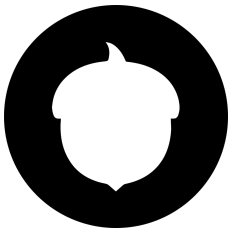
50.003 Elements of Software Construction

Cohort 3 Group 8 Final Report

Aaron Khoo 1002788
Wang Zilin 1003764
Jeremy Ng 1003565
Chen Yan 1003620



‘Acorn’ Service Platform

Built on Alcatel-Lucent Enterprise Rainbow

Executive Summary

Overview

Rainbow is a cloud communication platform providing unified communication (UC) services such as chat, audio and video calls (https://www.openrainbow.com/). Rainbow is frequently used by contact center agents to handle support requests from customers. Rainbow also provides SDK and API for developers to build custom UC applications. Contact center agents need a routing engine to organise incoming chat requests and voice calls from customers into queues and route them to available agents based on agent availability, skills etc.

Project Objectives

To develop a routing engine that routes incoming support requests (chat and audio calls) to the right agent based on agent availability and skills.

Deliverables

- A web page simulating a client's website where their users can request for support via chat or audio call.
- A routing engine that routes incoming chat and audio call requests to the right agent based on agent availability, skills etc

Proposed Solution

Our ‘client’ whom we are serving is Acorn, a technology company that designs, develops, and sells electronics, software, and online services to consumers. Our proposed solution will serve as a platform for their customers to seek assistance or enquiries on anything which is related to their company where agents will serve them via chat support or audio calls.

Our solution consists of four core components: a frontend web interface, an intermediary backend server, a database, and an admin dashboard. The frontend web interface will be the platform where customers can request support. The backend server will implement a routing-engine which will determine the connections between agents and customers. The database will be used to store agents and queue data. The admin dashboard will provide an interface for administrators to track the agent statuses and manage the system.

Implementation - Feature Overview

Github Repository Links

Support Platform: https://github.com/jeroee/esc-acorn-web
Admin Dashboard: https://github.com/aaronreulkhoo/esc_dashboard
[1]Backend: https://github.com/aaronreulkhoo/node_backend
Testing for Frontend/Backend: https://github.com/jeroee/esc-acorn-testcodes

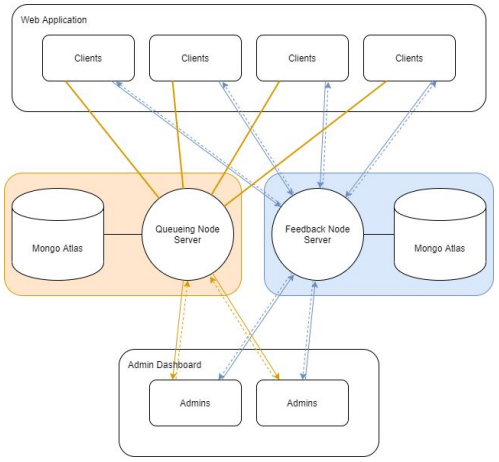
Application Links

Web Application - https://esc-frontend.netlify.app/
Admin Dashboard - https://esc-dashboard.netlify.app/

Overview

Our group has created our frontend Web Application in Vue which is served by a Node/Express server linked with a MongoDB instance. Notably, this is what is known as the MEVN stack, a single-language (Javascript) stack that is highly flexible, cloud-friendly, and very popular with small to mid-sized startups due to its inherent productivity.

In order to improve performance utilising free tiers on cloud platforms like Heroku, our group has decided to separate the functionalities of queueing (orange) and feedback (light blue) into their own subsystems. More of each component will be explained individually below.

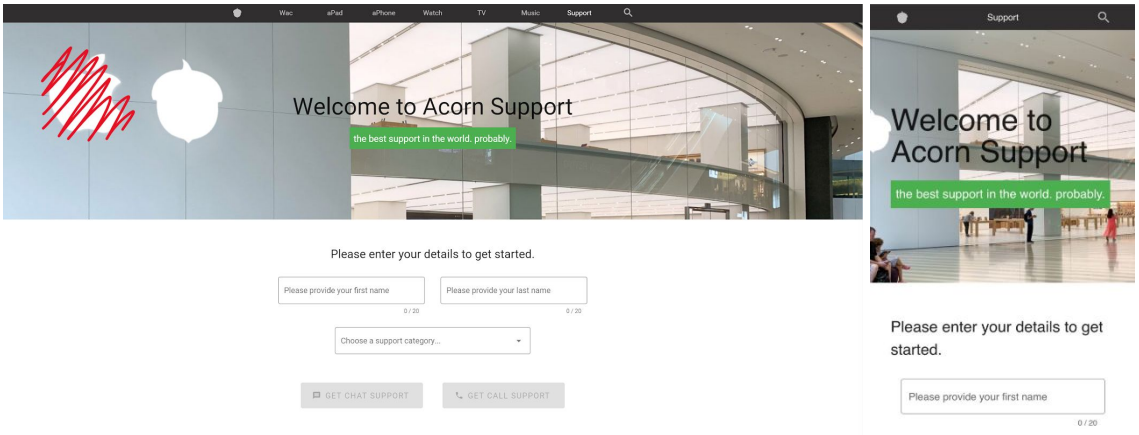


An overview of our system.

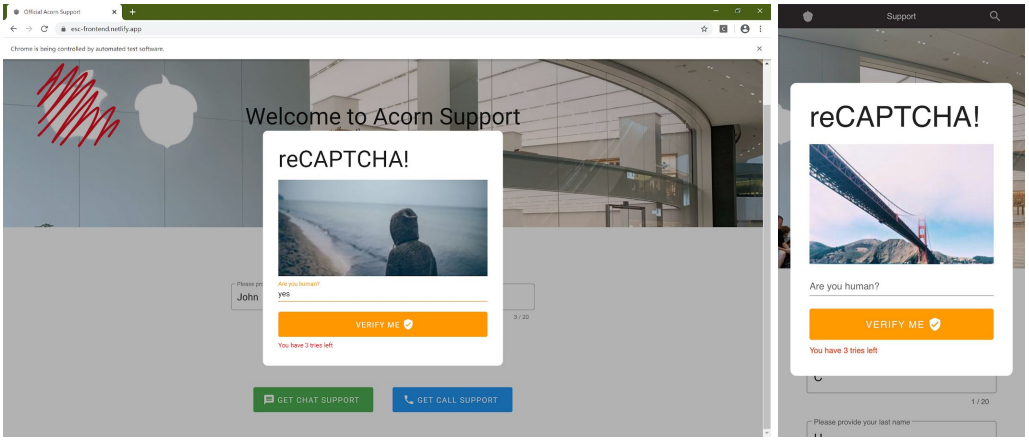
[1] *Queueing and Feedback server code can be found as server.js and rainbow_server.js respectively.*

Customer-Facing Web Application

Our Customer-Facing Web Application consists of four main pages (home, chat, call and feedback page). It is designed to provide service to all customers of Acorn to help them solve product issues and know more about Acorn. In the home page, after inputting the first and last name , customers are to select a category in which they need help in, and lastly to choose the type of support, chat or call. We mainly provide two functions , chat and call. You can choose whatever service types that you want.In addition , we also add a feedback page to listen to customers' suggestions and help us to improve agents' service quality.

