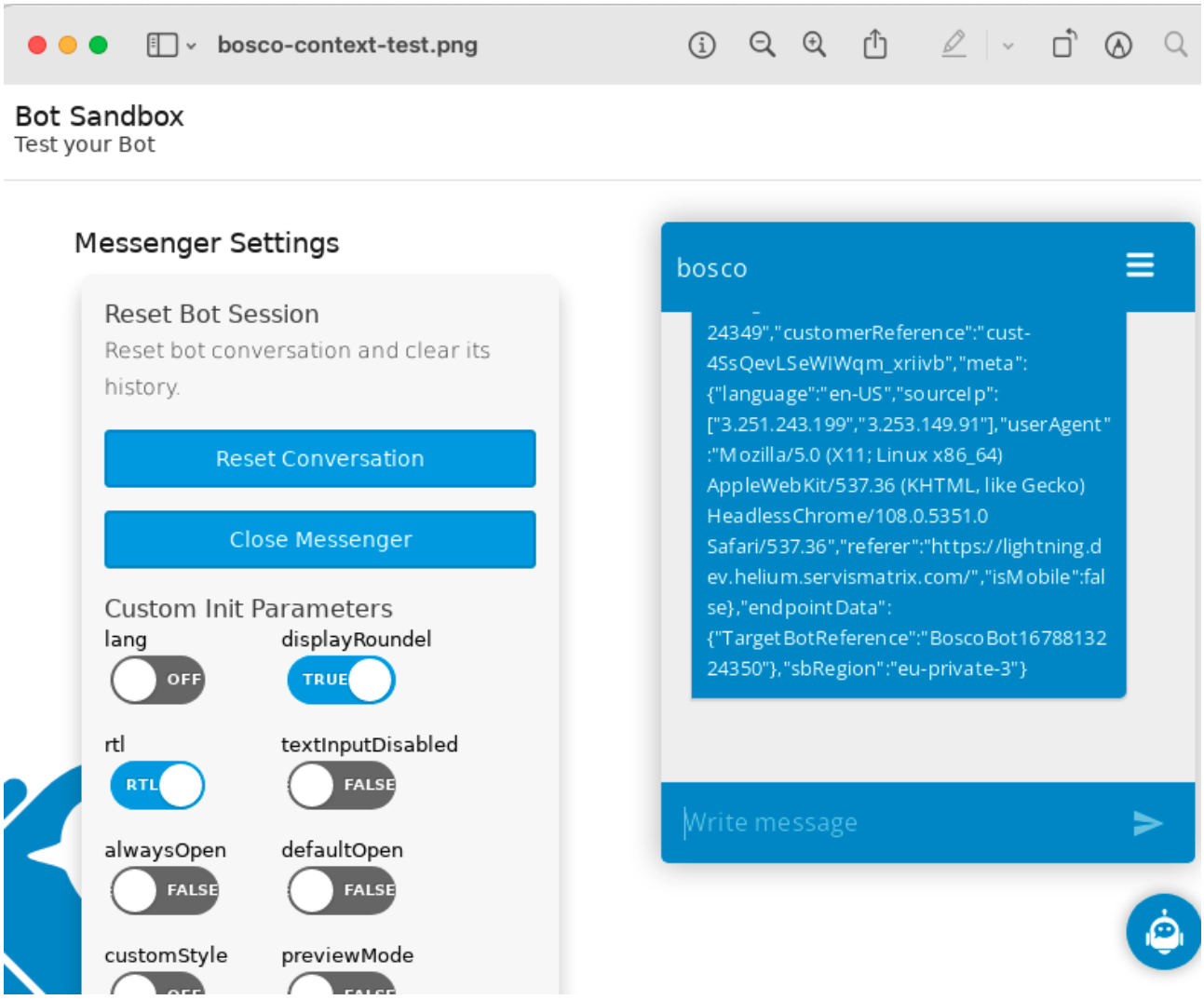


Figure 5.4 – Screenshot of failed test

Chapter 6

Conclusion

6.1 Reflection

Bosco has proven to be a valuable addition to the ServisBOT testing process, streamlining the testing process and providing faster, more efficient results. The project was developed to address limitations of the previous testing framework, Frankenstein which it has successfully achieved. The use of Puppeteer in Bosco allowed for more accurate and realistic testing of the ServisBOT Portal UI, while also improving the speed of the tests.

