



Waterford Institute *of* Technology

INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

PROJECT

HIGHER DIPLOMA IN SCIENCE

IN COMPUTER SCIENCE (NFQ – LEVEL 8)

2019-2021

**Chat Generator Test Harness with Automated
Deployment on Docker**

FINAL REPORT

Name: Dermot Sheerin

Student Number: 20086620

Supervisor: Eamonn de Leastar

Abstract

The problem this project is attempting to solve is to build, test and deploy a performance test environment capable of simulating customer chat interactions with a Mock Contact Center (CC). The test environment developed and discussed in this project consists of the following applications:

- Chat Generator back-end (nodeJS)
 - Establish a chat interaction via REST requests to a Mock CC
 - Exchange a predefined number of chat interactions with a Mock CC agent
 - Terminate the chat interaction
- User Interface (React JS) to perform the following:
 - Retrieve default chat parameter settings
 - Ability to update test parameters before each test run
 - Display live traffic statistics throughout the performance test run identifying a pass or fail for each sequence in the chat flow.
 - Display current Memory, User and System CPU time resource usage for the chat generator
 - Report peak Memory, CPU reached throughout the test run
 - Live Graph of current Memory and CPU utilization with timestamp
- Continuous Integration /Continuous Deployment (CI/CD) using Jenkins pipeline to automate the build and deployment of back-end (NodeJS) and front-end (React) applications into docker containers each time a change is pushed to a GitHub repository.

The CI/CD environment promotes iterative development, whether a new feature is added to the test environment or a new technology is introduced to achieve a better throughput, the CI/CD pipeline will help build, test and deploy software faster.

The chat generator is a modular design, effectively allowing different technologies to be swapped in or out at relative ease. An example of this is discussed in this document where I assess the performance of Express Web Framework versus Fastify Web Framework.



Keywords

NodeJS; React; ReCharts; React Hooks; Custom Hooks; Express Framework; Fastify Framework; SPA; SocketIO; Webhook; Docker; Docker Containers; Jenkins; CI/CD

Declaration of authenticity

I declare that the work that follows is my own, and that any quotations from any sources (e.g. books, journals, the internet) are clearly identified as such by the use of 'single quotation marks', for shorter excerpt and identified italics for longer quotations. All quotations and paraphrases are accompanied by (date, author) in the text and a fuller citation is the bibliography. I have not submitted the work represented in this report in any other course of study leading to an academic award.

Signed: _____

Statement of Copyright

The author and Waterford Institute of Technology retain copyright of this project. Ideas contained in this project remain the intellectual property of the author except where explicitly referenced otherwise. All rights reserved. The use of any part of this document reproduced, transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise stored in a retrieval system without the prior written consent of the author and Waterford Institute of Technology is not permitted.

Table of Contents

Abstract.....	i
Keywords.....	i
List of Figures	vii
1 Introduction	1
1.1 Background to the problem	1
1.2 Description of Project Aims	1
2 Tools and technologies	2
3 System Analysis and Design	3
3.1 User Stories.....	3
4 Methodology.....	4
4.1 Sprint 1 Deliverables	4
4.2 Sprint 2 Deliverables	4
4.3 Sprint 3 Deliverables	4
4.4 Sprint 4 Deliverables	4
4.5 Sprint 5 Deliverables	4
4.6 Sprint 6 Deliverables	4
4.7 Sprint 7 Deliverables	5
4.8 Sprint 8 Deliverables	5
4.9 Sprint 9 Deliverables	5
4.10 Sprint 10 Deliverables	5
4.11 Sprint 11 Deliverables	5
4.12 Sprint 12 Deliverables	5
4.13 Sprint 13 Deliverables	5
4.14 Sprint 14 Deliverables	5
4.15 Sprint 15 Deliverables	5
4.16 Sprint 16 Deliverables	5
4.17 Sprint 17 Deliverables	5
4.18 Sprint 18 Deliverables	5
4.19 Sprint 19 Deliverables	6
4.20 Sprint 20 Deliverables	6
4.21 Sprint 21 Deliverables	6
4.22 Sprint 22 Deliverables	6
5 Design and Implementation.....	7

5.1	Back End Development – NodeJS Chat Generator	8
5.1.1	Chat Interaction Flow.....	8
5.1.2	Process Usage	9
5.1.3	SocketIO	10
5.1.4	Reset Statistics	11
5.1.5	Express / Fastify Web Framework	11
5.2	Front End Development – React JS.....	12
5.2.1	API Component	12
5.2.2	ChatParameters Component	12
5.2.3	ChatStats Component	13
5.2.4	Resource Chart Component.....	15
5.2.5	Websocket Component - SocketIO Client.....	16
5.3	Automate Test environment using Jenkins.....	17
5.3.1	Jenkinsfile (Declarative Pipeline) from Source Control (SCM).....	18
5.3.2	Dockerfile	20
5.3.3	Docker-compose	20
6	Sample Test run “Express V Fastify” comparison	23
6.1	Modular development of NodeJS Chat Generator	24
6.2	20k chat generator test using EXPRESS	26
6.3	20k chat generator test using FASTIFY.....	29
6.4	Comparison Summary.....	37
7	Problems Encountered	38
7.1	Displaying Realtime chat statistics in an efficient manner	38
7.2	Upgrading Jenkins container.....	38
7.3	High socket count open to Fastify during performance test	39
7.4	Jenkins Deployment.....	39
7.5	Jenkins Build Triggers.....	39
8	Reflection	40
8.1	Project development path	40
8.2	Independent Learning.....	41
8.3	Future work.....	42
9	References	43
9.1	Jenkins / Docker	43
9.2	Chat Generator	43

9.3	React	43
10	Appendix	44
10.1	Useful Docker Commands:.....	44
10.1.1	Copy files from ubuntu server to a Docker container running on ubuntu.	44
10.1.2	Monitor the docker logs live using ‘-f’ option.....	44
10.1.3	Restart a container.....	44
10.1.4	Execute a command inside a Docker Container from the ubuntu server CLI.....	44
10.1.5	Access the bash CLI of a Docker Container	44
10.1.6	Display what is being outputted from inside docker container.....	44
10.1.7	Remove docker image.....	44
10.1.8	List all Docker containers	44
10.1.9	Stop Docker Container	44
10.1.10	Remove Docker container.....	44
10.2	Pipeline deployment files.....	45
10.2.1	Jenkinsfile.....	45
10.2.2	Dockerfile – Chat Generator	45
10.2.3	Dockerfile - React Front End	46
10.2.4	Docker-compose – Chat Generator	46
	Docker-compose – React Front End.....	46

List of Figures

Figure 1 Environment Overview	7
Figure 2 CI/CD Development Test Harness Overview.....	7
Figure 3 Chat Generator Parameter settings.....	12
Figure 4 Resource Usage Statistics	13
Figure 5 Chat Generator Test Statistics	14
Figure 6 Jenkins Declarative Pipeline from SCM.....	18
Figure 7 Jenkins Build Triggers.....	19
Figure 8 Jenkins Dashboard	19
Figure 9 Chat Generator Pipeline.....	20
Figure 10 React Front End Pipeline	21
Figure 11 Successful Pipeline Build Log	22
Figure 12 Sample console output from Mock Server	23
Figure 13 Sample console output from Chat Generator.....	24
Figure 14 How to switch between Express and Fastify.....	24
Figure 15 NodeJS Modular File structure.....	25
Figure 16 Express parameter settings for 20k test	26
Figure 17 Express idle resource usage	26
Figure 18 Express chat statistics at 5k interactions	27
Figure 19 Express chat statistics at 20k interactions	28
Figure 20 Fastify parameter settings for 20k test.....	29
Figure 21 Fastify idle resource usage.....	29
Figure 22 Fastify failures at 5k interactions	30
Figure 23 Open port connections for Fastify test	31
Figure 24 Open port connections for Express test	32
Figure 25 Open port connections for Fastify after resolution implemented	34
Figure 26 Fastify chat statistics at 17k interactions.....	35
Figure 27 Fastify chat statistics at 20k interactions.....	36

1 Introduction

1.1 Background to the problem

A test framework team has been tasked with identifying a chat generator tool capable of simulating customer chat interactions with Contact Center (CC) agents. The project I have selected is to develop a test generator capable of simulating up to 20k customer chat interactions using nodeJS. As part of this proof of concept I will perform a number of tests using two Web Frameworks

- Express Web Application Framework
- Fastify Web Application Framework

I will integrate the above two technologies with nodeJS and perform several tests against a Mock Call Center Server. All tests will be performed against the same mock server.

To complete the full stack, a front-end UI will be necessary to start and monitor each test run.

1.2 Description of Project Aims

The Mock CC is not covered as part of this document, it was designed by a separate engineering team to simulate a Contact Center back end. The chat generator discussed in this document will first establish a client session with the CC, this will be achieved via a number of API requests where the generator will seek and retrieve session and engagement IDs necessary to establish a valid client session. Once the session is established the generator will send a chat message and wait for the CC agent to respond. After several chat message exchanges the chat generator will terminate the call.

I decided to use React JS as the front end for the chat generator, I found it to be a very powerful tool for building single page applications (SPA). I really enjoyed the React experience during our ICT Skills module and not only does it suit my project needs but I felt this would be an ideal way to refresh and expand my knowledge in this area. The UI will be responsible for setting the test parameters, starting/stopping each test run, and providing live test statistics and resource usage of the chat generator.

To complete the test environment, I decided to build a Continuous Integration /Continuous Deployment (CI/CD) pipeline using Jenkins. The pipeline will automate the build and deployment of the chat generator and React applications into docker containers each time a change is pushed to a GitHub repository. This will help to build, test, and deploy the test applications in an efficient manner.

2 Tools and technologies

- NodeJS Client Chat Generator simulating customer chat interactions
- Express framework with routes to handle various incoming API requests from the SUT
- Fastify framework with routes to handle various incoming API requests from the SUT
- React UI to apply Chat Generator parameters, start/stop test, reset chat statistics, display Realtime Statistics and Resource Usage
- Ubuntu Servers to host the nodeJS client and mock CC server for large scale performance run
- Ubuntu Server to host Docker deployments for test and development purposes
- Jenkins Docker container for automated build and deployment of React and Chat Generator applications

Equipment/Software	Version/Release
NodeJS Chat Generator (Backend)	v12.14.1
Express	V4.17.1
Fastify Fastify-cors Fastify-socket-io	V3.15.1 V6.0.0 V2.0.0
Socket IO	V3.1.2
OS-Utills	V0.0.14
React (Front End)	v16.13.1
Recharts	V2.0.8
Socket IO Client	V3.1.2
Jenkins (running in Docker)	V2.281
Docker	v19.03.8
Ubuntu Server	4G Memory Dual Intel(R) Xeon(R) CPU E5-2690 0 @ 2.90GHz Processor

3 System Analysis and Design

The following user stories outline at a high level, the areas the end user requires to solve the problem outlined in **Section 1**.

3.1 User Stories

Front End – React JS (Sprints 13 - 21)

- User should be able to retrieve and update existing configuration parameters, at a minimum the following:
 - Specify number of chat interactions to simulate
 - Number of chat message exchanges must be configurable
 - Times for sending/responding to chat interactions must be configurable
 - In the event an agent does not answer the interaction in time the generator must be able to handle this gracefully
- User must be able start/stop performance test run
- Resetting previous chat run statistics before each run is necessary
- Details of each chat interaction should be viewable, broken down into various stages of the interaction flow if possible
- High level view of what areas of the interaction flow passed or failed
- Other interaction information such as the time the first and last chat interaction sessions were created, number of events received from Mock server
- View Resource usage of the chat generator, at a minimum:
 - Memory and processor consumption throughout the test
 - Need to be able view this resource usage over the entire test to identify any patterns that lead to failures

Back End – NodeJS (Sprints 1, 6-12, 22)

- Generator will establish and maintain communication with Mock CC server using supported REST APIs
- Resource usage should be tracked and passed to the Front End using a reliable real-time bi-directional technology
- Chat statistics information should be tracked and passed to the Front End using a reliable real-time bi-directional technology
- Identify a mechanism to track expected events from the Mock server, any events that are not received within the expected time frame should be marked as a failed interaction. The chat generator must recover and continue in the event of failures.
- To assist future development, the back-end code should be modular, allowing future development teams to easily replace existing code/technology for another e.g., the web framework used as part of the chat generator

Development Environment / Test Harness / CI-CD Pipeline (Sprints 2-5)

- Regular tested features must be delivered to adhere to the agile project management methodology
- Automation of the build, test and deployment process is required. Identify a technology to achieve a CI/CD delivery pipeline

4 Methodology

I decided to use an Agile project management approach for several reasons

- I knew my project was liable to change
- I was uncertain as to what my end solution would encompass from both a technology and a feature point of view
- I was focused on delivering working features at regular intervals throughout my development cycle

4.1 Sprint 1 Deliverables

- Deploy Ubuntu Server for test environment
- Identify APIs required for Chat Generator

4.2 Sprint 2 Deliverables

- Investigate technology to automate the build and deployment of Chat Generator Back/Front End test environment
- Chosen technology – Docker for container deployment and Jenkins to automate build and deployment of application containers

4.3 Sprint 3 Deliverables

- Deploy Jenkins Container on Docker (running on Ubuntu Server)

4.4 Sprint 4 Deliverables

- Configure Dockerfile and docker-compose files to build and deploy nodeJS back end image into Docker container
- Add Jenkins pipeline job to automate build and deployment of back end server on push to Git repository

4.5 Sprint 5 Deliverables

- Configure Dockerfile and docker-compose files to build and deploy React front end image onto Docker container
- Add Jenkins pipeline job to automate build and deployment of front end on push to Git repository

4.6 Sprint 6 Deliverables

- Implement Create Session and Engagement APIs
- Implement logic to utilise Create Session and Engagement APIs

4.7 Sprint 7 Deliverables

- Implement Create Webhook and Send Chat APIs
- Implement logic to utilise Create Session and Engagement APIs

4.8 Sprint 8 Deliverables

- Investigate and Implement Express Web Framework to handle incoming Mock Server Requests
- Investigate logic on how to handle wait for agent to answer interaction / reply to chat message

4.9 Sprint 9 Deliverables

- Back end Chat Statistics Object

4.10 Sprint 10 Deliverables

- Deploy Chat Generator on ubuntu
- Attempt basic chat interaction to Mock Server

4.11 Sprint 11 Deliverables

- Investigate and Implement Fastify Web Framework to handle Mock Server Requests

4.12 Sprint 12 Deliverables

- Implement Chat parameter/timer variables that can be managed via Front End

4.13 Sprint 13 Deliverables

- Create React App
- Implement React API component to handle Chat Parameter retrieval and modification
- Implement React chatParameters component

4.14 Sprint 14 Deliverables

- Implement React Chat Stats API
- Implement React Chat Statistics Component

4.15 Sprint 15 Deliverables

- Implement React Pages and Header components

4.16 Sprint 16 Deliverables

- Investigate Web socket technology for real-time bidirectional communication between Front End and Back End

4.17 Sprint 17 Deliverables

- Implement Socket IO on backend with Express Framework
- Implement Socket IO Client on Front End

4.18 Sprint 18 Deliverables

- Implement Socket IO on backend with Fastify Framework

4.19 Sprint 19 Deliverables

- Investigate and Implement Resource Monitoring mechanism for backend chat generator

4.20 Sprint 20 Deliverables

- Implement Resource Statistics component on Front End i.e., table to display current resources and max resources (in real time) during test run