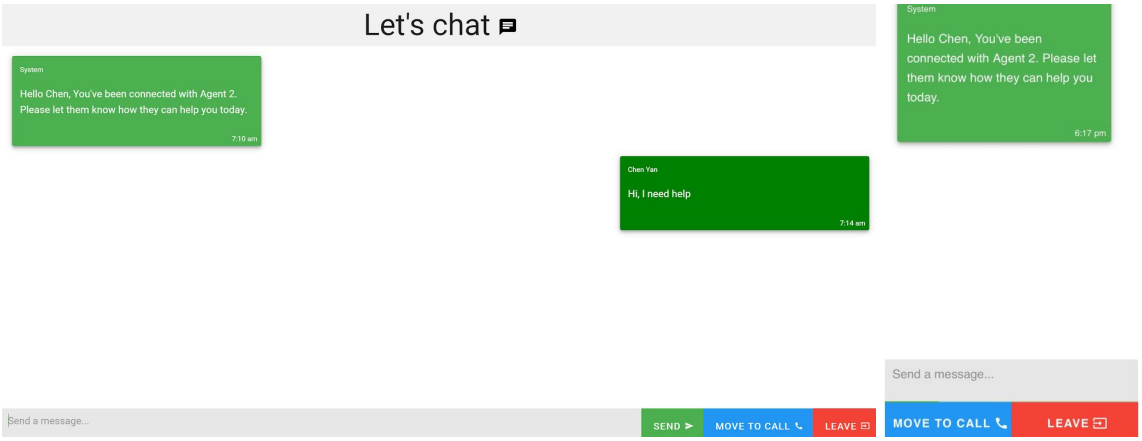


In the homepage , we added a ReCaptcha verification implementation to verify that all users using the system are humans. This implementation is used to protect websites against spam and abuse The goal is to stop interactive websites from being spammed by filtering out automatically generated input. Hence, in order to prevent this issue, we require every customer to clear the ReCaptcha test before they request for support from an Acorn agent.

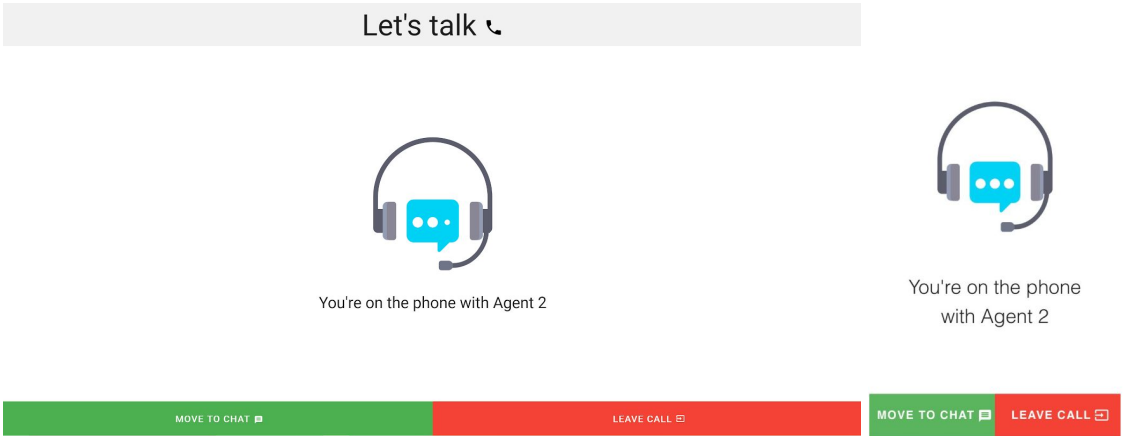


Customers will be able to chat with Acorn's agents to get support. For the chat page, this includes basic chat functions, writing messages, sending messages, moving to call and leaving. In order to provide a better service to customers, we consider that agents may

not really solve customer's issues if we only provide chat functions. Hence, in the chat page , we add a ' move to call ' button to help customers to access the call service.If you want to leave the chat page, you will go to the feedback page after you click the 'leave' button at the same time.

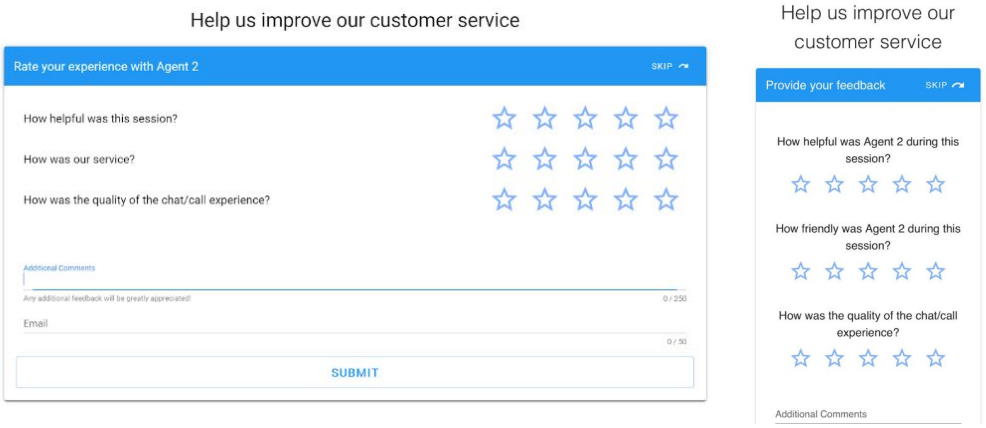


Customers will also be able to call with Acorn's agents to get support. For the call page, this includes basic call functions, moving to chat and leaving. In order to provide better service to customers, we consider that agents feel that it's better to solve the issue via chat. Hence, in the call page , we also add a ' move to chat ' button to help customers to access the call service.If you want to leave the chat page, you will go to the feedback page after you click the 'leave' button at the same time.



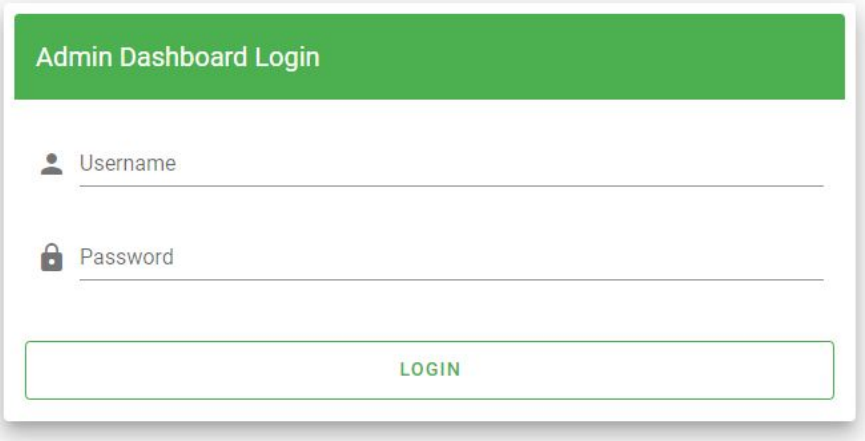
After customers have gotten Acorn's support, we set up a feedback page to listen to customers' advice to help us improve our agents' service quality. For the feedback page, this includes basic feedback form functions. First, we set three questions about our service quality that customers can use a rating rule to evaluate. Customers have the option to write additional comments to agents in the feedback form. Customers are also able to provide their email in case they would like Acorn to follow up with their feedback. In addition, if customers do not feel the need to provide feedback, we also

provide a ‘skip’ button to help them skip this step. The customer will eventually return back to the Home Page.



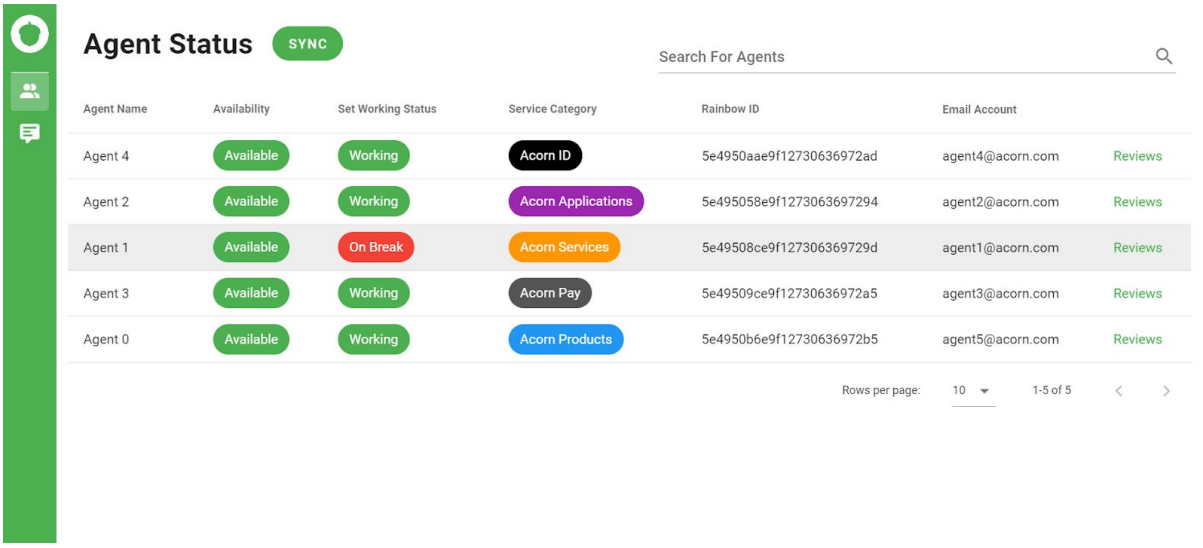
Admin Dashboard Interface

An Admin Dashboard is designed to keep track of activities and statuses of all support agents of Acorn. It mainly consists of the login page where only people with Acorn administrator privileges are able to access it and the dashboard where all agents’ activities can be observed. Similar to our customer-facing Web Application, the Admin Dashboard is also optimised to be mobile friendly.