Running the tests on Lambda instead of EC2 provided several advantages for the Bosco project. One of the most significant improvements was the reduced cost of running the tests. While the exact cost at this time cannot be determined, there is strong indications that running tests on Lambda will be more cost-effective than using EC2 instances.

Another advantage of using Lambda was the increased scalability and flexibility which will allow the team to add tests and manage resources more efficiently. Additionally, Lambda can be easily integrated with other AWS services, such as S3 and CloudWatch, which provides a more streamlined and automated testing process.

One of the main reasons why running tests on Lambda was a vast improvement over EC2 was the faster and more reliable performance. Lambda functions can be invoked in milliseconds, which allows for faster test execution and improved overall test cycle time. Furthermore, Lambda provides high availability and fault tolerance, which means that tests are less likely to fail due to infrastructure issues or downtime.

In conclusion the development of Bosco has been a valuable experience. It allowed me to explore and learn more about AWS Lambda, CloudFormation, and other related services. It was satisfying to see that the tests ran smoothly on the Lambda function and that the cost was significantly reduced compared to EC2. Although the actual cost is still uncertain, we can be confident that Lambda is more cost-effective. The Bosco test runner has been integrated with the development workflow, and its effectiveness in testing the ServisBOT Portal UI has been demonstrated. Overall, I am proud to have contributed to the ServisBOT team's efforts in improving the quality and reliability of their platform through Bosco.

6.2 Key skills

- Sprint planning.

- Amazon Web services, in particular Lambda, Step Functions, CloudFormation, 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. The following is a list of some of the challenges I did face however.

- 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.
- The complexity of the project was incredibly challenging. Although I was given time to move at my own pace, trying to understand all the different components of Bosco was often overwhelming.
- Learning how to use LateX for the project report initially was very confusing and stunted the process. It was not a requirement but a challenge I set myself.

6.4 Future Work

6.4.1 Migrating the Remainder of the Tests

Incredibly at the end of the project, Bosco is at a state where the rest of the team can now add the remainder of the tests. There is on average sixty tests that are yet to be transitioned from Frankenstein to Bosco which any team member once lightly versed in Puppeteer and Mocha can work on easily. Bosco is running in production and testing the few tests that were written during this project. The tests in Frankenstein will be turned off one by one as they are migrated to Bosco.

6.4.2 API

Currently an API for Bosco is being developed which will retrieve the latest test results for the DynamoDB table. Tests have been developed to health check the routes of this API.

6.4.3 Alert of Test Failures

Alerts that are reported to the Slack channel are yet to be developed.

Bosco will continue to be a work in progress for the ServisBOT team. The tasks related to Bosco will be migrated to the Core team's planning board as the enrire team will take over the development of Bosco.