4.21 Sprint 21 Deliverables

- Investigate and implement Front End technology to Graph real time resource statistics

4.22 Sprint 22 Deliverables

- Modularize back end code i.e., separate code into modular components i.e., APIs, Logger, Utilities, Routes

5 Design and Implementation

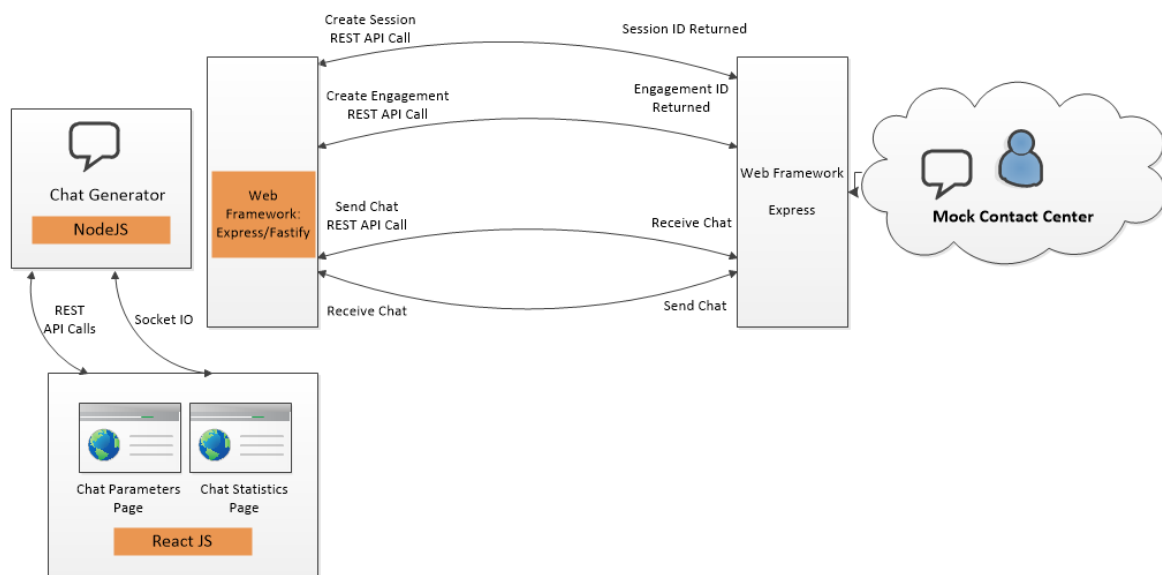


Figure 1 Environment Overview

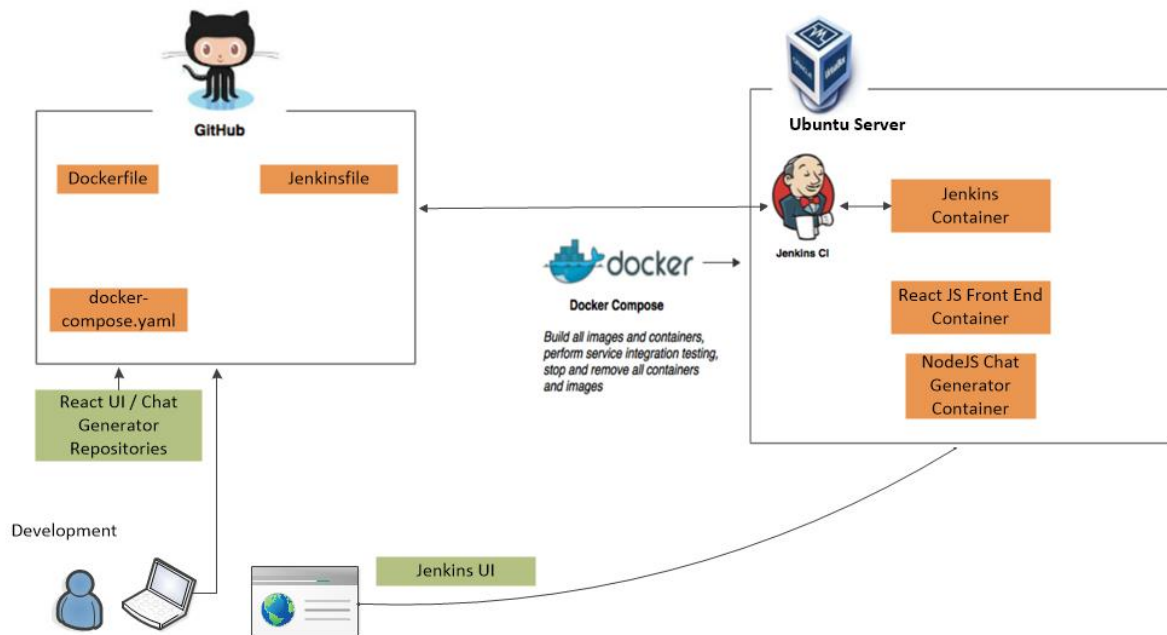


Figure 2 CI/CD Development Test Harness Overview

5.1 Back End Development – NodeJS Chat Generator

For Sprint 1 the focus was to identify the APIs required by the SUT to simulate a customer initialising a chat interaction to the Contact Center and exchanging several messages with the agent. A description of the chat interaction flow is provided below:

5.1.1 Chat Interaction Flow

- Client registers with Webhook: NodeJS sends API POST to register Express/Fastify Framework IP with webhook and detail of what events it wants to receive. All events defined in the POST will be directed to nodeJS to handle the requests. Routes are configured in Express/ Fastify to handle and direct events to appropriate functions in chat generator (Note this is an early implementation for a future feature on the Mock server, webhooks were not tested during this project).
- Client sends POST request to Session Create API:
requires several parameters to be passed in the POST body e.g., customer display name, agent account ID, preferred language to use.
This API call expects a Status 201 response to indicate the request was successful. The API also returns a Session ID for the call which is required for the Engagement API request.
If the request fails, the chatStats object is updated to record a failed create session attempt and the simulated customer transaction is stopped. Note that other chat interactions will continue to proceed.
If the request is successful, the chatStats object is updated to record a pass for the create session attempt.
- Client sends POST request to Engagement Create API, passing the Session ID created during the Session Create API call, along with a number of other parameters in the request body. This API call expects a Status 200 response to indicate the request was successful. The API returns an Engagement ID and a number of other parameters that will be used during the chat flow.
If the request fails, the chatStats object is updated to record a failed create Engagement attempt and the simulated customer transaction is stopped.
If the request is successful, the chatStats object is updated to record a pass for the create Engagement attempt.
- One critical item that is tracked as part of the performance metrics is that the Agent answers each chat interaction within an expected timeframe specified by the user at test start time i.e., 'agentJoinTimeout'. I had to find a way to 'pause' the chat generator flow until an Agent Join event (sent by Mock server) is received, to confirm that the agent did answer the interaction. If the agent fails to answer within this timeframe the chat interaction will end and the chatStats object is updated to record a failed Agent Join attempt.
In order to 'pause' the flow while waiting for an Agent to Join, I decided to create a promise for each Agent Join request and store this promise inside a Map, using the unique engagement ID as the key. When the promise is resolved/rejected, the entry in the Map will be updated with the result of the promise and the flow will continue/end.
I now had to consider how to handle events that are never received by the chat generator otherwise the interaction would remain at this point indefinitely.
To handle the above I created a promiseTimeout function. This function contains a Promise.race() method which will return a promise that fulfills or rejects as soon as one of the promises fulfills or rejects.
The promiseTimeout takes in two parameters:
- a **timeframe** in milliseconds i.e., agentJoinTimeout, this is the max number of milliseconds allowed for an agent to answer an incoming chat interaction before the loop is exited and an Agent Join failure is recorded

- a **Promise** i.e., Agent Join Promise mentioned above.

If the Agent answers before the agentJoinTimeout then the Agent Join Promise is retrieved from the map and resolved to true, and the next step in the flow will be invoked.

If the agent fails to answer before the agentJoinTimeout then the promiseTimeout will return a false and the interaction will end. The chatStats object is updated to record a failed Agent Join event and the loop will exit.

To summarise the promiseTimeout function: The Promise.race function in conjunction with setTimeout will return the result of a race between the agent answering and the agent join timeout value defined by the user.

- Once an Agent Join is successfully received, the Client sends a POST request to the SendChat API. A new 'Wait for Chat' Promise is created which stores the pending/unresolved result in local MAP, the promise will resolve on receipt of a chat message from the agent. This send chat message loop will continue for a predefined number of loops set by the user i.e., 'chatSendMax'.
- When the 'chatSendMax' value is reached the agent will send a ###BYE### to terminate the interaction.
- If it is necessary to initiate a terminate interaction from the agent/server then the client will handle this event inside the processAgentDisconnectEvent function.

5.1.2 Process Usage

Process is a global Node.js object which contains information about the current Node.js process. The heap is a memory segment used for storing objects, strings, and closures. To find out how much heap is used by chat generator (nodeJS) an in-built method 'memoryUsage()' is accessed which returns an object. Contained within this object is a field called 'heapUsed'. This is the value returned by the chat generator to identify the current memory heap allocated to the nodeJS chat generator during test.

The process.cpuUsage() method is an inbuilt application programming interface of the Process module which is used to get the user and system CPU time usage of the current process.

The above memory and CPU details are passed back to the front end using SocketIO.

5.1.3 SocketIO

Socket.IO is a JavaScript library for realtime web applications. It enables realtime, bi-directional communication between web clients and servers. It has two parts: a client-side library that will run within the React UI, and a server-side library for Node.js.

Socket.IO `on()` takes two arguments: the name of the event, in this case "connection", and a callback which will be executed after every connection event. The connection event returns a socket object which will be passed to the callback function. By using this socket object it is possible to listen for socket connections which will be established when the ChatStats page is accessed from React front end. When this occurs the chat statistics will send data back to React and the UI will be updated.

```
const io = socketIo(server, {
  cors: {
    origin: '*',
  }
});

// listen for socket connection, this will be invoked each time the ChatStats page is invoked from React Front End
io.on("connection", (socket) => {
  logMessage("New client connected");
  if (interval) {
    clearInterval(interval);
  }
  interval = setInterval(() => getChatStats(socket), 1000);

  // clearing the interval on any new connection, and on disconnection to avoid flooding the server
  socket.on("disconnect", () => {
    logMessage("Client disconnected");
    clearInterval(interval);
  });
});
```

The following statistics are emitted back to the front end via SocketIO. There is a function `getMaxValues` that is called to return the current max CPU and Memory resource usage. The 'graphData' object is structured in accordance with what the Rechart component expects to display the data correctly in the graph.

```
const getChatStats = socket => {
  // Emitting a new message. Will be consumed by the client
  const usedMem = utils.usedMem();
  const cpuTime = utils.cpuTime();

  maxValues = utils.getMaxValues(usedMem, cpuTime.userTime, cpuTime.systemTime);

  socket.emit("FromAPI",
    {
      chatStatsMap: index.chatStatsMap,
      eventCounter: eventCounter,
      testTime: { startTime: index.chatParameters.startTime, stopTime: index.chatParameters.stopTime },
      graphData: { time: utils.currentTime(), usedMem: usedMem, userTime: cpuTime.userTime, systemTime: cpuTime.systemTime },
      resourceStats: {
        usedMem: [`${usedMem} MB`, maxValues.maxMem],
        userTime: [`${cpuTime.userTime} secs`, maxValues.maxUserTime],
        systemTime: [`${cpuTime.systemTime} secs`, maxValues.maxSystemTime ]
      },
    },
  );
};
```

5.1.4 Reset Statistics

Within the Chat statistics UI page there is a field called 'Events Received' which will display the total number of requests received at the back-end framework. To avoid having to restart the back-end server to reset the chat statistics and event counter I created a small function to reset the statistics and event counter back to 0. This function will be invoked from the UI via a Reset button which will make a REST request to an endpoint '/resetStats' at the back-end framework.

```
const resetChatStats = () => {  
  // iterate chatStatsMap to retrieve the value (array) that contains the stats for pass/fail and reset  
  to 0  
  for (const [key, value] of Object.entries(chatStatsMap)) {  
    value.forEach((stat, index) => {  
      // reset each statistic to 0  
      value[index] = 0;  
    });  
  }  
  // reset event count to 0  
  resetEventCounter();  
}
```

5.1.5 Express / Fastify Web Framework

Both web frameworks will receive all events to a POST method route 'allEvents'. Within this route there is logic to determine the correct course of action for the request received. The two main requests the frameworks will handle are for agent join events which will trigger the 'processAgentJoinEvent' function to execute, and for agent send message events which will trigger the 'processAgentSendMsgEvent' function to execute.

Separate to the chat flow events received there are also routes that listen and respond to the following requests from the Front-End UI

- Retrieve Chat parameters (retrieve chat parameters when the chat parameters page is accessed)
- Change chat parameters
- Retrieve the current chat statistics
- Stop chat generator
- Start chat generator
- Reset Chat Statistics

5.2 Front End Development – React JS

The following is a high-level summary of the main components used within the UI.

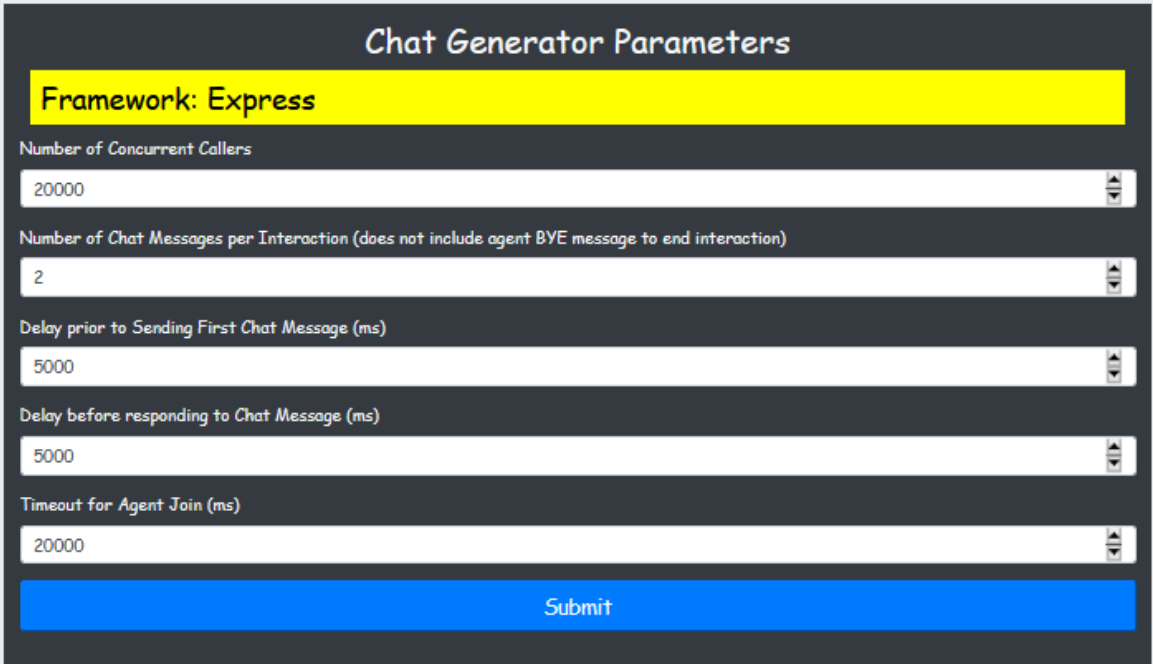
5.2.1 API Component

Contains the IP and Port to connect to chat generator for REST API calls. Some examples of the API calls are to:

- Start/Stop the test run
- Retrieve/Update parameters
- Reset Chat Statistics

5.2.2 ChatParameters Component

The chat parameters component utilises React-hook-form, useState and useEffect hooks to fetch, modify, store and display chat generator test parameters. On selecting the Chat Generator Parameters page a REST API call is initiated to the chat generator which will return and populate the parameters form. Changes can be made to each parameter, on submission of the form another REST API call is made to the chat generator where the updated parameters are stored.