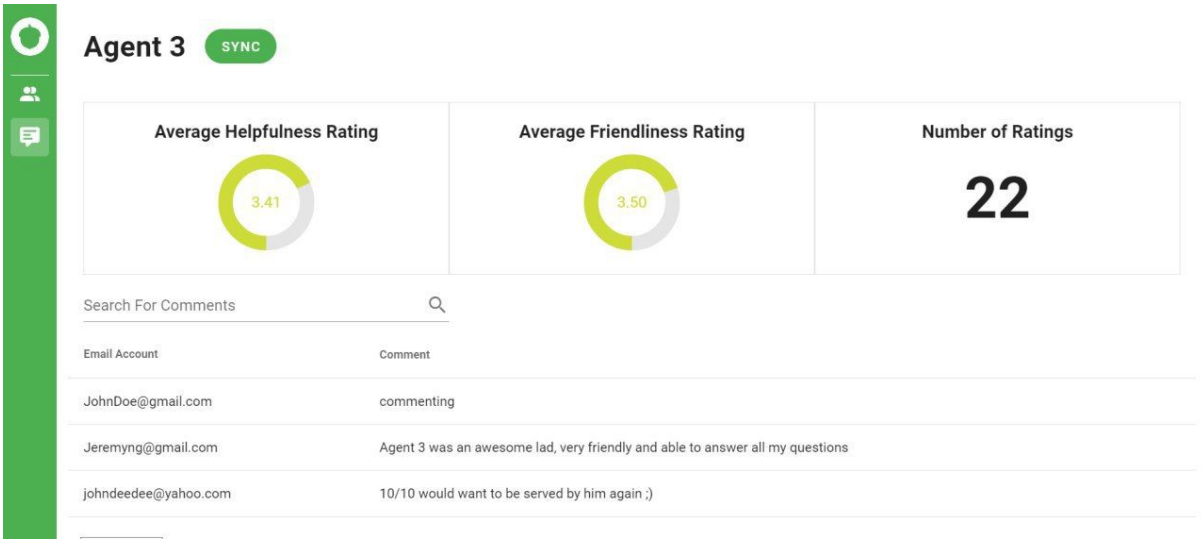


Administrators will be able to view all of Acorn’s agents and their respective details upon logging in to the Dashboard. This includes their current availability, working status, service category which they are tagged in, rainbow ID, and their email account. The availability status of the agent will be set to true only if the agent is online and currently available. If the agent is offline, away, or currently in connection with a customer serving chat or call services, the availability status will be set to false. The administrator would also have privileges to set the working directory of the agent by just clicking on the

status of the agent itself. This will prompt a confirmation the status can be toggled between ‘On-break’ and ‘Working’. This allows the agent to take short breaks even when he is available. When the status is ‘On-break’, no customers will be able to connect with the agent even though he is currently available. They will only be able to do so if the agent’s status is set to ‘Working’. Currently, a simplified solution is being employed, in that agents in one category must still engage whoever is left in their category regardless of whether they are on break or not (since customers have been waiting), however more nuanced logic or systems such as ticketing or session continuation can be integrated later on.

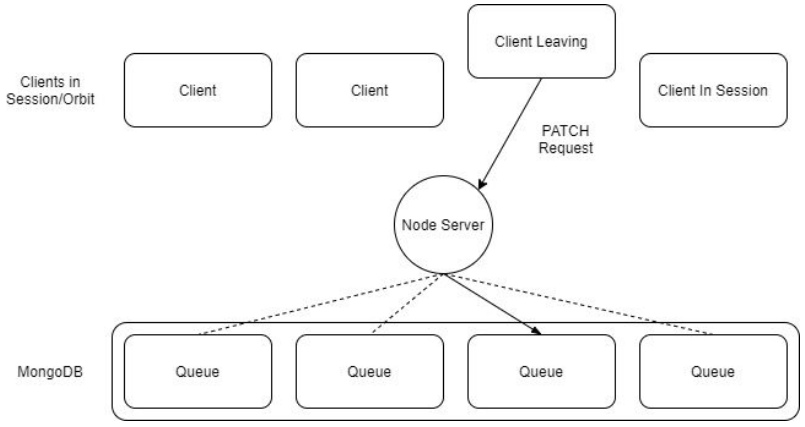


Administrators would also be able to review every agent by selecting at the reviews column at the right side of the dashboard. In the reviews section, the collated ratings and additional feedback which are provided by the customers can be viewed. Customers who left additional feedback also provided their email to allow the company to follow-up on it. Such data provided by the customers which is shown on the dashboard will allow the company to improve the agents’ overall customer service.



Queueing Node Server

The Node server that serves the frontend clients was what we started with before we decided to branch our functionalities out, and is notable due to its stateful operation utilising web sockets. Although our team started with a RESTful API (stateless) approach and had fully implemented the queueing logic, we encountered many problems such as high latency (each request requires a new HTTP connection which runs on TCP creating wasteful overhead each time).



The last PATCH request was critical for a RESTful system to remain error free from disconnecting clients.

Instead of utilising a complex cleanup mechanism that could result in further unexpected behaviours or browser-dependent fixes, our team decided to redesign our mechanism from scratch using web sockets. Notably, this:

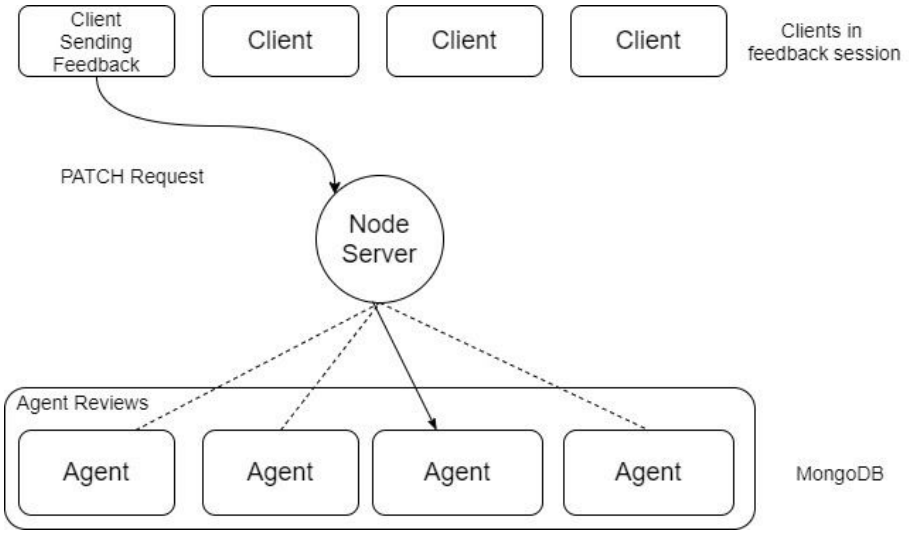
1) Reduced latency between important state changes leading to better performance.
2) Removed the problem of clients disconnecting without the server knowing.
3) Allowed us to move to an event-based system, simplifying the logic on both the client and server side.