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

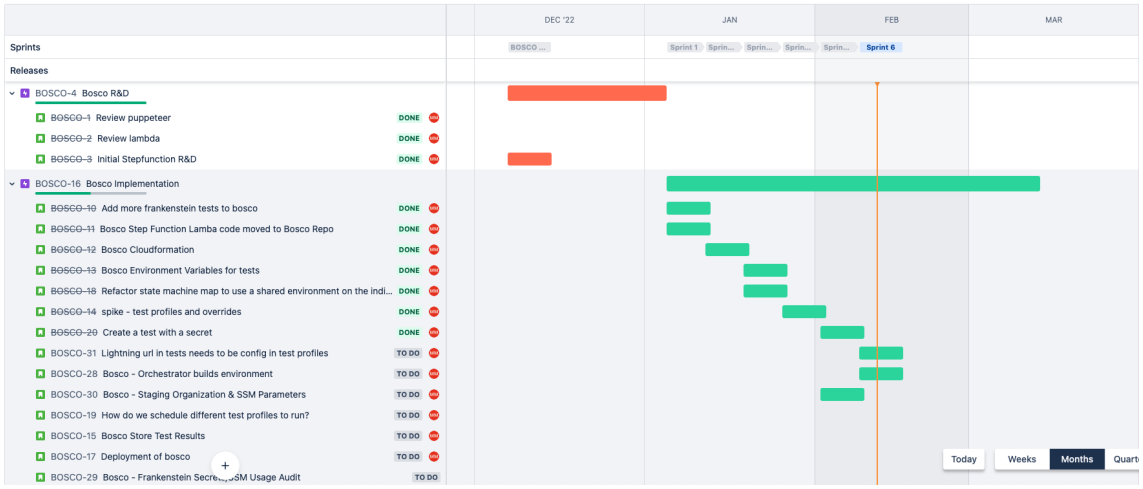


Figure A.1 – R&D Phase and Bosco Implementation Phase

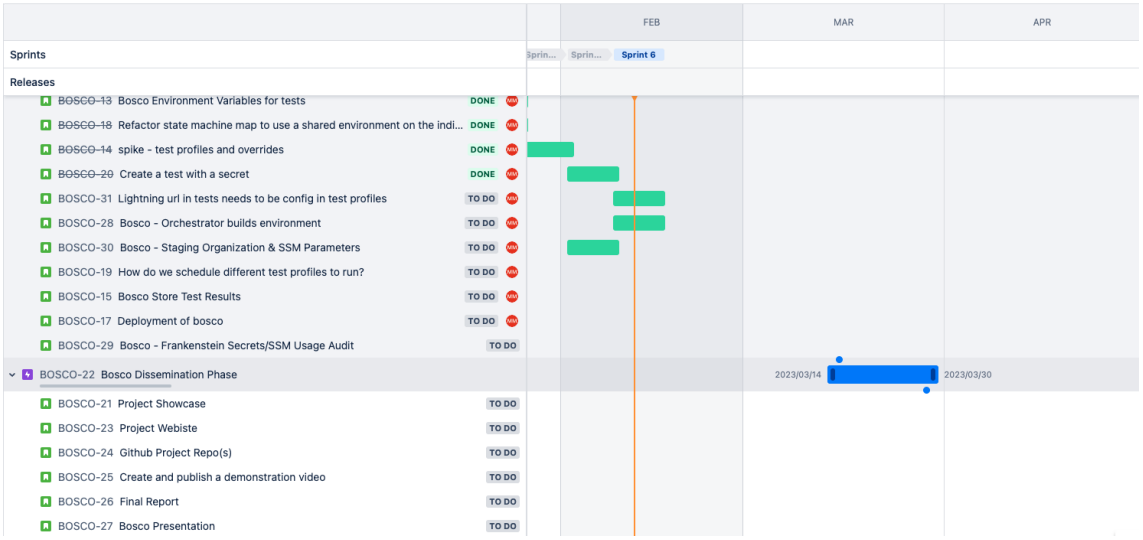


Figure A.2 – Bosco Dissemination Phase

Appendix B

Dependencies and Dev-Dependencies

dependencies

@aws-sdk/client-dynamodb @aws-sdk/lib-dynamodb @servisbot/servisbot-cli bluebird cors csv-parse express loglevel was added to replace the console.logs. **luxon number-to-words-en puppeteer serverless-http sinon supertest**

dev-dependencies

aws-sdk depcheck: a dependency check package which checks for unused dependencies before commits. **eslint**: a linting tool that ensures all the code is formatted consistently by anyone who makes changes to the code. **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. **mocha pre-commit**

Figure B.1 – dependencies and dev dependencies used for Bosco

References

[1] about.gitlab.com. (n.d.) *What is CI/CD?* 2023 (cit. on p. 27).

[2] Inc Amazon Web Services. *AWS Step Functions Pricing — Serverless Microservice Orchestration — Amazon Web Services*. 2023 (cit. on p. 14).

[3] 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 (cit. on p. 9).

[4] Evan Halley. *Retrieving AWS SSM Parameters with Node*. 2021. URL: https://evanhalley.dev/post/aws-ssm-node (visited on 02/12/2023) (cit. on p. 23).

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

[6] Mochajs.org. *Mocha - the fun, simple, flexible JavaScript test framework*. 2019 (cit. on p. 6).

[7] npm. *Puppeteer*. 2023 (cit. on p. 3).

[8] C Richardson. *Microservices*. 2017 (cit. on p. 1).

Appendix C

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