The screenshot shows a web form titled "Chat Generator Parameters" with a dark grey background. At the top, a yellow banner displays "Framework: Express". Below this, there are five input fields, each with a label and a value: "Number of Concurrent Callers" (20000), "Number of Chat Messages per Interaction (does not include agent BYE message to end interaction)" (2), "Delay prior to Sending First Chat Message (ms)" (5000), "Delay before responding to Chat Message (ms)" (5000), and "Timeout for Agent Join (ms)" (20000). Each input field has a small up/down arrow icon on the right. At the bottom of the form is a blue "Submit" button.

Figure 3 Chat Generator Parameter settings

5.2.3 ChatStats Component

The following live data is displayed within the ChatStats page. The following data is received every second from the back end using the SocketIO Client Library:

- Total Events received by the Chat Generator Express/Fastify web framework
- Times of first and last chat interaction session created
- ResourceGraph component to display live memory, system time and user time resource usage during performance runs. See further below for details of this ResourceGraph implementation using socket-io client and the recharts library.

I decided to use the ReChart Library due to its ease of use and customization options available (see **Appendix** for reference). The following AreaChart plots the current memory, system time and user time consumed by the chat generator every second. By hovering over the graph, a tooltip will appear identifying the resource usage at the specified time.

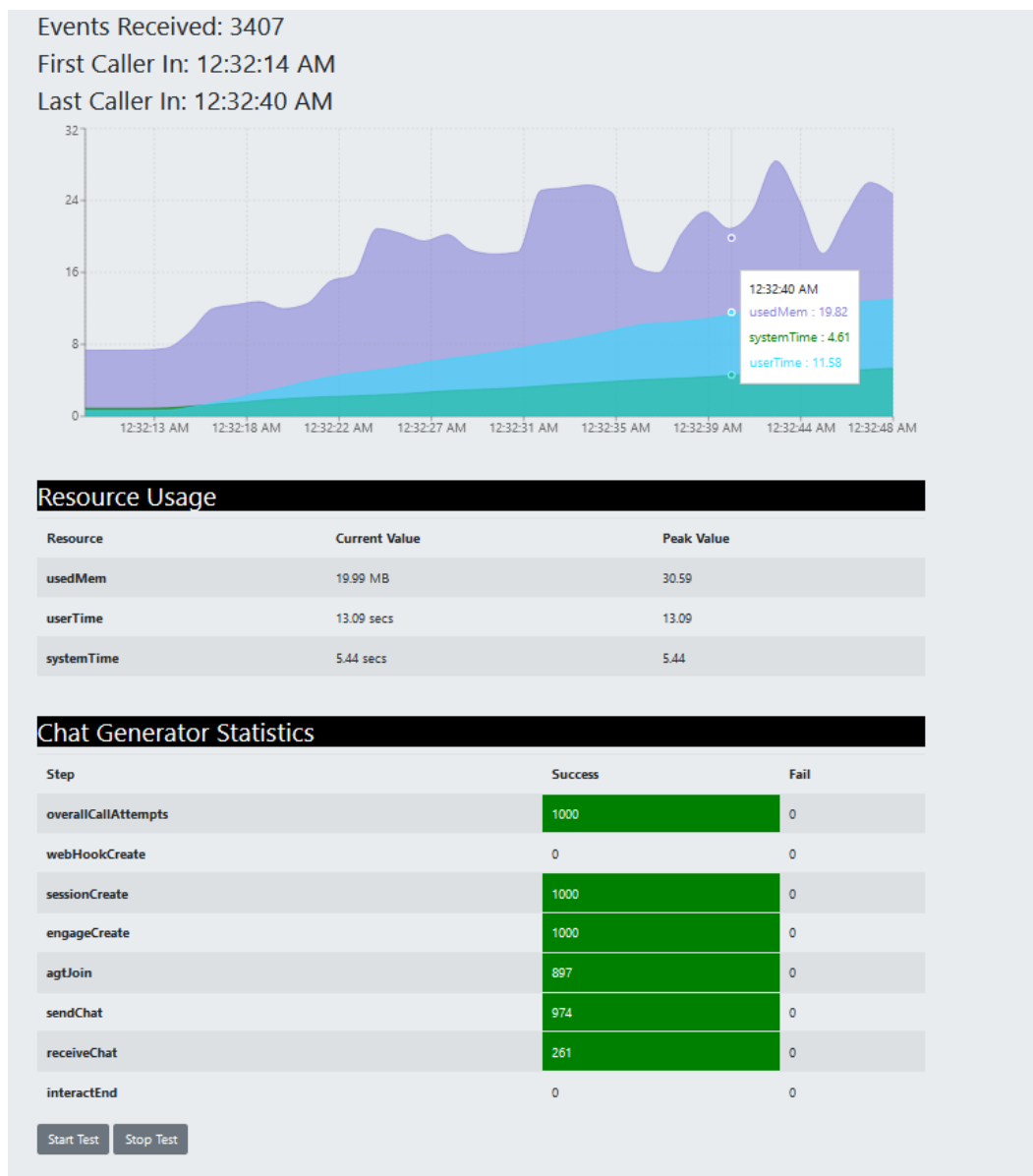


Figure 4 Resource Usage Statistics

The chat statistics table updates with live data when the test has started. Using a ternary operator, the background colour is updated based on a set condition i.e., a green background is displayed when the success value is greater than 0 and a red background is displayed when the failure value is greater than 0. This condition can be easily altered should the tester wish to set a higher threshold for accepted failures for example.

```
<td className={` ${value[0] >= 1 ? "green-highlight" : ""} `}>{value[0]}</td>

<td className={` ${value[1] >= 1 ? "red-highlight" : ""} `}>{value[1]}</td>
```

Chat Generator Statistics		
Step	Success	Fail
overallCallAttempts	2	0
webHookCreate	0	0
sessionCreate	1	1
engageCreate	1	0
agtJoin	1	0
sendChat	3	0
receiveChat	3	0
interactEnd	1	0
<div>Start Test</div> <div>Stop Test</div>		

Figure 5 Chat Generator Test Statistics

5.2.4 Resource Chart Component

Recharts are used to display several different chart types inside a React UI. I decided to use an area chart to display the resource usage statistics on the chat statistics page.

One thing to note about the data object the Recharts component expects, the data object must be an array of objects. Each key within the object will be plotted on the graph. I created a graphData object at the backend to match the object the Rechart component expects.

Backend **graphData** object:

```
graphData: {time: utils.currentTime(), usedMem: usedMem, userTime: cpuTime.userTime, systemTime: cpuTime.systemTime}
```

The ResourceChart component will receive the above graphData object from the WebSocketStats component. The graphData object will be stored inside the AreaChart 'data' property. Each graphData object value will be retrieved and plotted on the graph by assigning each graphData object key to the dataKey property within the Area component.

```
const ResourceChart = ({ graphData }) => {  
  return (  
    <div>  
      <h2 className="table-space">Resource Graph </h2>  
      <AreaChart  
        width={1100}  
        height={400}  
        data={graphData}  
        margin={{  
          top: 10,  
          right: 30,  
          left: 0,  
          bottom: 0,  
        }}  
      >
```

```
        <Area  
          type="monotone"  
          dataKey="usedMem"  
          stroke="#8884d8"  
          fill="#8884d8"  
        />  
        <Area  
          type="monotone"  
          dataKey="systemTime"  
          stroke="#008000"  
          fill="#008000"  
        />  
        <Area  
          type="monotone"  
          dataKey="userTime"  
          stroke="#33ddff"  
          fill="#33ddff"  
        />  
      </AreaChart>  
    </div>  
  )  
}
```

5.2.5 Websocket Component - SocketIO Client

Within the `useEffect` hook a connection is made to the Socket Server (chat generator back-end). The socket first listens for an event i.e., chat and resource statistics data from back-end, then proceeds to update React state with these values. One thing I discovered about the `useState` hook is that the function it returns to update the current state can take a callback, in the code snippet below the callback for updating the `graphData` state was labelled 'currentData'. The reason I use a callback here is that this callback will grab the current graph data stored in state i.e., 'graphData' at the time of the callback function call. Next using the spread operator to make a copy of the existing `graphData` state array, I append this array with updated resource values passed in via socket-io client. This will result in the graph updating every second while maintaining historical data.

The websocket component will then return the chat and resource statistics to the `ChatStats` component where it will be rendered using the `ReChart` library discussed above.

The logic for cleaning up timers and listeners in JavaScript is paramount to avoid memory leaks in the frontend. The connection needs to be closed when the component disappears from the DOM i.e., when the user navigates away from the page. To do so, a function is returned from `useEffect`, with a call to `disconnect()` on the client. To view the entire resource usage throughout the test via the graph, it is recommended NOT to navigate away from the chat statistics page until the test run has completed as this will clear historical data displayed in the graph. This is a side effect of cleaning up timers and listeners in to avoid memory leaks.

```
const WebsocketStats = () => {
  const [chatStats, setChatStats] = useState("");
  const [graphData, setGraphData] = useState([]);

  // retrieve SUT IP and Port from chatStats-api
  const ENDPOINT = `http://${baseIP}:${port}/`;

  // connect to the socket server on component mount with useEffect. Then, we save each new incoming message in the component's state.
  useEffect(() => {
    const socket = socketIOClient(ENDPOINT);
    socket.on("FromAPI", data => {
      setChatStats(data);
      // setGraphData can not only take a value but it can also take a callback
      // currentData will grab the current value stored in state when this callback is invoked, I then append this array each time with updated resource values
      setGraphData( (currentData) => [
        ...currentData, data.graphData
      ]);
    });
    // the logic for cleaning up timers and listeners in JavaScript is paramount to avoid memory leaks in the frontend. We need also to close the connection when the component disappears from the DOM. To do so, we return a function from useEffect, with a call to disconnect() on the client
    return () => {
      socket.disconnect();
    };
  }, []);

  return [chatStats, graphData];
}
```


5.3 Automate Test environment using Jenkins

I decided to implement a CI/CD pipeline using Jenkins to manage the build and deployment of the Front and Backend applications throughout the development cycle. I chose Docker as the platform to deploy the Jenkins container (and later the chat generator and React front end containers).

Jenkins allowed me to continuously integrate new changes that would automatically deploy the chat generator and React front end applications into a Docker container for verification tests to be performed.

Using the following Dockerfile I created a custom Jenkins image on an Ubuntu server **References [1, 2]**.

```
FROM jenkinsci/jenkins
MAINTAINER Sreeprakash Neelakantan <sree@schogini.com>
USER root
RUN apt-get update && apt-get install -y tree nano curl sudo
RUN apt-get install ca-certificates
RUN curl https://get.docker.com/builds/Linux/x86_64/docker-latest.tgz | tar xzv -C /tmp/ && mv /tmp/docker/docker /usr/bin/docker
RUN curl -L "https://github.com/docker/compose/releases/download/1.23.1/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
RUN chmod 755 /usr/local/bin/docker-compose
RUN usermod -a -G sudo jenkins
RUN echo "jenkins ALL=(ALL:ALL) NOPASSWD:ALL" >> /etc/sudoers
USER jenkins
```

Using 'docker run' and 'docker compose' I created and deployed the Jenkins container. Note within the below docker run command, the docker socket is mapped so that docker commands running within the Jenkins (job) can talk to the host Docker Machine. This was important as I need to be able to create and deploy containers for the Back End chat generator and React Front End from within the Jenkins container.

docker.sock is the UNIX socket that Docker daemon is listening to.

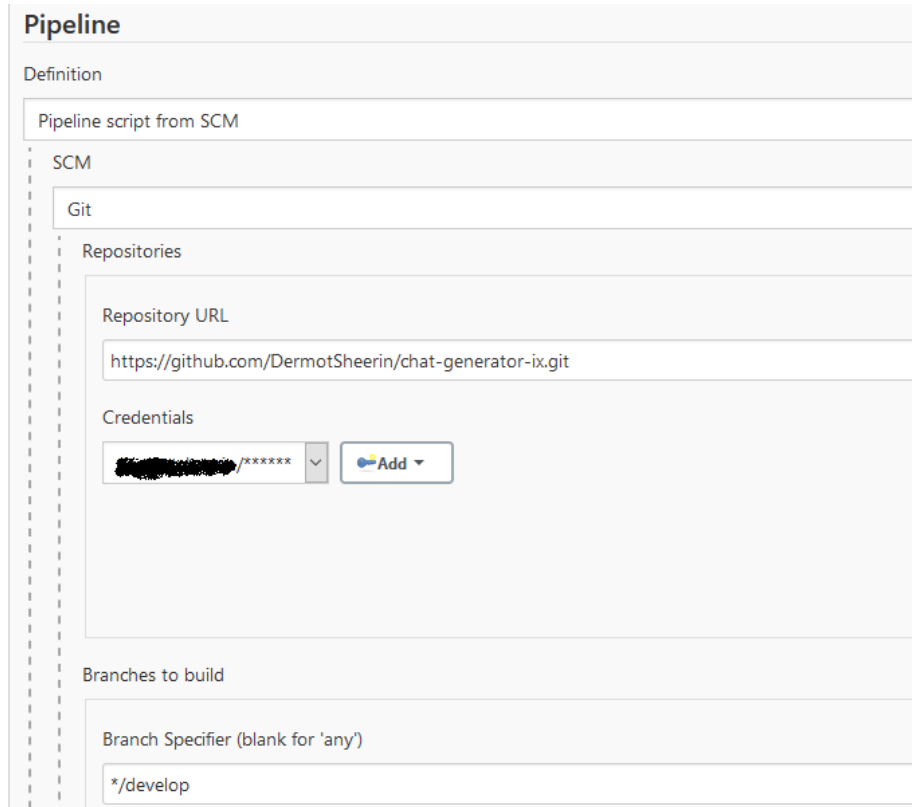
```
sudo docker run -itd -e JENKINS_USER=$(id -u) \
-v /var/run/docker.sock:/var/run/docker.sock \
-v $(pwd)/jenkins_home:/var/jenkins_home \
-v $(which docker):/usr/bin/docker \
-p 8880:8080 -p 50000:50000 \
-u root \
schogini/jenkinsci-docker-compose:v1
```

See **Appendix** for useful Docker commands I discovered during my research.

5.3.1 Jenkinsfile (Declarative Pipeline) from Source Control (SCM)

Two pipelines were created using Jenkinsfile, one for the nodeJS Chat Generator and one for React Front End. The Jenkinsfile is checked into both repositories (Github).

The pipeline will checkout the Jenkinsfile from Github as part of the Pipeline project's build process. It will then proceed to execute the Pipeline. See below example for the SCM configuration used.