The logic for the sequence of events that occur when a client connects is detailed in our sequence diagram.

The API endpoint: https://esc-acorn-backend.herokuapp.com/

Feedback Node Server

As we have a two-way system to support queueing and feedback respectively, feedback execution can be done independently so it will not delay the execution for the Queueing server. Hence, it is implemented using a feedback API.



The node server is linked with MongoDB, enabling the real-time data update. The database stores received reviews and basic public information of agents (e.g. Agent Rainbow ID). Once a feedback is filled up and the PATCH request is sent, the certain agent will be found in the database with rainbow ID and the data (e.g. rating, comment, client's email) will be updated.

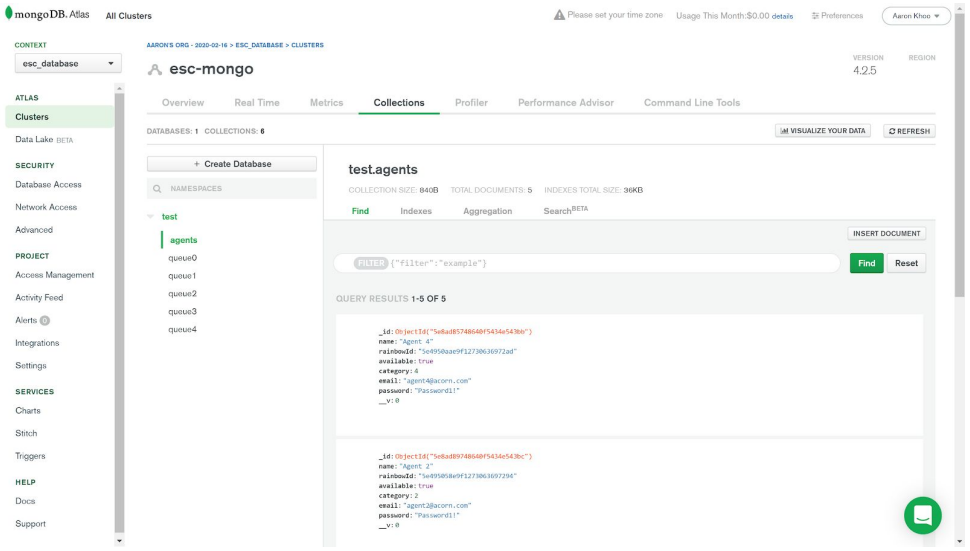
For the PATCH function, the path parameter is /review, and we use query string parameters to send requesting data. The feedback contains a total of 5 parameters (3 ratings ,additional comment and client's email). Agent's rainbow ID is needed for locating the agent in the database which can be extracted from the agent-client connection. Additional comments and email are optional to fill in. To prevent SQL injection, some characters are set as invalid in comment and email inputs. We also add an authorization header checker and jwt verification as middleware functions to prevent unauthorized attack.

The API endpoint: https://still-sea-41149.herokuapp.com/

MongoDB Atlas

As shown in the overview, we utilised two free clusters of MongoDB Atlas, each of which were used as the persistent data structure in their own separate subsystems. The choice of this versus a hosted Heroku server was simple:

1) Higher throughput and low latency scoped specifically for database read and writes.
2) Status monitoring, automated failure recovery, and clear performance measures.
3) Real-time data provisioning and visualisation, allow us to debug our system logic against the source of truth easily.
4) No fiddling around with command lines. Given our limited man hours for this project, any time saved was more time to put into improving our main deliverables.



Utilising a NoSQL database was highly beneficial in our project, allowing us to manually add on later document fields as required as our scope expanded and extra functionality was added in.

In terms of the structure, having such flexibility over our document collections allowed us to programmatically increase the number of queues that we could use in our multi-queue system. This would theoretically lead to much better scalability.

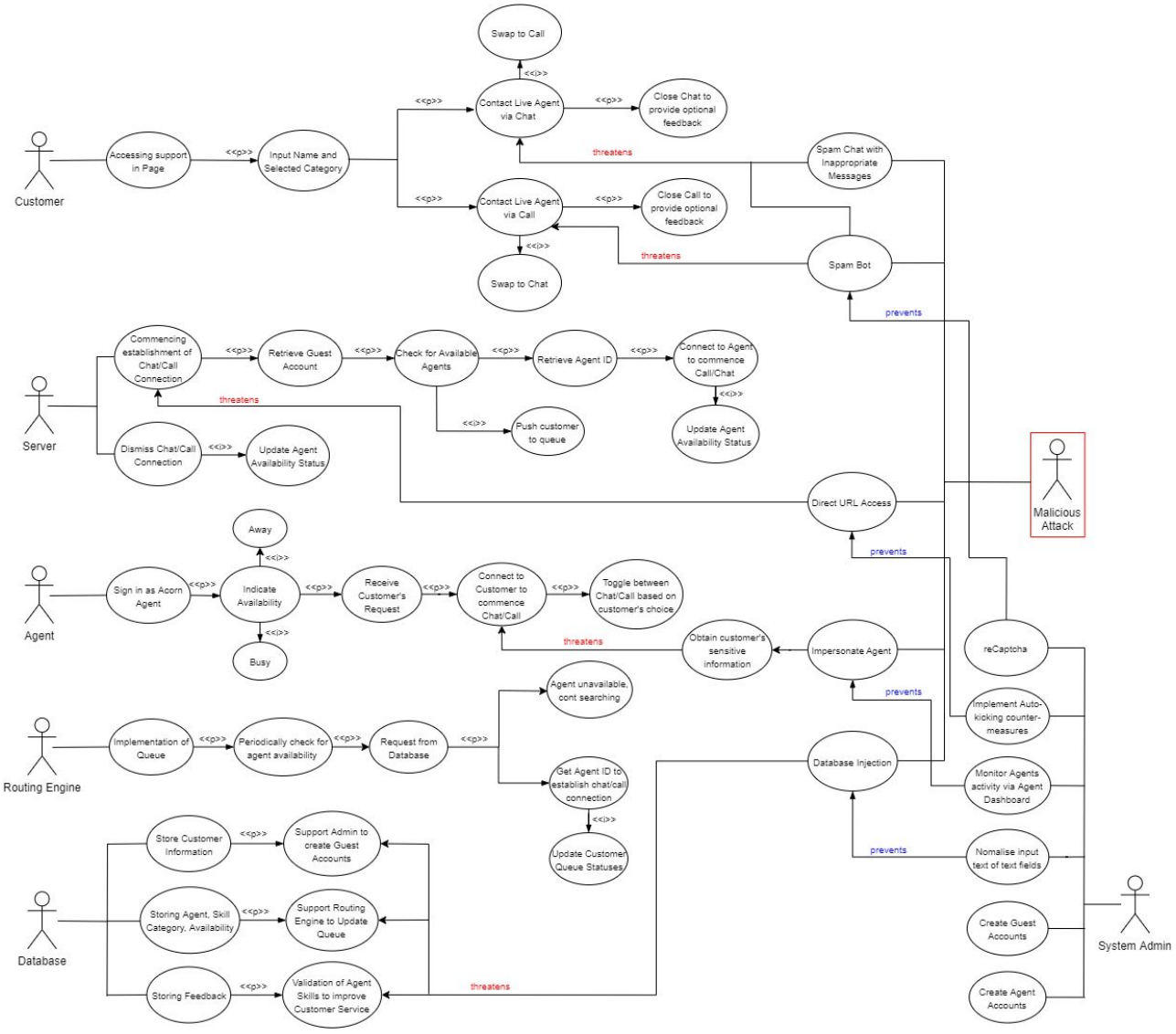
While Mongo is schema-less, discipline was enforced for our APIs utilising Mongoose, a schema-like validation which defines the inputs to every collection and provides useful wrappers for different MongoDB commands. Aside from preventing insidious activity like noSQL attacks, it allows us to catch errors even before the data reaches the database.