The image shows the Jenkins Pipeline configuration interface. The title is "Pipeline". Under the "Definition" section, there is a dropdown menu set to "Pipeline script from SCM". Below this, the "SCM" section is expanded, showing "Git" as the selected provider. Under "Repositories", the "Repository URL" is set to "https://github.com/DermotSheerin/chat-generator-ix.git". The "Credentials" section shows a dropdown menu with a redacted name and an "Add" button. Under "Branches to build", the "Branch Specifier (blank for 'any')" is set to "*/develop".

Figure 6 Jenkins Declarative Pipeline from SCM

Build Triggers allows you to select when you wish to trigger your pipeline. My preferred option was to use *GitHub webhook trigger for GITScm polling* where the pipeline would be triggered when changes were pushed to the repository. However due to network constraints between the SUT and Github when using the webhook callback, I opted for the *Poll SCM* trigger. With this option selected, Github is polled periodically for new commits to the branch specified. This is not a very efficient way to implement the build trigger however for demo and test purposes only I opted for the Poll SCM trigger.

Build Triggers

- ☐ Build after other projects are built
- ☐ Build periodically
- ☐ GitHub Branches
- ☐ GitHub Pull Requests
- ☐ GitHub hook trigger for GITScm polling
- ☒ Poll SCM

Schedule

⚠ Do you really mean "every minute" when you say "*****"? Perhaps you meant "H *****" to poll once per hour

Figure 7 Jenkins Build Triggers

See the **Appendix** section for contents of the Jenkinsfile used.

Jenkins

Dashboard

- New Item
- People
- Build History
- Project Relationship
- Check File Fingerprint

S	W	Name ↓	Last Success
		chat-generator	9 days 15 hr - #57
		react-front-end	5 days 15 hr - #28

Icon: S M L

Figure 8 Jenkins Dashboard

5.3.2 Dockerfile

Docker will build images automatically by reading instructions from a Dockerfile. It contains all the commands a user could call on the command line to assemble an image. See **Appendix** for Dockerfiles used to create the chat generator and React images.

5.3.3 Docker-compose

Docker-compose will assemble and deploy docker applications. Docker-compose is used in the Deployment stage of the Jenkinsfile.

Both the Dockerfile and the Docker-compose files will specify the port exposed to access the application inside the container. See **Appendix** for ports specified for the Chat Generator and React UI.

At this point the CI/CD pipeline is ready.

Implementing the above means I can now make changes to my local branch and once I push to Github, an automated build and deployment of my code will be executed. Once complete, the latest code changes will be running in a Docker container.

The follow displays the status screen for both pipelines, a **Build History** will indicate whether the pipeline build was a Success or Failure.

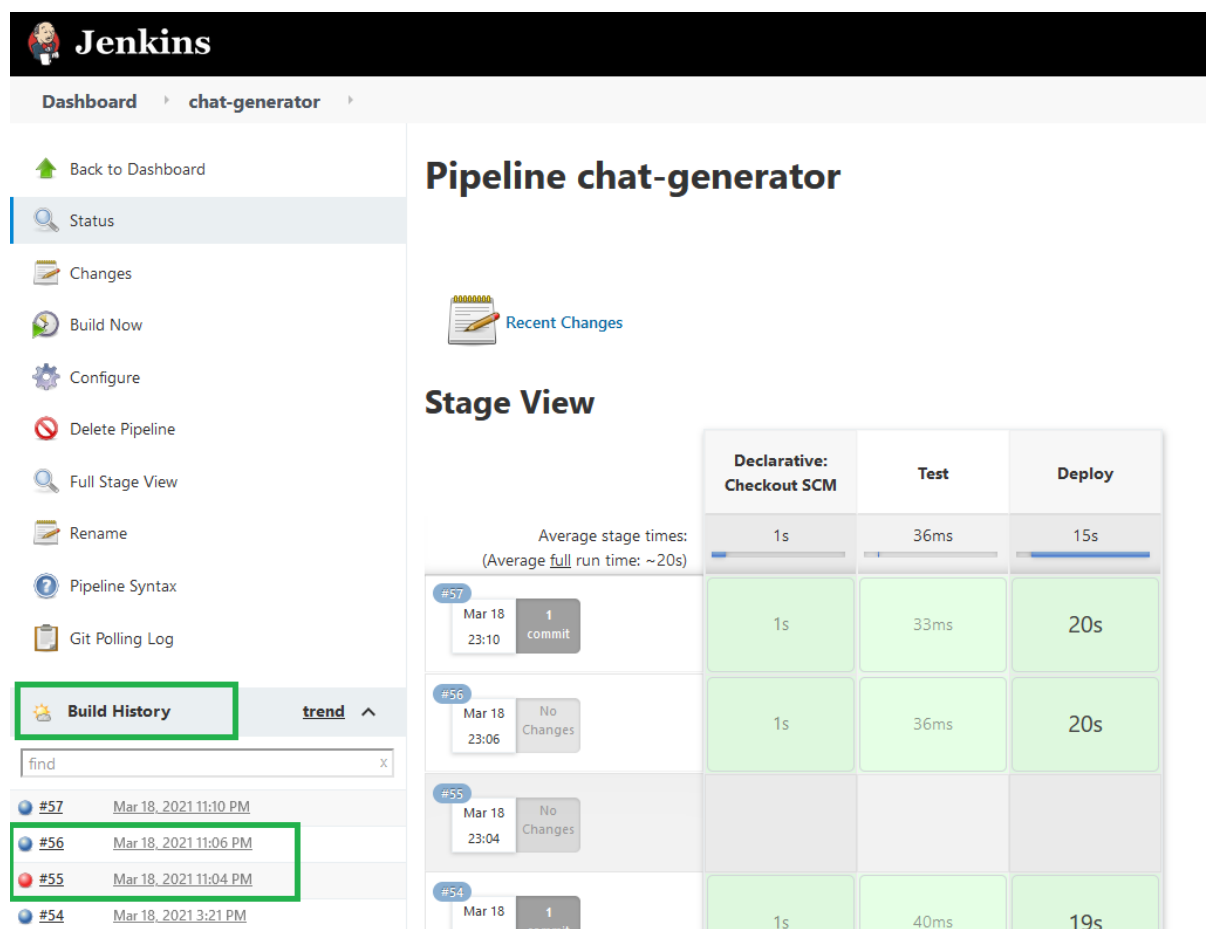


Figure 9 Chat Generator Pipeline

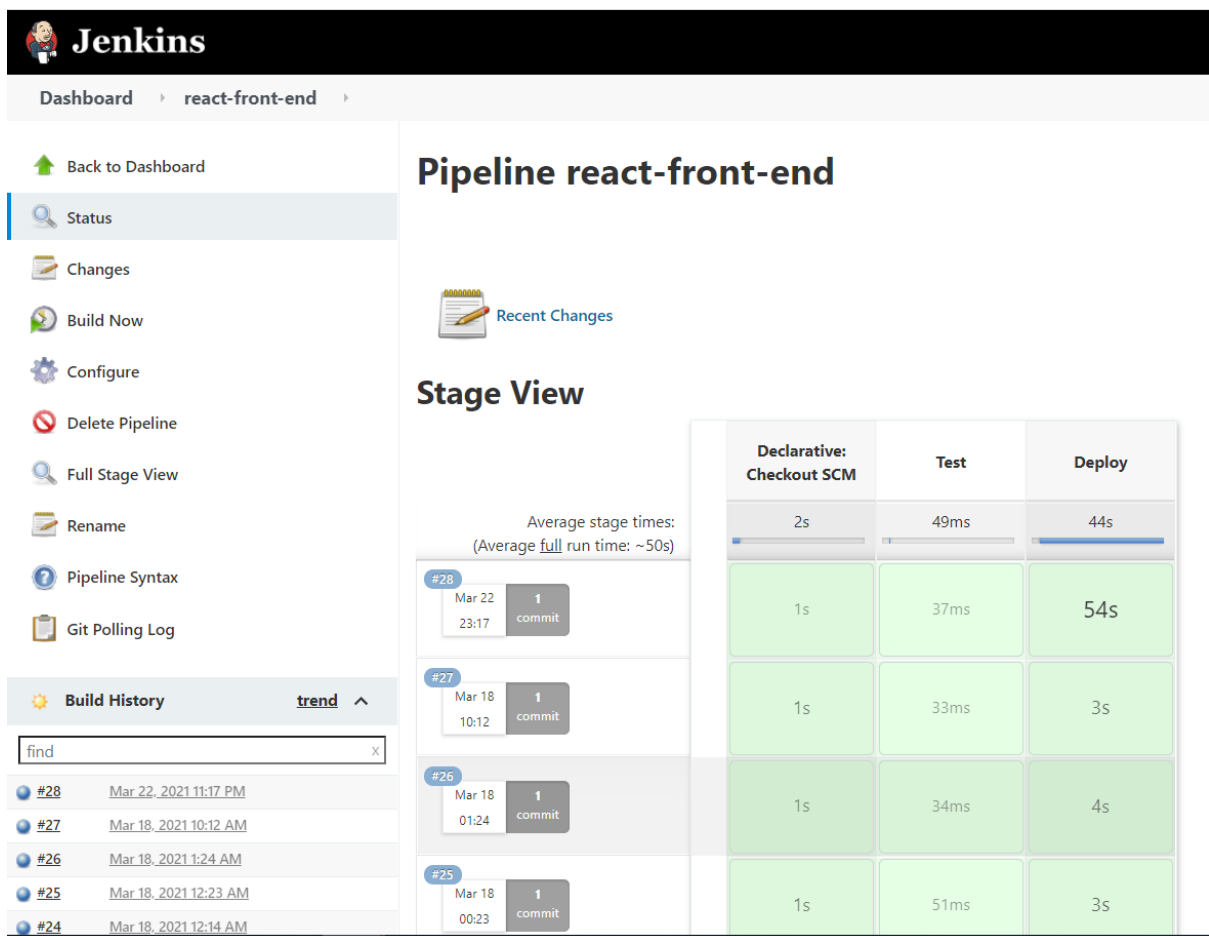


Figure 10 React Front End Pipeline

```

+ docker-compose build
Building chatgenerator
Step 1/5 : FROM node:12
---> af3e1e2da75b
Step 2/5 : COPY . /
---> ae6d477493ae
Step 3/5 : RUN npm install
---> Running in 880f26755ece
[91mnpn[0m[91m WARN chat-generator-ix@1.0.0 No description
[0m[91m
[0madded 147 packages from 153 contributors and audited 147 packages in 3.477s

5 packages are looking for funding
  run `npm fund` for details

found 1 high severity vulnerability
  run `npm audit fix` to fix them, or `npm audit` for details
Removing intermediate container 880f26755ece
---> 169ad4fb6126
Step 4/5 : EXPOSE 8001
---> Running in 5a633cb158b2
Removing intermediate container 5a633cb158b2
---> 8ecb39bc717b
Step 5/5 : CMD [ "node", "index.js" ]
---> Running in edabd99c4334
Removing intermediate container edabd99c4334
---> 26dd7e986bcd

Successfully built 26dd7e986bcd
Successfully tagged chat-generator_chatgenerator:latest
[Pipeline] sh
+ docker-compose up -d
Recreating chatgenerator ...
[1A[2K
Recreating chatgenerator ... [32mdone[0m
[1B
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS

```

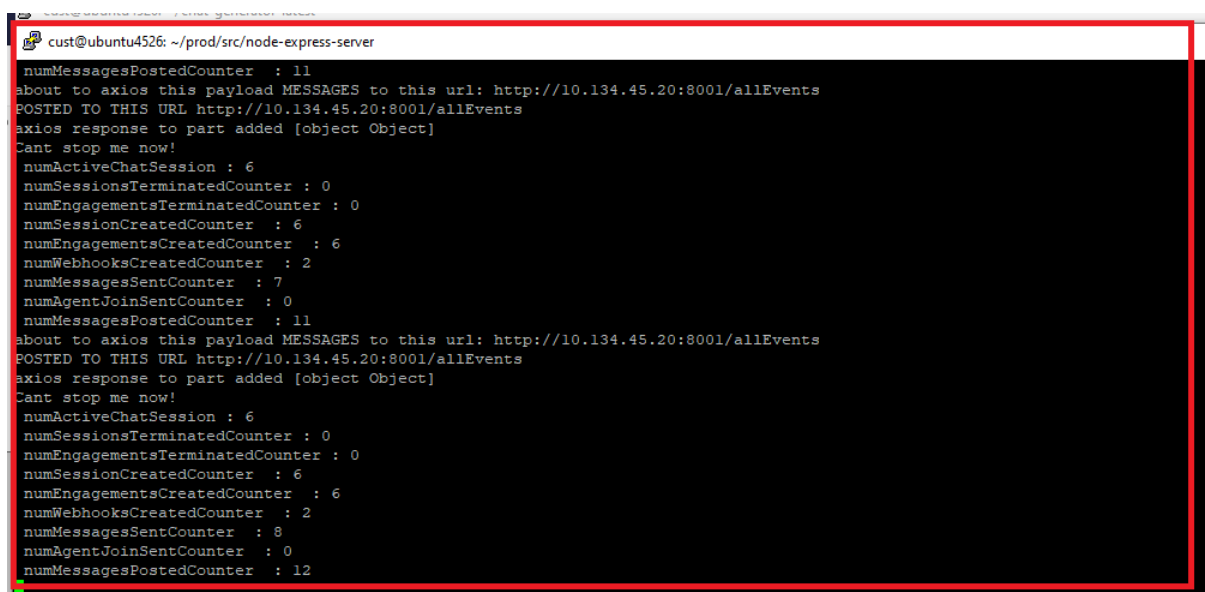
Figure 11 Successful Pipeline Build Log

6 Sample Test run “Express V Fastify” comparison

Using the test environment discussed above I was able to automatically build and deploy the chat generator and chat UI applications into a docker container using the Jenkins pipeline. Once deployed I was able to run low level traffic to test the Express and Fastify frameworks.

The next step was to deploy the chat generator onto an Ubuntu server. The mock server and chat generator will reside on separate servers.

Sample output from Mock Server (developed using Express) listening on port 4000 for events from chat generator.



```
cust@ubuntu4526: ~/prod/src/node-express-server
numMessagesPostedCounter : 11
about to axios this payload MESSAGES to this url: http://10.134.45.20:8001/allEvents
POSTED TO THIS URL http://10.134.45.20:8001/allEvents
axios response to part added [object Object]
Cant stop me now!
numActiveChatSession : 6
numSessionsTerminatedCounter : 0
numEngagementsTerminatedCounter : 0
numSessionCreatedCounter : 6
numEngagementsCreatedCounter : 6
numWebhooksCreatedCounter : 2
numMessagesSentCounter : 7
numAgentJoinSentCounter : 0
numMessagesPostedCounter : 11
about to axios this payload MESSAGES to this url: http://10.134.45.20:8001/allEvents
POSTED TO THIS URL http://10.134.45.20:8001/allEvents
axios response to part added [object Object]
Cant stop me now!
numActiveChatSession : 6
numSessionsTerminatedCounter : 0
numEngagementsTerminatedCounter : 0
numSessionCreatedCounter : 6
numEngagementsCreatedCounter : 6
numWebhooksCreatedCounter : 2
numMessagesSentCounter : 8
numAgentJoinSentCounter : 0
numMessagesPostedCounter : 12
```

Figure 12 Sample console output from Mock Server

Here is a sample output from Chat Generator for a single chat interaction, listening on port 8001 for events from the Mock Server. The agent join timer was set to 20secs, in the first call example below the agent did not answer within the specified time and as a result the error was logged out to the console. The second call attempt was successful, the customer sent a message and received one back from the agent. This loop was repeated twice before the customer sends a BYE message to terminate the interaction.

```

2021-04-30 15:53:12.850 Listening on IP: 10.134.45.20: port 8001
2021-04-30 15:53:51.812 New client connected
2021-04-30 15:53:58.637 Client disconnected
2021-04-30 15:54:19.742 received chatParameters, concurrentCallers: 1
2021-04-30 15:54:22.331 New client connected
2021-04-30 15:54:25.869 ##### Chat Generator Started #####
2021-04-30 15:54:25.875 Running createChatSession - (Dermot0 and concurrentCallers: 1)
2021-04-30 15:54:25.878 =====> sessionId created: 100105
2021-04-30 15:54:27.882 =====> engagementID created: 101005
2021-04-30 15:54:27.882 Wait for promise to be resolved for agent join, engId: 101005
2021-04-30 15:54:47.883 Agent JOIN Timed out or Agent did not answer in 20000 ms !!! for engID: 101005, agentJoinTimer returned: false
2021-04-30 15:55:18.130 ----- Stats Reset -----
2021-04-30 15:55:21.463 ##### Chat Generator Started #####
2021-04-30 15:55:21.464 Running createChatSession - (Dermot0 and concurrentCallers: 1)
2021-04-30 15:55:21.466 =====> sessionId created: 100106
2021-04-30 15:55:23.468 =====> engagementID created: 101006
2021-04-30 15:55:23.468 Wait for promise to be resolved for agent join, engId: 101006
2021-04-30 15:55:31.476 Agent Join Received for engId: 101006
2021-04-30 15:55:31.477 Engagement ID IS contained within the promiseMap: 101006
2021-04-30 15:55:31.477 =====> Agent Join Promise has been resolved and set to true
2021-04-30 15:55:36.480 Customer Dermot0 sent message success
2021-04-30 15:55:36.480 Wait for Agent MESSAGE
2021-04-30 15:55:44.483 Agent Message has been Received: true engId: 101006
2021-04-30 15:55:49.485 Customer Dermot0 sent message success
2021-04-30 15:55:49.485 Wait for Agent MESSAGE
2021-04-30 15:55:57.493 Agent Message has been Received: true engId: 101006
2021-04-30 15:56:02.493 Customer about to send bye, engID: 101006
2021-04-30 15:56:02.495 Customer Dermot0 sent message success
2021-04-30 15:56:02.496 Agent Terminated successfully engId: 101006 chatStats: overallCallAttempts,1,0,webHookCreate,0,0,sessionCreate,1,0,0

```

Figure 13 Sample console output from Chat Generator

6.1 Modular development of NodeJS Chat Generator

The chat generator code is implemented in a modular way which results in a more manageable code structure. Should the user wish to test against a different set of APIs, web framework or simply change some of the resource usage calculations then this implementation accommodates such requests.

Logger → to avoid duplication, this module contains a small wrapper around nodeJS standard output method (console.log()) by adding the current date to each log message. This is a useful addition for troubleshooting as the timestamp is logged for each event that is written to standard out.

Routes → this contains two files, one for the Express handlers including the Socket IO implementation and one for Fastify including fastify-socket-io. To switch between using Express or Fastify a simple one line change is all that is required within the index.js module.

```

6 // for Express framework import the following module
7 //let { server, framework, resetEventCounter } = require("../routes/index").server;
8
9 // for Fastify framework import the following module
10 let { server, framework, resetEventCounter } = require("../routes/indexFastify").server;

```

Figure 14 How to switch between Express and Fastify

Services → all REST APIs used to interact with the Mock Contact Center.

Utilities → utility functions to calculate the current resource usage of the chat generator e.g., memory and CPU resource consumption.

Index → the main file for application start up that imports all the above modules.

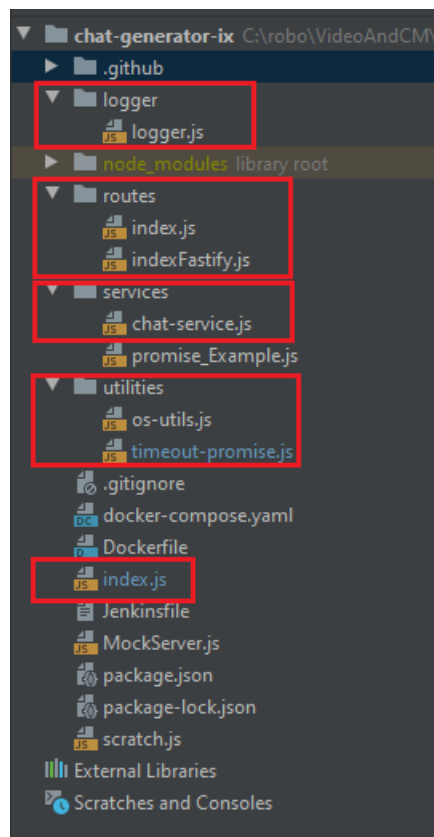
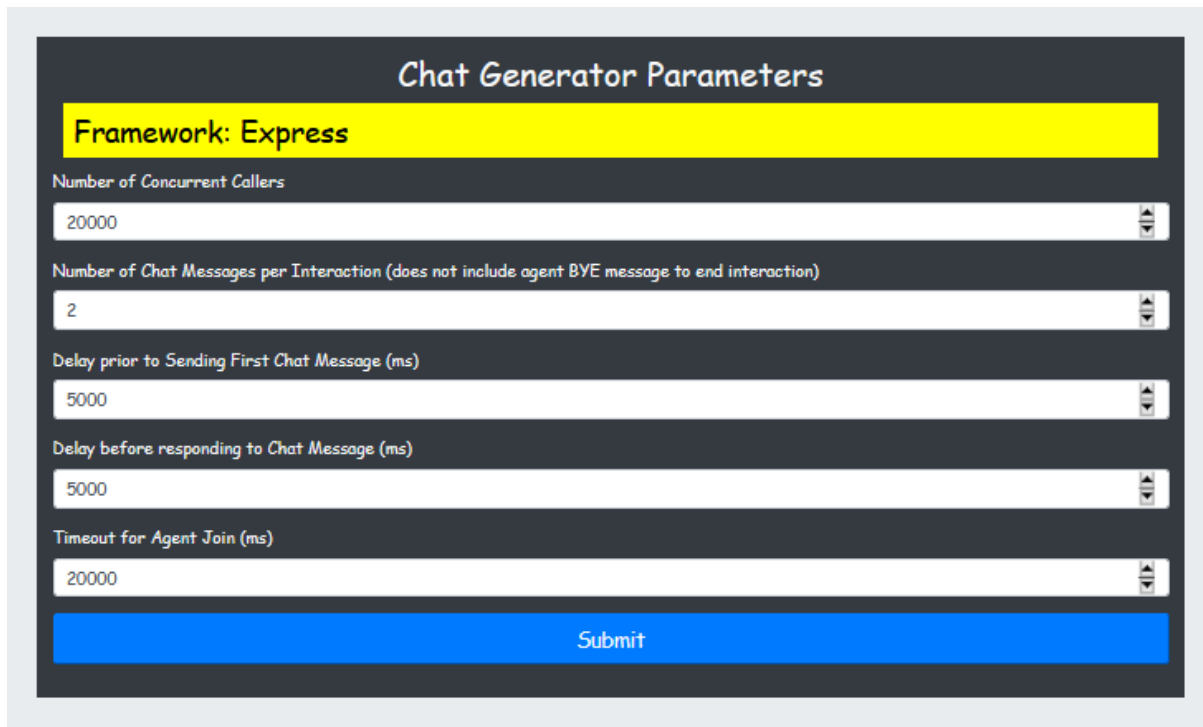


Figure 15 NodeJS Modular File structure

6.2 20k chat generator test using EXPRESS

The following parameters were set for the performance run using Express. Note that the number of chat messages specified below i.e., 2 does not include the BYE message sent by the generator i.e., the customer will send 2 messages and the third message will be the BYE to terminate the interaction. The chat statistics UI will however register the BYE message as a chat sent from the customer therefore displaying 3 chats sent from customer to agent.



The screenshot shows a configuration window titled "Chat Generator Parameters". It features a yellow header bar with the text "Framework: Express". Below this, there are five input fields, each with a label and a value: "Number of Concurrent Callers" (20000), "Number of Chat Messages per Interaction (does not include agent BYE message to end interaction)" (2), "Delay prior to Sending First Chat Message (ms)" (5000), "Delay before responding to Chat Message (ms)" (5000), and "Timeout for Agent Join (ms)" (20000). Each input field has a small up/down arrow icon on the right. At the bottom of the window is a large blue button labeled "Submit".

Parameter	Value
Framework	Express
Number of Concurrent Callers	20000
Number of Chat Messages per Interaction (does not include agent BYE message to end interaction)	2
Delay prior to Sending First Chat Message (ms)	5000
Delay before responding to Chat Message (ms)	5000
Timeout for Agent Join (ms)	20000

Figure 16 Express parameter settings for 20k test

The following screen details the resource usage prior to the test run i.e., idle time.



The screenshot shows a table titled "Resource Usage". The table has three columns: "Resource", "Current Value", and "Peak Value". There are three rows of data: "usedMem" with a current value of 11.11 MB and a peak value of 13.86; "userTime" with a current value of 0.89 secs and a peak value of 0.89; and "systemTime" with a current value of 0.09 secs and a peak value of 0.09.

Resource	Current Value	Peak Value
usedMem	11.11 MB	13.86
userTime	0.89 secs	0.89
systemTime	0.09 secs	0.09

Figure 17 Express idle resource usage

The following displays the resource usage during the 20k chat interaction run. The graph displays a steady increase in CPU user time (User CPU time is the amount of time the processor spends in running the application code) as the number of interactions reach over 5k.

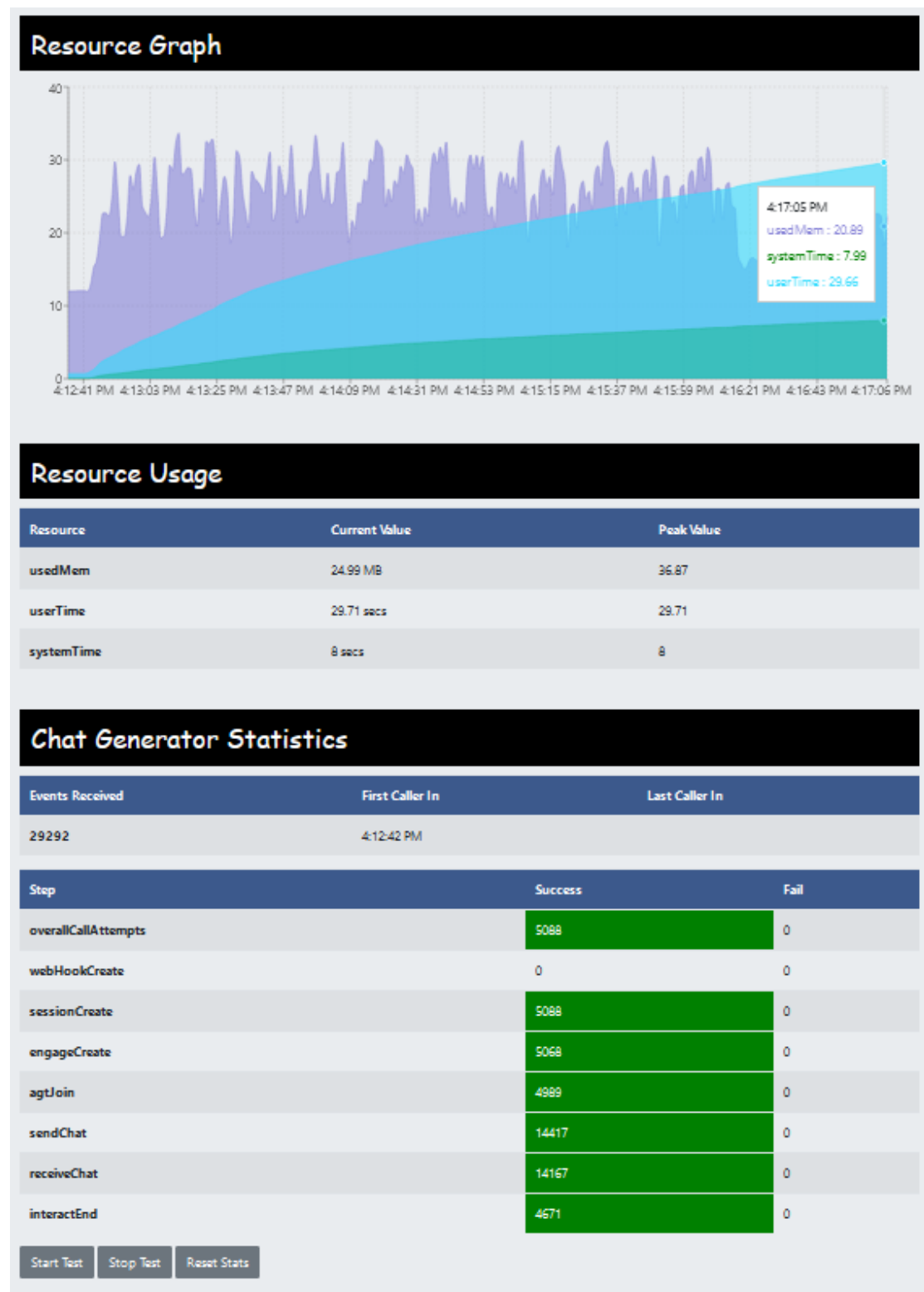


Figure 18 Express chat statistics at 5k interactions

On completion of the 20k chat interaction run we see that the memory usage altered between 20Mb→40Mb and registered a peak memory spike of approximately 45Mb, the CPU user time rose steadily throughout the test. The test in total took approximately 67 minutes for all 20k customer interactions to complete with no failures reported.