Code Structure - UDL Diagrams

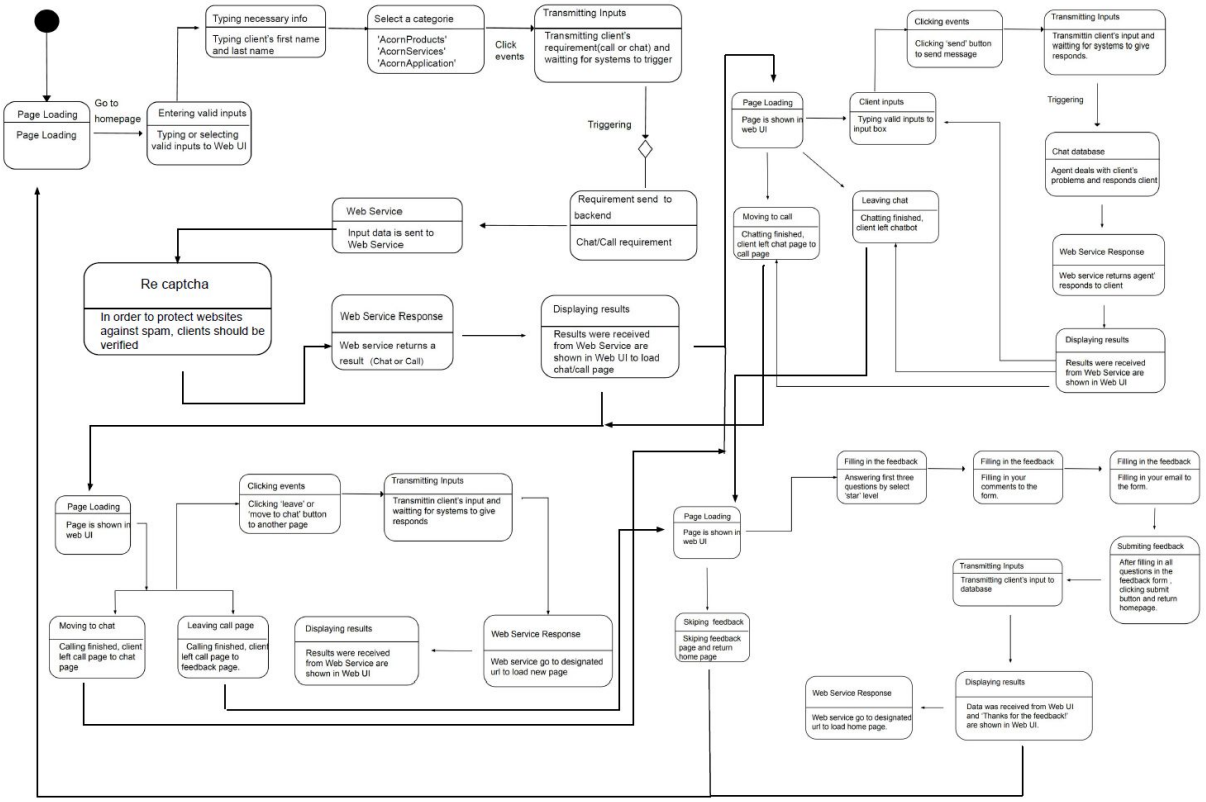
Below are our updated UML Diagrams. Our old diagrams can be found in our appendix.

Use Case Diagram

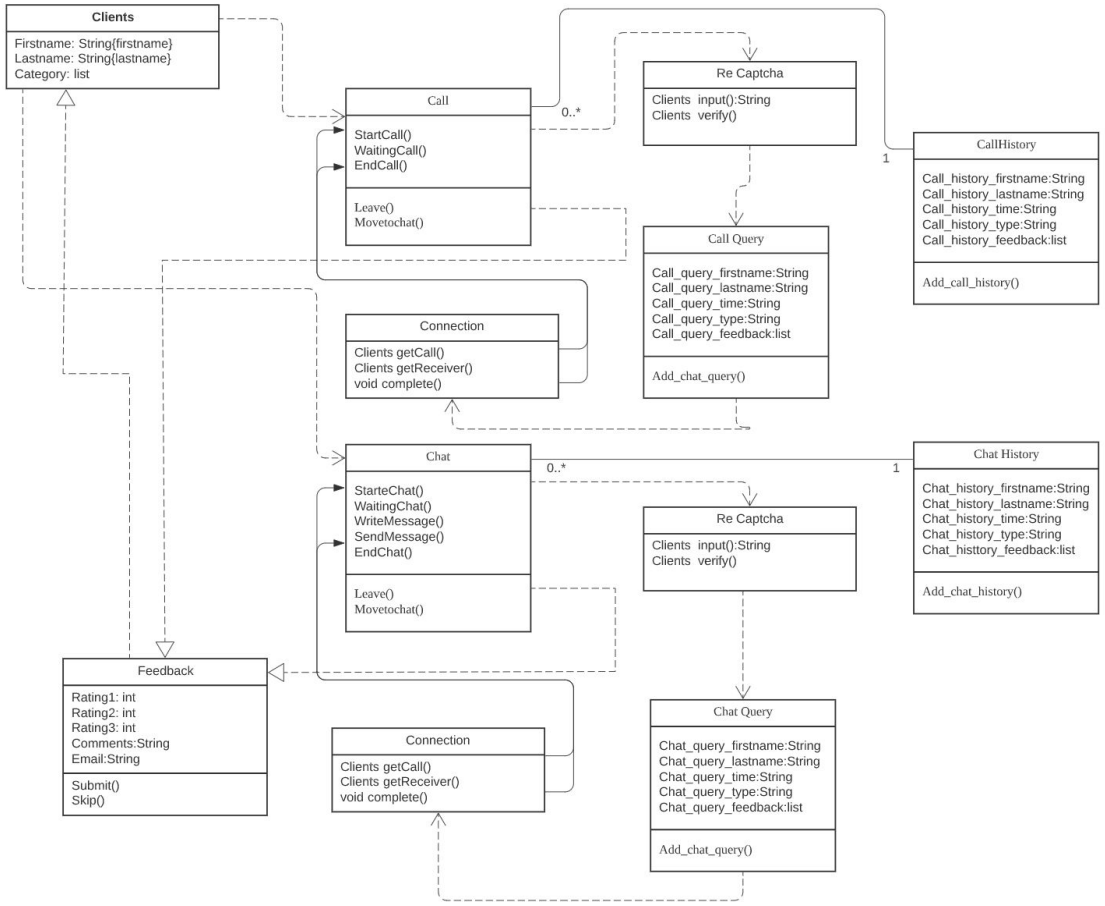
Our use case diagram comprehensively describes how each actor interacts with the system in every possible way. Potential misuse cases are included to show how adversarial scenarios can affect the system and how our implementations in our system can deny such scenarios from happening.