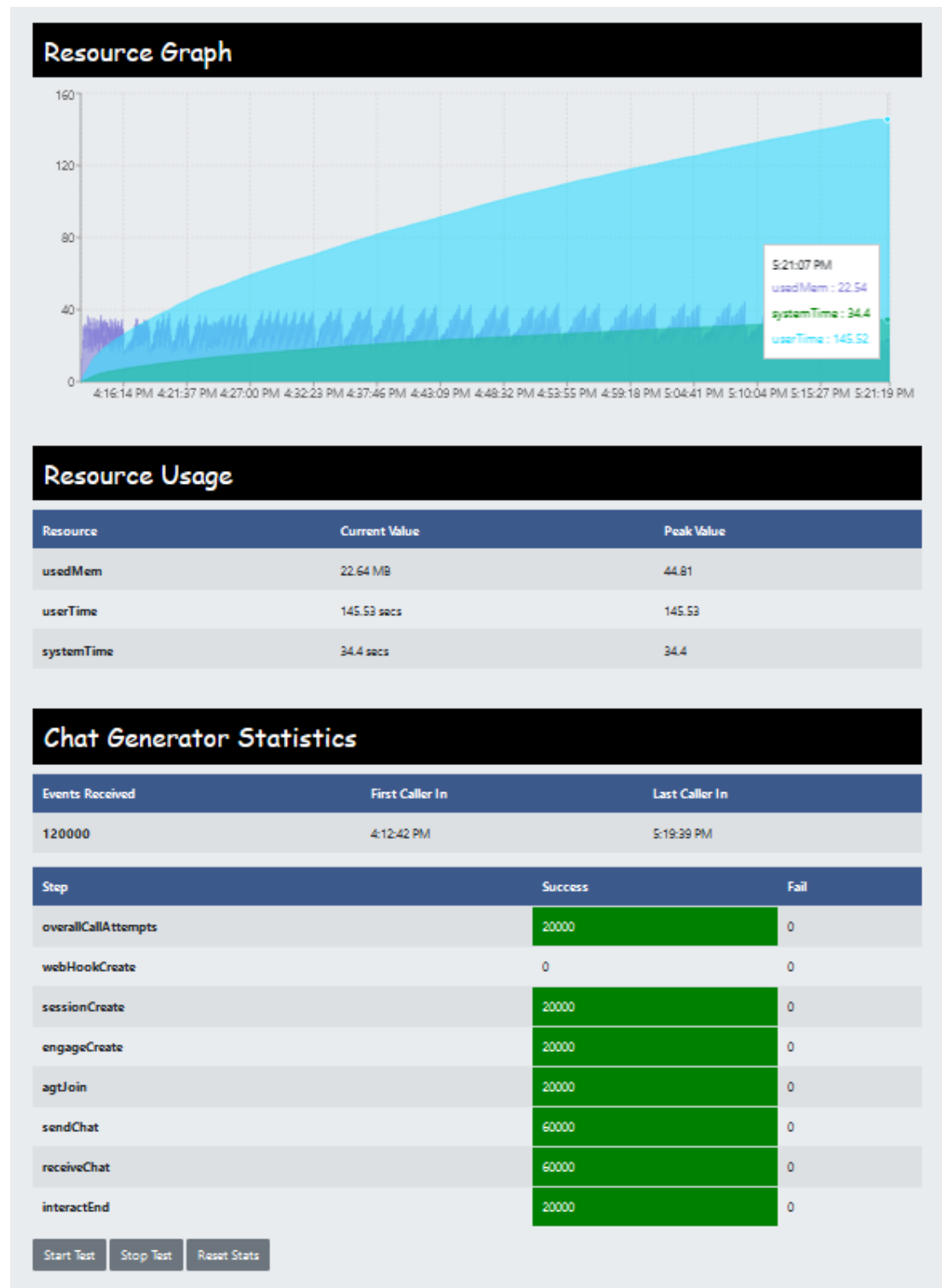
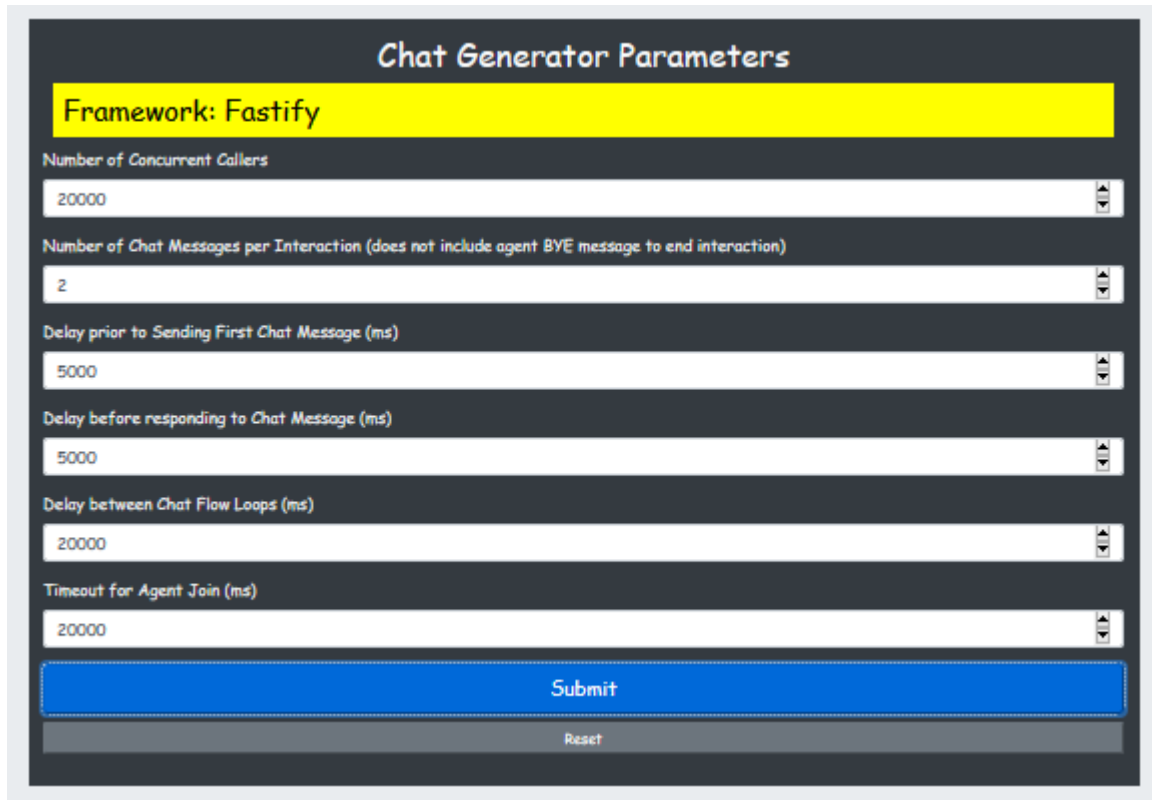


Figure 19 Express chat statistics at 20k interactions

6.3 20k chat generator test using FASTIFY

The same chat parameter settings were used for the Fastify performance run. The Framework header receives its value from the back end allowing the user to easily confirm the appropriate framework is in place for the test.



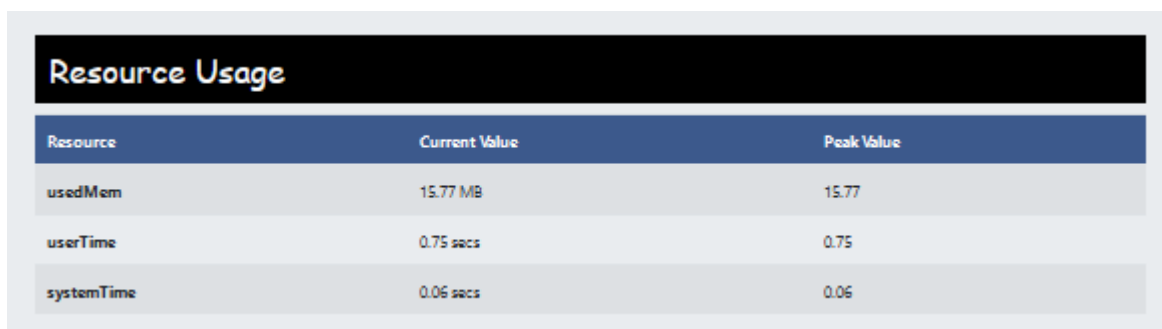
The screenshot shows a web interface titled "Chat Generator Parameters". A yellow banner at the top indicates the "Framework: Fastify". Below this, several parameters are listed with input fields and up/down arrows for adjustment:

- Number of Concurrent Callers: 20000
- Number of Chat Messages per Interaction (does not include agent BYE message to end interaction): 2
- Delay prior to Sending First Chat Message (ms): 5000
- Delay before responding to Chat Message (ms): 5000
- Delay between Chat Flow Loops (ms): 20000
- Timeout for Agent Join (ms): 20000

At the bottom, there are two buttons: a blue "Submit" button and a grey "Reset" button.

Figure 20 Fastify parameter settings for 20k test

The following screen details the resource usage prior to the test run i.e., idle time.



Resource Usage		
Resource	Current Value	Peak Value
usedMem	15.77 MB	15.77
userTime	0.75 secs	0.75
systemTime	0.06 secs	0.06

Figure 21 Fastify idle resource usage

The test using Fastify did not go well, during test I noticed a rapid increase in memory usage from the outset. After approximately four minutes errors were displayed on the UI stats relating to failed Agent Join events. The Mock server was displaying network connectivity errors.

Error: connect EADDRNOTAVAIL 10.134.45.20:8001 - Local (10.134.45.26:0)

Errors reported by the chat generator related to agent join timeouts which means the chat generator did not receive an agent join event within the agent timeout interval specified in the chat parameters configuration. This turned out to be a good test for my promise race implementation and proved its effectiveness.

2021-04-30 18:26:23.621 Agent JOIN Timed out or Agent did not answer in 20000 ms !!! for engID: 107508, agentJoinTimer returned: false

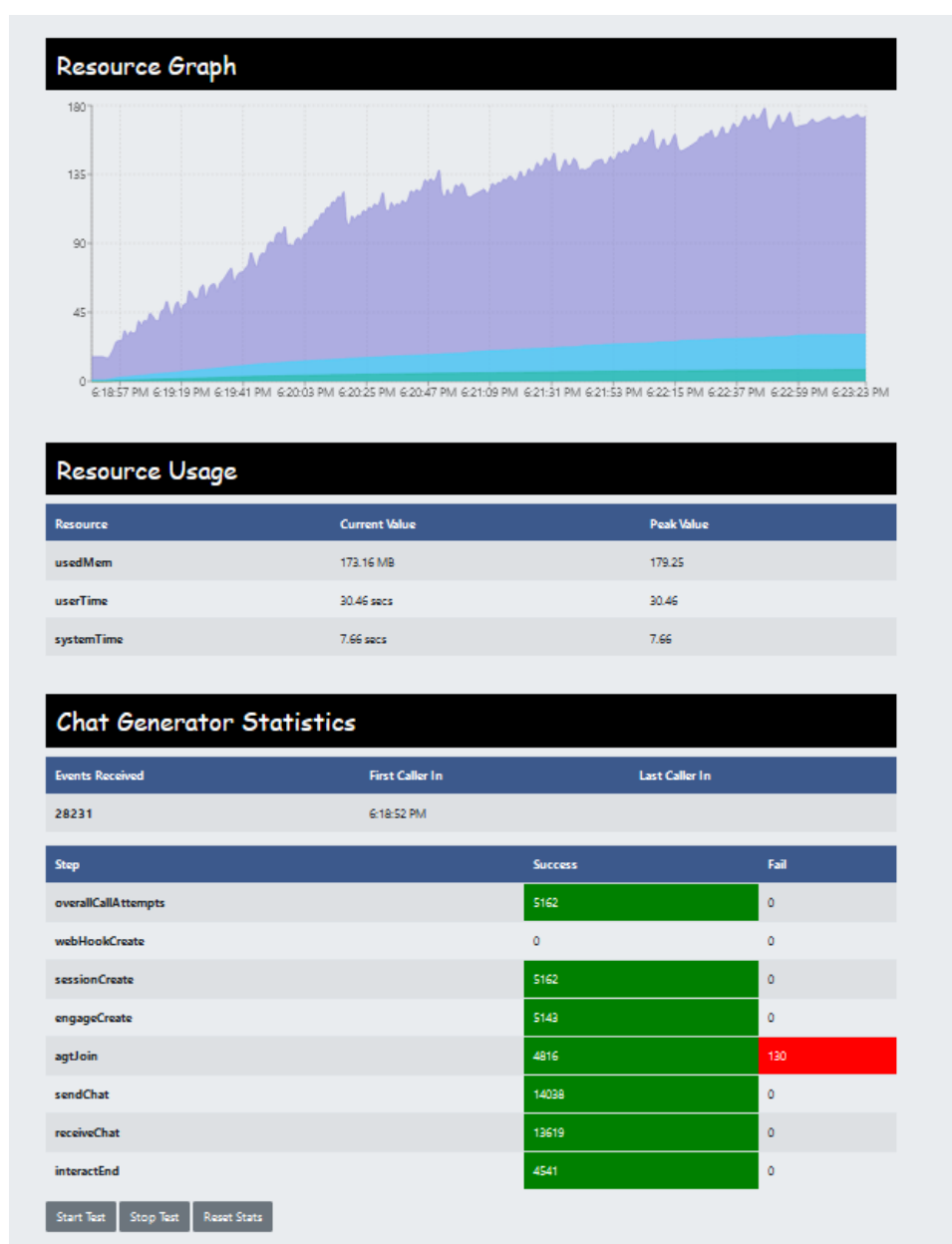


Figure 22 Fastify failures at 5k interactions

On first assessment the only difference between both tests was the web framework implementation. After some further research into the Fastify implementation I realised I was registering 'socket.io' as plug-in whereas 'fastify-socket.io' is the recommended use of Socket.io in a Fastify application. However, after repeating the test I experienced the same behaviour once the generator reached 5k sessions.

I then focused my analysis on the network side as the error reported by the Mock server was network related. For each failed test the same observations were noted, after approximately four minutes into the test the number of chat interactions created was approximately 5000. Using 'netstat' to see the network connections I noticed there were over 28k connections open to the chat generator on port 8001, which is what the generator was listening on. This number matched the event count received by the fastify framework. Could it be that all network connections bound to the chat generator on port 8001 are not closing? This in turn must be causing the large memory consumption. I am unsure if the cause of this issue is down to an incorrect implementation of the fastify framework or a genuine issue. The chat generator code is the same for both Express and Fastify tests with the only difference being the implementation of both frameworks in nodeJS.

28k connections open to port 8001 (matching the number of events received at Fastify framework)

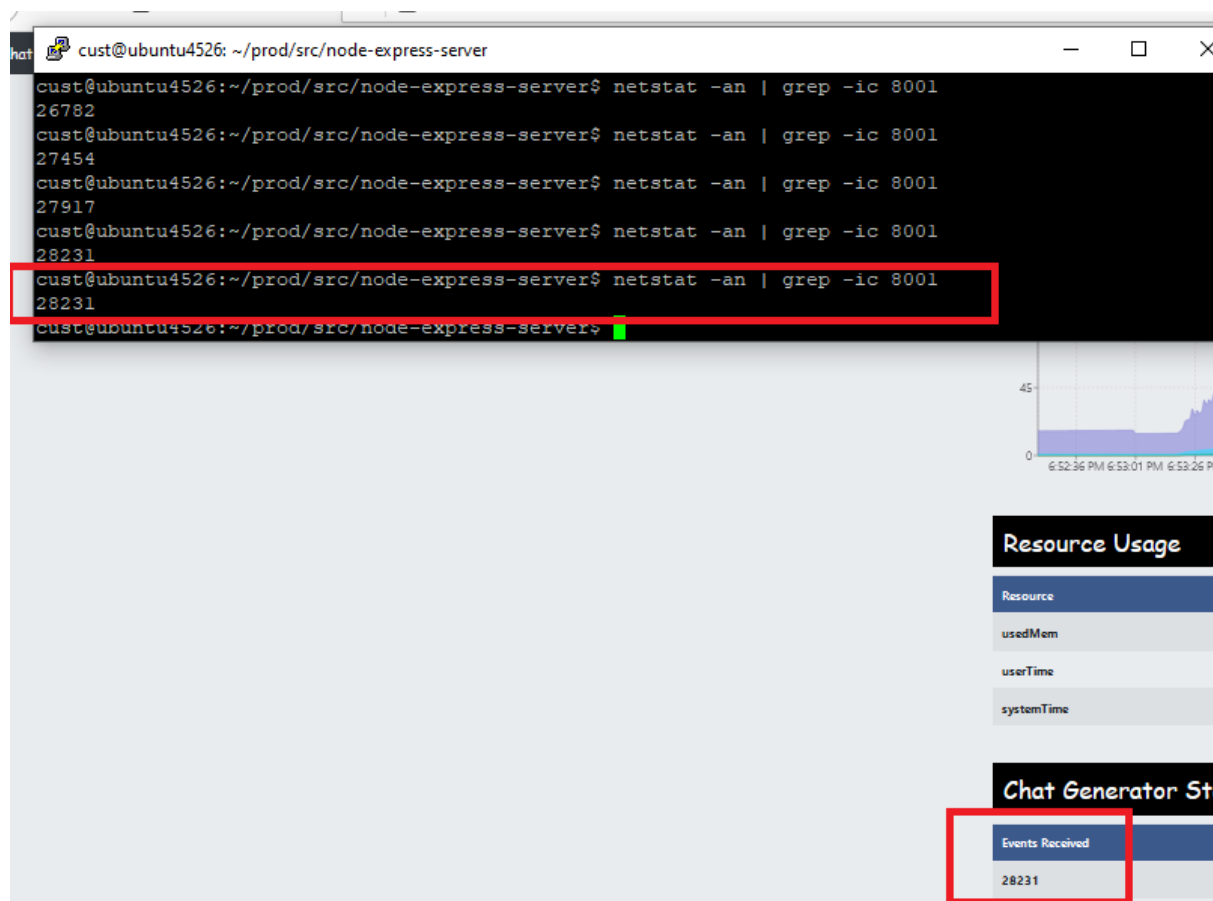


Figure 23 Open port connections for Fastify test

After comparing network statistics from the **Express** test I noticed the network connections were vastly lower at the 5k call attempts mark (over 40 times lower!):