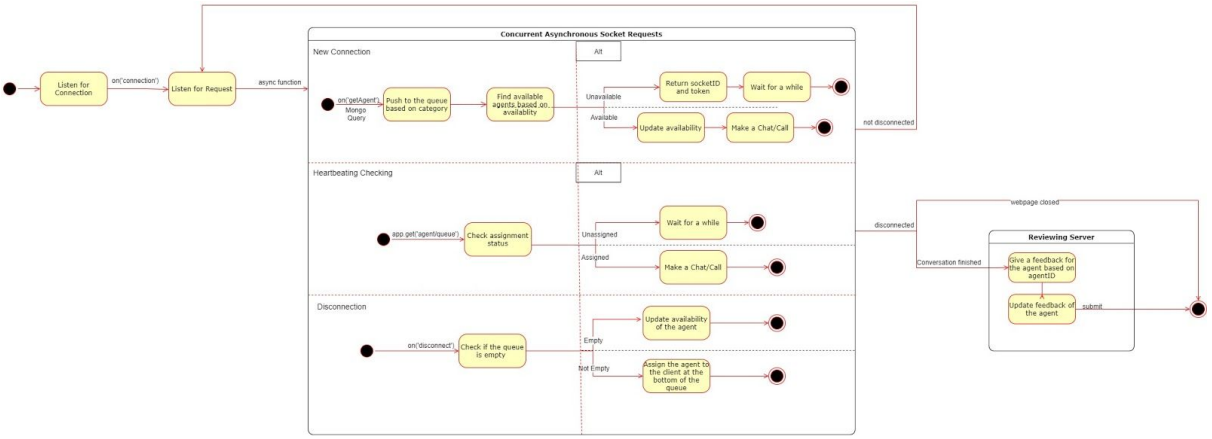
Frontend State Diagram



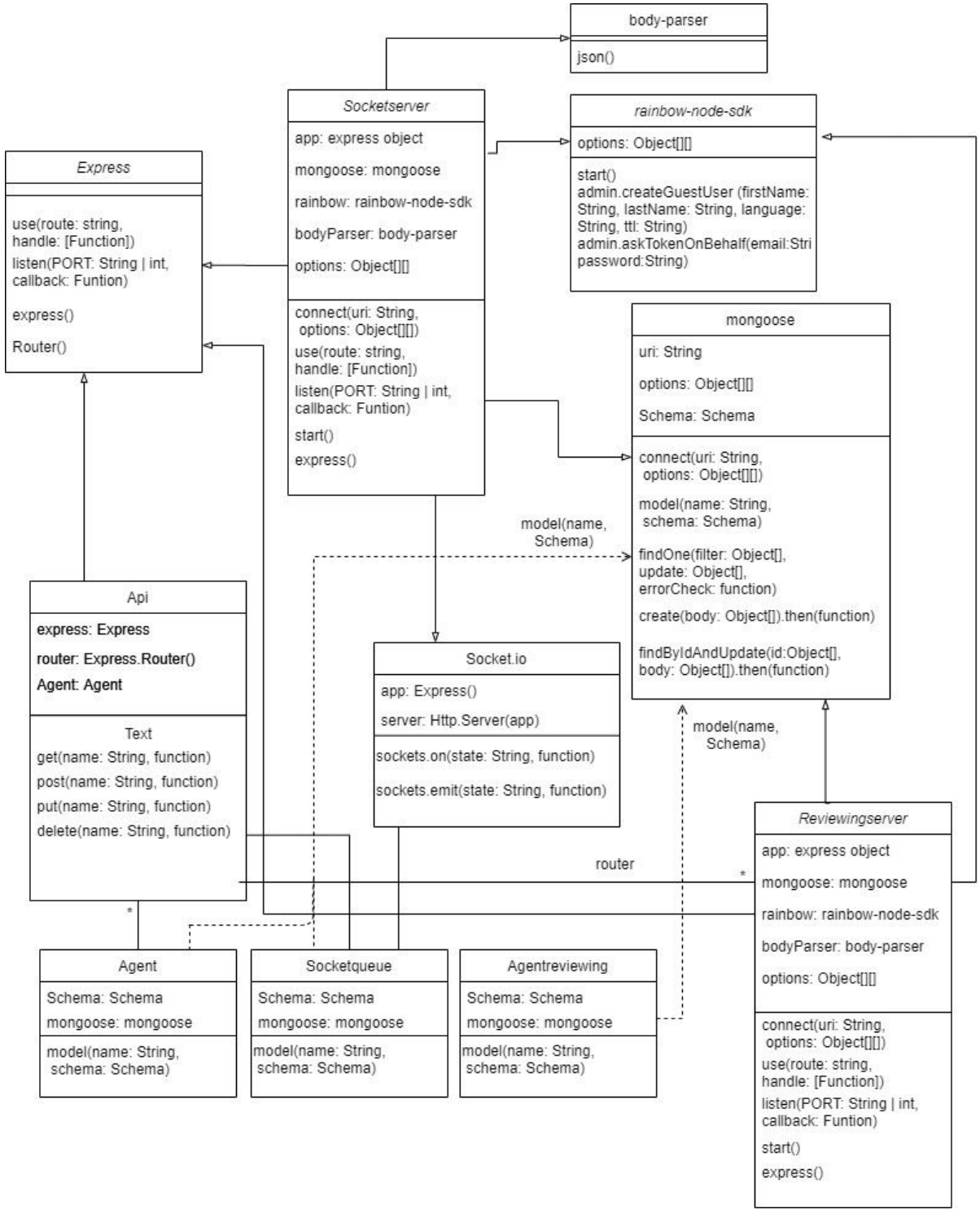
Frontend Class Diagram



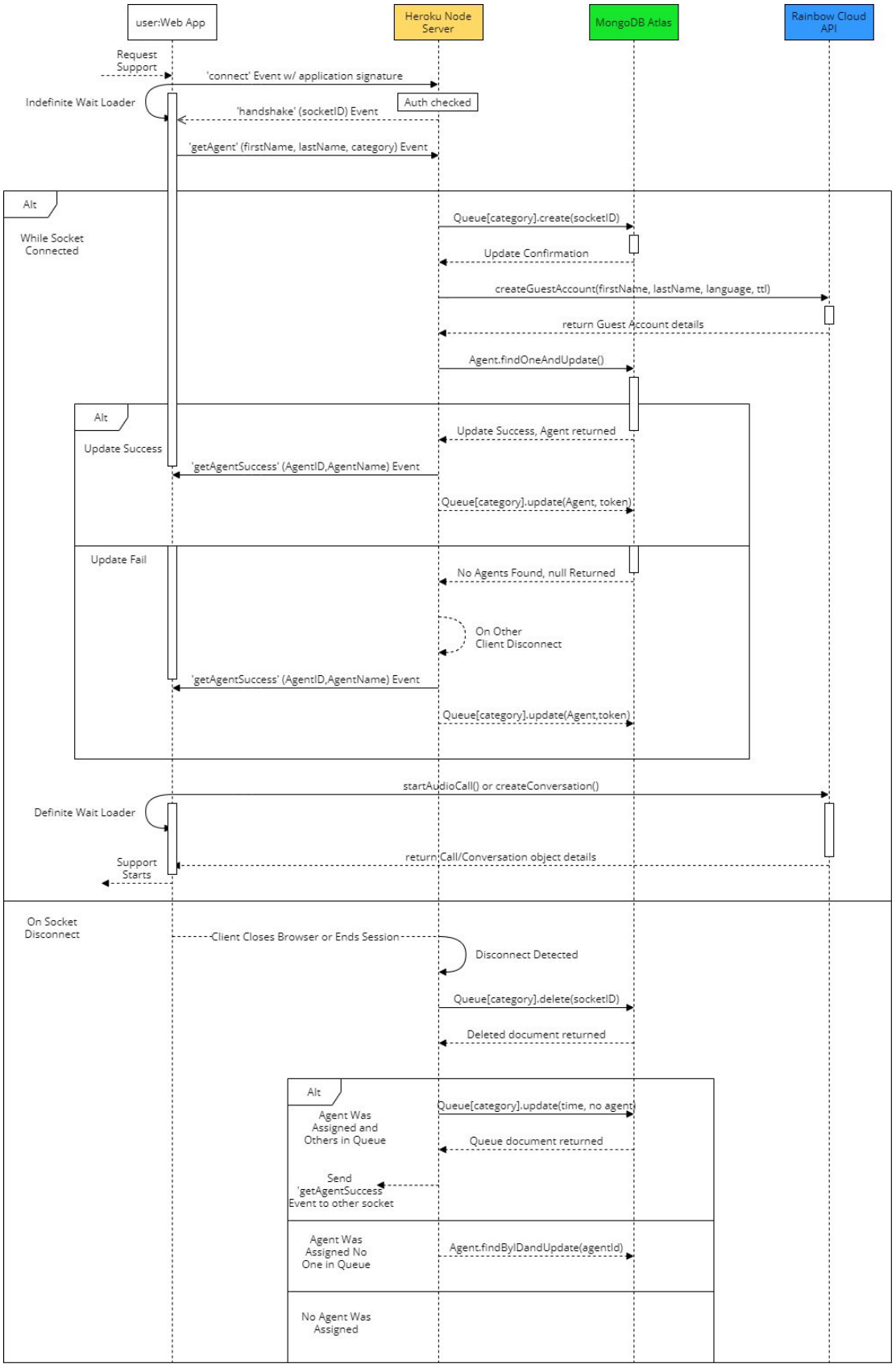
Backend State Diagram



Backend Class Diagram



Sequence Diagram



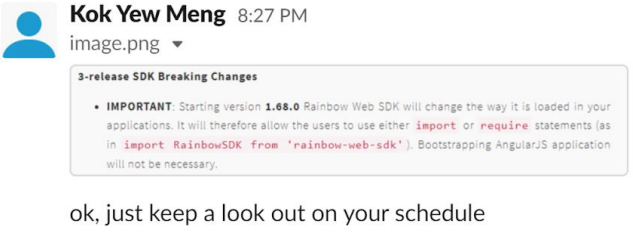
Implementation Challenges

Engineering Challenges

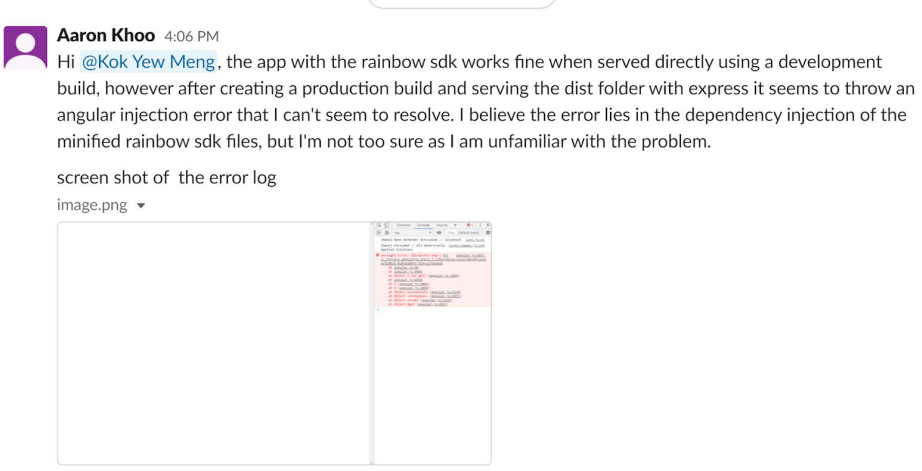
Rainbow SDK Webpack Compatibility

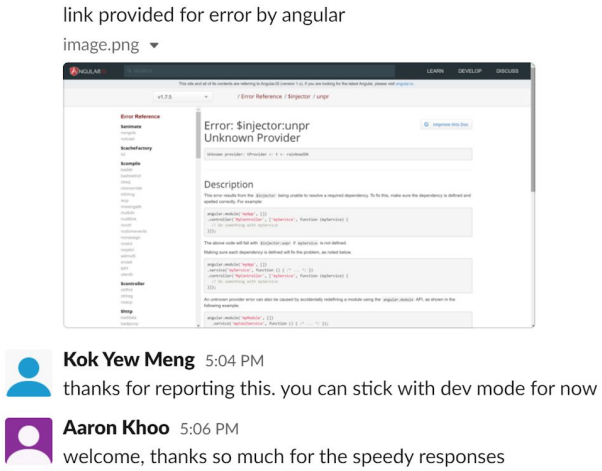
The latest version of the Rainbow SDK did not in fact support Webpack when we started the project, a module bundler that the Vue community utilises to implement several useful packages such as Vuetify. Instead, the SDK utilised legacy ES dependency injection through Jquery which was not compatible with the way Webpack compiled modules.

After communicating some of our attempts to fix this to the Alcatel engineer, he actually relayed out that a Webpack compatible version would in fact be released during recess week. As such, our team was able to continue to use Webpack which sped up our development.



Because of this, we were also able to catch an injection error that we found while attempting to host our website using a production build compiled with Webpack. We believe that highlights of such errors will certainly help Alcatel boost their platform as they seek to move it out to the mainstream developer community in the future.





Tethering Client Disconnections To State

The main implementation challenge we faced was whether to use web sockets for our connection. We had a fully 'working' polling RESTful procedure already written that often created problems in our system due to unexpected client disconnections , and were unsure whether to switch. After doing experimentation, we performed the switch and rewrote our logic from scratch, reaping the aforementioned benefits of this.

Due to us rewriting this, we also had to create a new test suite to stress test the system, which we wrote in mocha and is detailed below in our testing section. Notably, we were able to write much more scalable and robust tests to complement the new system, and have managed a load of 20 clients comfortably without any failure, something we were unable to guarantee using long polling.

Initialisation of Web/Node Rainbow SDK

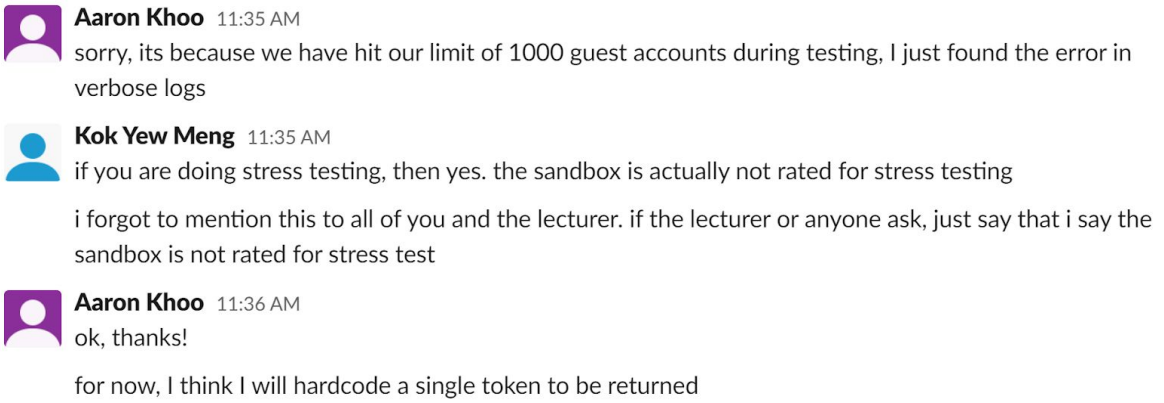
The Rainbow SDK is a fairly large object in memory that takes a substantial amount of time to be loaded, both on the server side and on the client side. For the server side, this can cause errors due to Heroku's uptime limits for free dynos, throwing errors that cannot be prevented but only caught during this initialization from sleeping status. Unfortunately, there is no way for us to circumvent this other than preparing the server before any demonstration or period of usage.

For the client side code, the Web SDK is built in Angular, and one problem we faced because of this was our escalation/de-escalation feature of moving from chat to call and vice-versa, where problems would be raised if switches occured too quickly or if the user hit the back button. To prevent this unfixable error, we used hard route guards along with sleep timers to make sure we would avoid unwanted behaviour from end-users.

Rainbow Guest Tokens

Rainbow API guest accounts just like our authentication protocol uses JSON web tokens, however due to our rigorous testing both utilizing the platform as well as our automated backend testing, we discovered that our logs were producing errors that were being thrown but not caught within our code (undefined). Suspecting the Rainbow SDK, we found out that it was indeed due to the fact that our sandbox account had hit a maximum limit to the number of temporary guest accounts.

This occurred twice before we could properly identify the error, and eventually we just hardcoded a token to carry out more tests, even doing it for our final presentation. In total, we have made over 4200 guest accounts which is an indicator of how much testing our team did both manually and using automated tools.



Testing Challenges

Frontend Testing:

- It is a challenge to plan for robustness and functionality tests as we need to come up with many different potential ways that might cause the page to crash or result in some sort of failure and we might not even cover everything. Much brainstorming and trials were conducted to come up with potential ways of breaking the system.
- Due to the lack of security knowledge, our range of tests and level of tests will only be limited to our current technical skills and competencies, hence only certain tests to prevent simple malicious attacks were conducted.
- Implementation of Recaptcha verification System might prevent us from utilising Selenium to conduct frontend automated testing properly since Recaptcha verifies human users when using the system, the use of Selenium might cause certain system conflicts.