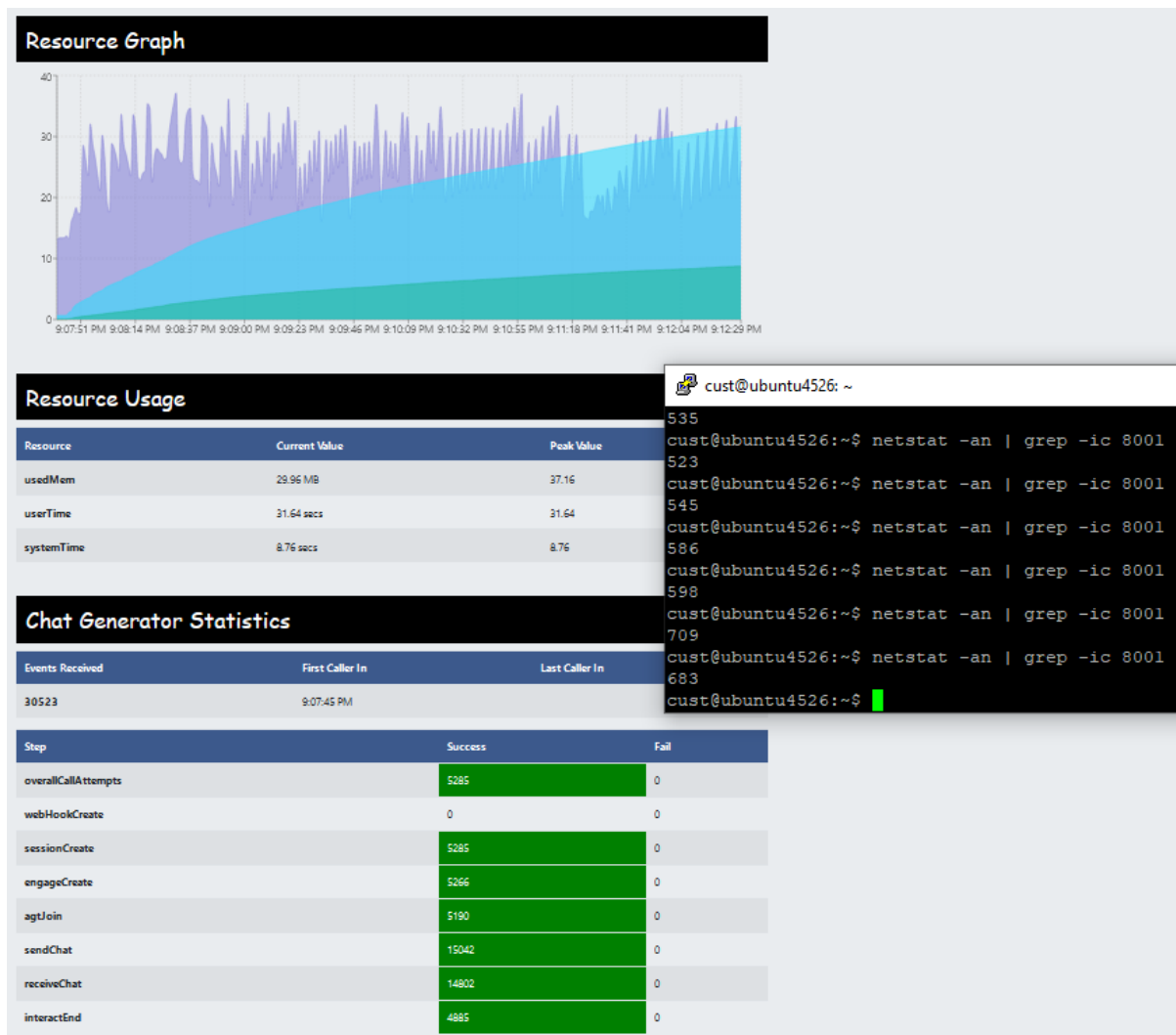


Figure 24 Open port connections for Express test

After spending some time thinking about the issue seen with Fastify, I suspected the port connections on the Mock server may not disconnect if they do not receive the 200 OK back from the Fastify framework. I decided to create a test endpoint '/demoTest' on Fastify that listens for POST events. I configured this endpoint in the same way as the chat generator endpoints i.e., in response to an incoming POST request to this endpoint, a reply with a status code 200 is sent back to indicate to the far end that the request was received.

```
server.post("/demoTest", (request, reply) => {  
  reply.code(200);  
});
```

Using POSTMAN I sent a POST request to this endpoint, however I did not get a 200 OK back. Instead the request remained in a 'Sending request ...' state. This might explain why the Mock Server was not releasing the port each time a POST request was received at the Fastify framework, the request was stuck in a sending request state as it never received the 200 OK back from Fastify.

After some further investigation I found that using 'reply.send(payload)' would in fact return a 200 OK. This response is normally used to send a payload to the far end which I didn't really need to do however for now I decided to change my Fastify POST replies to use the 'send' method. I am unsure why 'reply.code(200)' did not work as expected, this is something I will need to investigate further in the future.

With the updated change to the Fastify response codes I ran the 20k chat interaction test again. This time things were better; the network connections were on a par with Express when the 5k chat interaction was reached.

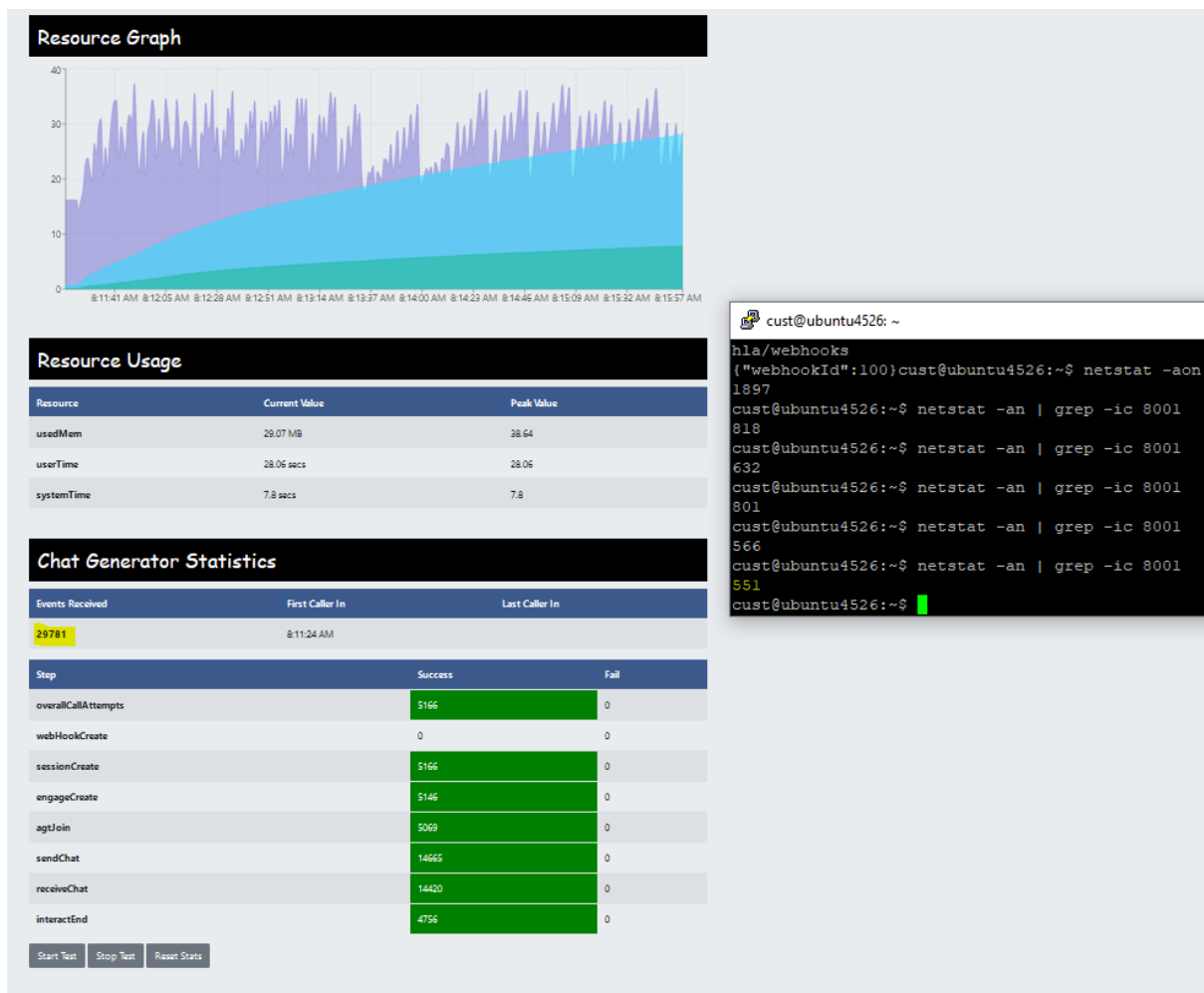


Figure 25 Open port connections for Fastify after resolution implemented

At 17k chat interactions, the resource utilisation was much better than the previous Fastify test run, it was also aligned with the Express behavior.

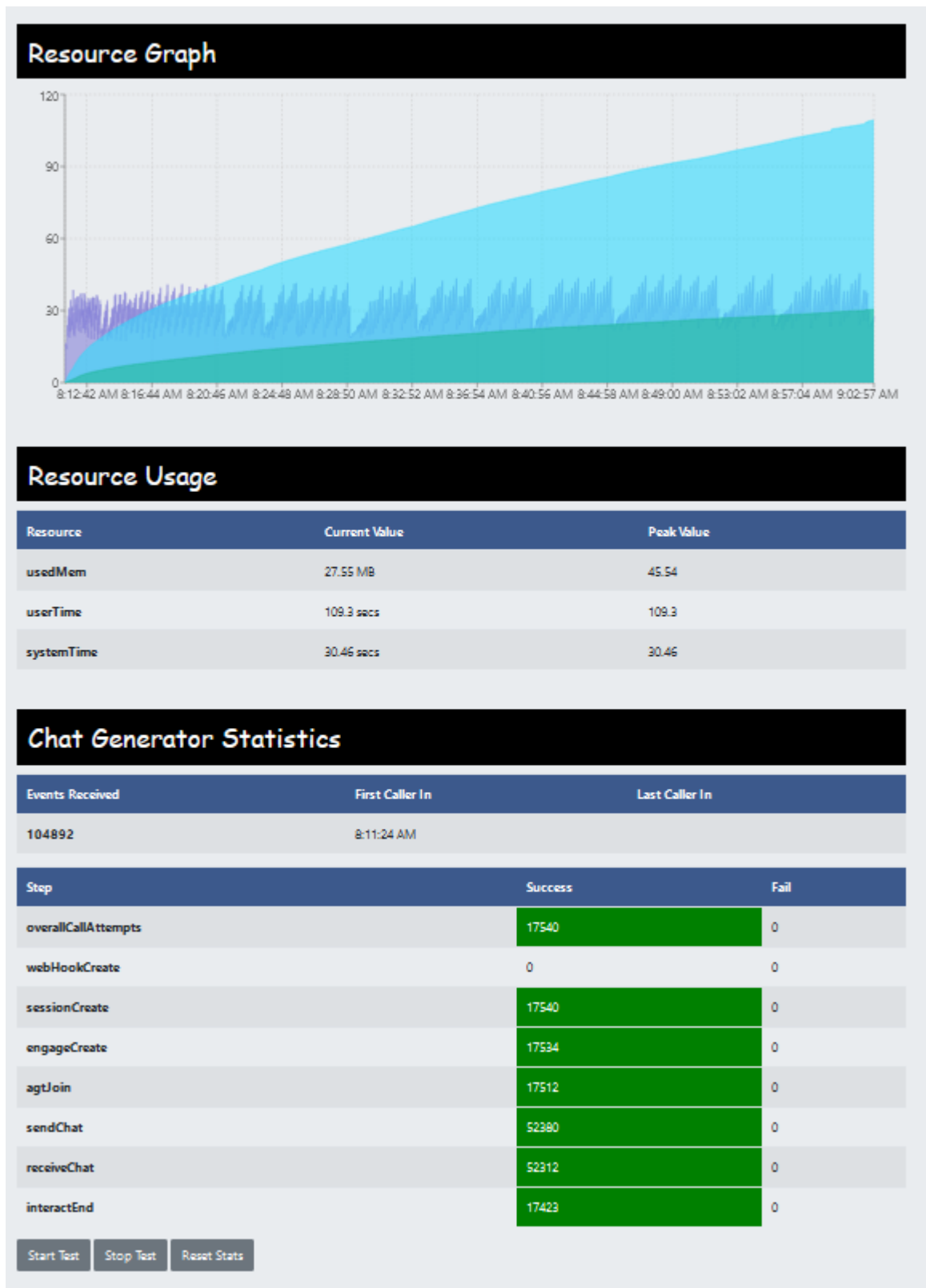


Figure 26 Fastify chat statistics at 17k interactions

The 20k chat interaction test completed with no failures.

Resource Usage		
Resource	Current Value	Peak Value
usedMem	29 MB	45.54
userTime	128.19 secs	128.19
systemTime	35.42 secs	35.42

Chat Generator Statistics		
Events Received	First Caller In	Last Caller In
120000	8:11:24 AM	9:18:22 AM

Step	Success	Fail
overallCallAttempts	20000	0
webHookCreate	0	0
sessionCreate	20000	0
engageCreate	20000	0
agtJoin	20000	0
sendChat	60000	0
receiveChat	60000	0
interactEnd	20000	0

Figure 27 Fastify chat statistics at 20k interactions

6.4 Comparison Summary

There were no significant differences in **memory usage** between Fastify and Express for this 20k test. Both frameworks had a peak memory usage of around 45M. The **CPU user time** recorded for express was 145secs, for Fastify this was 128secs. Both tests took the same time to complete, approximately 1hr 07mins.

To reduce the overall test time and stress the web frameworks further, the next step would be to introduce a load balancer in front of the Mock server to alleviate any bottlenecks it may have with event handling. As the performance tests were running, there was a noticeable decline in events received per second at the chat generator frameworks. This may have been as a result of the slow processing of events by the Mock server.

Performing an in dept benchmark analysis between Express and Fastify is outside the scope of this project. The aim of this project is to build a full stack test framework for a chat generator, including a CI/CD pipeline for building and deploying applications into Docker containers. This test harness is to allow development teams implement and test new features in their local environment with low level traffic to the chat generator in a containerised environment.

Once the new feature is tested, the chat UI and chat generator are handed over to the test teams where they will deploy the applications onto bare metal servers for large scale performance testing.

The short framework comparison test above also demonstrates the benefits of a modular architecture implemented on the nodeJS chat generator. A single line change is all that is required to change from using an Express Framework to Fastify. A similar test could be performed relating to other aspects of the chat generator. One example is when a new set of APIs are being implemented and tested against a different Contact Center/Chat solution. The chat-service.js file which contains all APIs used to interact with the Mock server could be easily replaced with a new set of APIs.

7 Problems Encountered

7.1 Displaying Realtime chat statistics in an efficient manner

During Sprint 14 the ability to view the chat generator statistics from the UI was delivered. Initially I was using a custom React Hook to fetch the statistics every 5 seconds

```
const useChatStats = () => {
  const [chatStats, setChatStats] = useState(null);
  useEffect(() => {
    setInterval( () =>
      getStats().then((stats) => {
        setChatStats(stats.data);
      }
    ),5000);
  }, []);
  return [chatStats, setChatStats];
};

export default useChatStats;
```

This however was an obvious inefficient way to retrieve data from the chat generator, in Sprint 16 I decided to investigate alternative ways to retrieve this data in a more efficient way i.e., websockets. WebSockets allow for a higher amount of efficiency compared to REST because they do not require the HTTP request/response overhead for each message sent and received. When a client requires ongoing updates about the state of the resource, WebSockets are generally a good fit. In Sprint 17 I implemented the Socket IO library on the client side (React UI) and the server side (NodeJS) to address the inefficiencies.

7.2 Upgrading Jenkins container

The Jenkins version deployed inside the Docker container was outdated, updating this from the Jenkins dashboard was failing as my environment did not have the relevant certificates installed to fetch the update over HTTPS. As a temporary workaround I had to update the Jenkins Update Center URL to use HTTP instead.

```
<?xml version='1.1' encoding='UTF-8'?>
<sites>
  <site>
    <id>default</id>
    <url>http://updates.jenkins.io/update-center.json</url>
  </site>
</sites>
```

To do this I modified the above hudson.model.UpdateCenter.xml locally and replaced the Update Center file inside the Jenkins container using the following command (the text in bold is the Jenkins Container ID):

```
sudo docker cp hudson.model.UpdateCenter.xml a2b9d8da9ce0:/var/jenkins_home/hudson.model.UpdateCenter.xml
```

The above change resolved my issue with the Jenkins upgrade.

7.3 High socket count open to Fastify during performance test

As discussed in **Section 6**, an unexpectedly high number of open network connections from the Mock Server to the Chat Generator was observed. This was only detected when Fastify was integrated on the chat generator to handle incoming events. This resulted in a network communication outage between the generator and the mock server when the number of events reached approximately 28k.

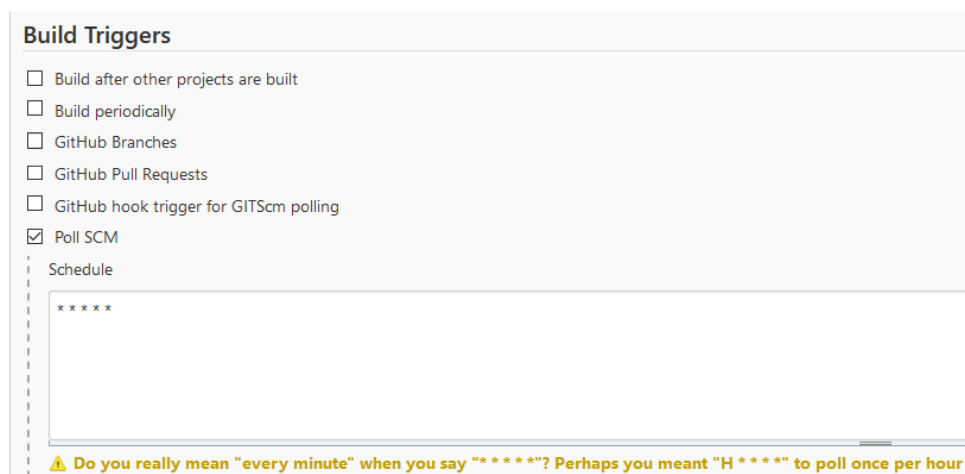
After some further investigation I found that the `'reply.code(200)'` method did not send a 200 OK back to the mock server. Therefore, the network connections were not closing, which led to a network communication outage. As discussed earlier when I changed the reply method to `'reply.send(payload)'`, a 200 OK was returned to the Mock CC and the issue was no longer seen.

7.4 Jenkins Deployment

Initially I deployed Jenkins onto an ubuntu server directly (not inside a docker container). However, I experienced issues deploying and connecting to the front/back end applications on the ports I had exposed. After further research I decided to focus on deploying Jenkins itself within a docker container. It is now possible to create and deploy containers for the back-end chat generator and React front end from within the Jenkins container

7.5 Jenkins Build Triggers

Build Triggers allows you to select when you wish to trigger your pipeline. My preferred option was to use *GitHub webhook trigger for GITScm polling* where the pipeline would be triggered when changes were pushed to my repository. However due to network constraints between my SUT and Github when using the webhook callback I opted for the *Poll SCM* trigger. With this option selected, Github is polled periodically for new commits to the branch specified. This is not a very efficient way to implement the build trigger however for demo and test purposes only I opted for the Poll SCM trigger.



Build Triggers

- ☐ Build after other projects are built
- ☐ Build periodically
- ☐ GitHub Branches
- ☐ GitHub Pull Requests
- ☐ GitHub hook trigger for GITScm polling
- ☒ Poll SCM

Schedule

⚠ Do you really mean "every minute" when you say "*****"? Perhaps you meant "H*****" to poll once per hour

8 Reflection

8.1 Project development path

The early days I knew from early on I would be looking to develop a chat generator using nodeJS however I was unsure as to where I would end up with it. I held some meetings with my supervisor and we came up with an initial project idea:

Express V Fastify: Find out which web framework deliver better performance of the chat generator.

Step 1: Identify the breaking points of both frameworks

Step 2: Investigate technologies that could address the break points. Would a Load Balancer address limitations? Would Fastify be more performant than Express and avoid the need for a Load Balancer (also a saving on hardware/resources). These were the questions I was initially looking to try to answer.

I focused first delivering a functioning chat generator using Express. How was I going to track each of the potential 20k chat interactions, how do I wait for an event that may never arrive at the chat generator or that may arrive at irregular times, how do I handle an event that never arrives, how do I identify a failure and provide that detail back to the user, how will the user update test parameters or start and stop a test. Getting started was the most difficult, my design had to be planned from the outset, so I arranged some high-level meetings with a senior engineer. It was through these meetings that I got a better appreciation and understanding of the value of using Maps/Objects to store data. The plan was to store each user engagement ID as the key in the Map and the value would contain all the relevant data for that engagement. This would allow me retrieve details for any chat interaction once I had a reference to the engagement ID.

To overcome the issue with dealing with events that may not arrive within an acceptable time or arrive at all I decided to create a promise that I would resolve when an event arrived, allowing the next step in the flow to execute.

To handle events that may not arrive during a predefined time frame I found a promise method called 'Promise.race' that would allow me to 'race' the event arriving against the acceptable time frame the user specifies at the start of the test. If the time elapsed before the event arrived, I would register this as a failed step and the chat interaction would end immediately.

With this in place I moved onto swapping out Express for the Fastify framework. After further discussions with my supervisor we discussed the possibility of moving away from the benchmarking idea mainly due to the fact that it may require a large number of benchmark scenarios in order to identify a clear difference in the performance of both frameworks.

Cloud Serverless Functions: Attempt to move the chat generator code to serverless functions in the Cloud

After further discussions with my supervisor we thought moving my existing chat generator code to the Cloud using serverless functions. This would mean hosting the existing chat generator functions into a managed cloud infrastructure. This would enable me to write less code and avoid the concern of deployment and maintenance.

I did some research on Azure functions and found that the application could scale on demand. As requests to the app increase, Azure functions would meet this demand with as many resources and function instances necessary.

After the initial analysis I decided not to go this route for two reasons, firstly the mock server I was testing against was deployed in an isolated network within my work place, this would result in extra tasks outside the scope of the project that I did not feel was going to benefit my learning and project implementation. Secondly, I wanted to focus on technologies that I knew were in demand in my workplace and that I would greatly benefit from while working on this project. This led me to my current project implementation below.

The chosen one Building a UI for the chat generator was a necessity, I really enjoy working with React during the ICT Skills module, so I started to revise existing course material and the React project I developed during semester 3. I found our React project to be a fantastic reference for the UI work that I was about to undertake with this project.

While working through the project I received several calls from recruitment agencies relating to a devOps role, the trigger for these enquiries were as a result of projects I documented on my LinkedIn profile, these projects were completed during the devOps module. In my current workplace the CI/CD teams use a Jenkins pipeline, I had a basic high-level idea of what Jenkins was but have always wanted to understand it on a deeper level. I felt introducing this technology to my test harness would not only improve the development, deployment, and test times for my project but that it would also give me a deep insight into this technology and how it operates, which will benefit any future employment in this area.

The above is a high-level diary of how I eventually decided on developing a back-end chat generator that would be driven and reported on via a React UI and deployed using a Jenkins CI/CD pipeline.

8.2 Independent Learning

The following lists out some of the new technologies I researched and gained knowledge of throughout this project

- Fastify Web Framework
- Socket IO Client front-end (React)
- Socket IO back-end (NodeJS)
- Promise Race (returns a promise that fulfills or rejects as soon as one of the promises in an iterable fulfills or rejects)
- Process Object NodeJS (provides information about, and control over, the current Node.js process)
- Webhooks (user defined HTTP callbacks which are triggered by specific events)
- Recharts (A composable charting library build on React components)
- Docker, Docker Containers
- Jenkins CI/CD
- Dockerfile, docker-compose, Jenkinsfile

8.3 Future work

The list is endless with where I would like to take this test harness. Potential areas are discussed below

There are several REST API functions implemented on the chat generator that have not yet been implemented by the Mock CC server. Once these features are implemented on the Mock CC they will then be integrated into the current chat interaction flow. A summary of the current APIs already implemented on the chat generator but not yet available on the mock server are:

Create Webhook API → Webhooks are user defined HTTP callbacks which are triggered by specific events. Whenever that trigger event occurs in the source site, the webhook sees the event, collects the data, and sends it to the URL specified by you in the form of an HTTP request.

In the context of the chat generator, when the generator is started, the createWebhook API will send a request to the Mock CC to register the webhook. Within the body of this request there will be two fields, the callback URL (URL of the chat generator framework) and the type of event the chat generator is interested in. As chat interaction traffic is running, the Mock CC will send the requested events back to the URL specified in the webhook request. These requests will then be handled by the Express/Fastify framework.

Delete Webhook API → The webhook must be de-registered/deleted from the Mock server, this API call will perform this task once the test has completed.

Terminate Session API → Should the chat generator wish to terminate a session, this API can be called.

getParticipantID API → In a future release the Participant ID will need to be passed as part of a chat message exchange with the Mock server, this API call will retrieve the relevant participant ID when needed.

Other areas for future development:

- Replace useState hook inside chat parameter component with useReducer hook to store chat parameter state
- Refactor the 'index.js' file i.e., move several chat functions into their own module
- Migrate the mock server from the isolated network and include in current test harness to aid transition to cloud architecture i.e., Cloud Serverless Functions
- Write a new back end generator in Python or GoLang for personal development
- Move application deployment from containers running on Docker to a microservice running on Kubernetes
- Investigate other technologies that would help improve the benchmark results e.g., introduce Load Balancer, multi-threading to back-end
- Add unit tests as part of CI/CD pipeline step

9 References

9.1 Jenkins / Docker

1. <https://medium.com/@schogini/running-docker-inside-and-outside-of-a-jenkins-container-along-with-docker-compose-a-tiny-c908c21557aa>
2. <https://stackoverflow.com/questions/35110146/can-anyone-explain-docker-sock>
3. <https://www.jenkins.io/doc/book/pipeline/getting-started/#defining-a-pipeline-in-scm>
4. <https://docs.docker.com/engine/reference/builder/>
5. <https://www.jenkins.io/doc/book/installing/docker/>

9.2 Chat Generator

6. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/race

9.3 React

7. <https://medium.com/forepaas/react-making-recharts-easy-to-use-e3d99d0641ba>
8. <https://www.youtube.com/watch?v=azvcvbeRZ08>

10 Appendix

10.1 Useful Docker Commands:

10.1.1 Copy files from ubuntu server to a Docker container running on ubuntu.

```
sudo docker cp <filename> <Docker Container ID>:<location on Docker container to copy file to>
```

10.1.2 Monitor the docker logs live using '-f' option

```
sudo docker logs <Docker Container ID> -f
```

10.1.3 Restart a container

```
sudo docker restart <Docker Container ID>
```

10.1.4 Execute a command inside a Docker Container from the ubuntu server CLI

```
sudo docker exec -it <Docker Container ID> <command to execute on container>
```

10.1.5 Access the bash CLI of a Docker Container

```
sudo docker exec -it <Docker Container ID> bash
```

10.1.6 Display what is being outputted from inside docker container

If you add in **-d** to detach then it runs in background eg 'sudo docker run -d -p'

Run live output: (good to test an image)

```
sudo docker run -p 3001:3001 react-front-end_chatgenerator
```

10.1.7 Remove docker image

```
sudo docker rmi <Docker Image name>
```

10.1.8 List all Docker containers

```
sudo docker container ls -a
```

10.1.9 Stop Docker Container

```
sudo docker stop <Docker Container ID>
```

10.1.10 Remove Docker container

```
sudo docker rm <Docker Container ID>
```

10.2 Pipeline deployment files

10.2.1 Jenkinsfile

The agent section specifies where the entire Pipeline, or a specific stage, will execute in the Jenkins environment depending on where the agent section is placed.

Any: Execute the Pipeline, or stage, on any available agent.

```
pipeline {
  agent any
  stages {
    stage('Test') {
      steps {
        echo 'In Test Stage'
      }
    }
    stage('Deploy') {
      steps {
        sh 'docker-compose build'
        sh 'docker-compose up -d'
      }
    }
  }
}
```

10.2.2 Dockerfile – Chat Generator

```
FROM node:12
COPY . /
#COPY package.json .
RUN npm install
EXPOSE 8001
#ENTRYPOINT [ "node", "index.js" ]
CMD [ "node", "index.js" ]
```

10.2.3 Dockerfile - React Front End

```
FROM node:12
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD [ "npm", "start" ]
```

10.2.4 Docker-compose – Chat Generator

```
version: '3.3'
services:
  chatgenerator:
    build: .
    container_name: chatgenerator
    ports:
      - "8001:8001"
```

Docker-compose – React Front End

```
version: '3.7'
services:
  chatgenerator:
    build: .
    container_name: reactFrontEnd
    ports:
      - "3000:3000"
    stdin_open: true
```