Backend Testing:

- We encountered a challenge when we wanted to design a test that goes through the full journey of a user. We used Postman to do testing for the Router API functions, but the testing on Postman can only send one request each time. We solved this problem by transferring the queueing system from the Router API to socketing executions. Using the sockets, the server can send requests to and listen requests from a client so the full journey test becomes feasible.
- Another challenge was found when doing the robustness testing for the socketing server. As the server will be definitely broken if multiple clients connect to it without any time interval as we used a web application which has its own threshold, and this situation will never happen in the real world. Later the challenge was solved by adding a setInterval() function to connect a client to and disconnect the client from the server every time interval. That also simulates a more realistic situation when the traffic on the server is busy.

Tests and Relevant Challenges

Frontend Testing

Frontend Testing is conducted with automation using Selenium. The main purpose of frontend testing is to show that all navigations and activities within the web application are conducted smoothly without any errors or crashes and also to provide the desired output when given an input in the web application. The test codes can be found in the list of Github Repository Links in the report.

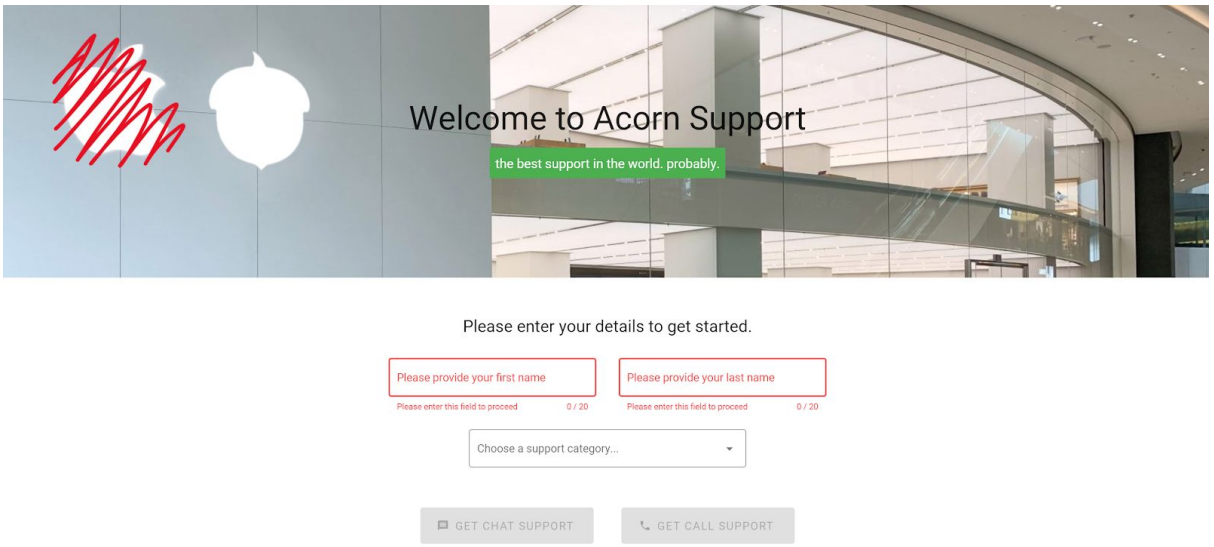
For our project, we conducted a total of 6 different test cases:

Full User Journey

The Full User Journey test case will show how a customer uses our frontend web application extensively (from the start to the end) which will utilise all the functions within the application.
This is the flow of the full User Journey for a an ordinary customer:
Home Page → Key in Details → Get Connected to Agent → Call/Chat → Feedback

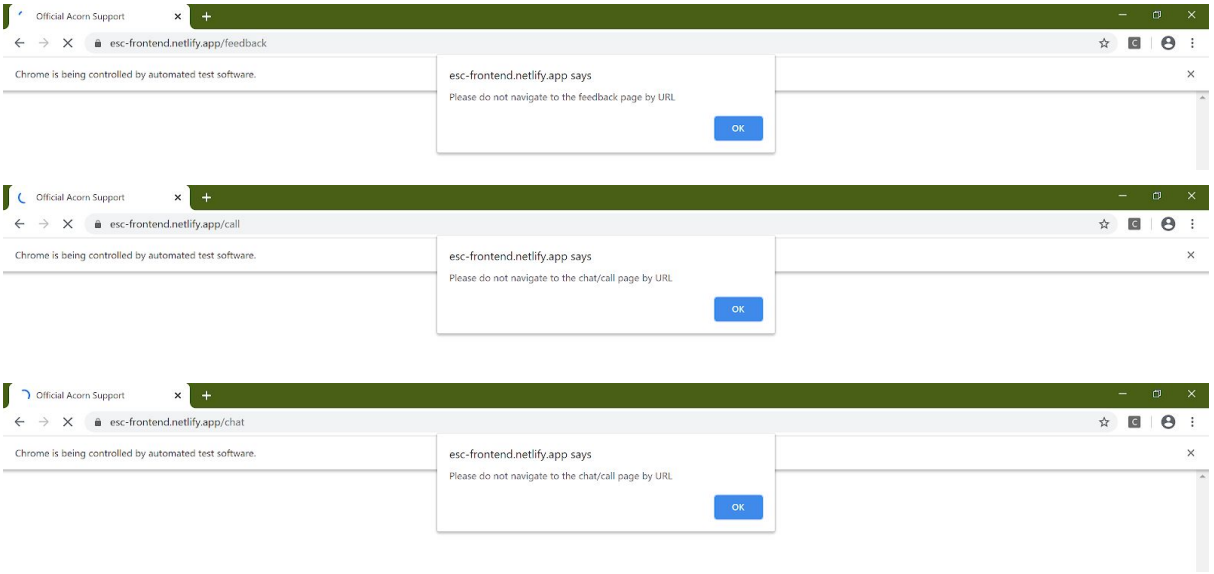
Input Validation for Homepage

This test case will show proper validation of input at our web application's Homepage. The request to seek for chat or call support will be granted only if the customer provides a valid first name, last name and has selected a category in which he/she seeks assistance in. First name and last name must only contain alphabets and cannot be more than a length of 20 characters each. If neither of the conditions are met, there will not be an option available to get chat/call support.



Direct URL Access

This test case will validate if a customer is authorised to access a particular URL in the web application. A customer who accesses a URL directly without proper authorisation will be sent back to the Homepage.

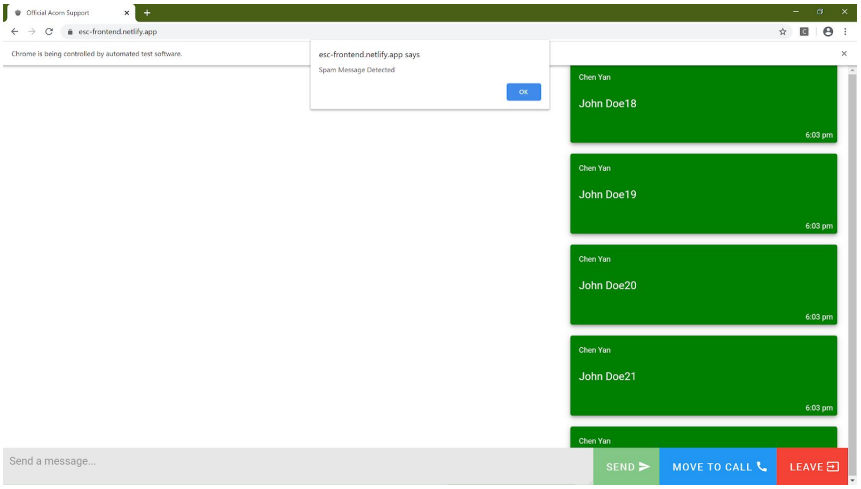


Back Prevention

This test case will show how we prevent customers from going back by using the back button from the internet browser to load the previously loaded page This is to prevent page crashing and other irregularities. This test will display the same prompt message in the browser as the Direct URL test.

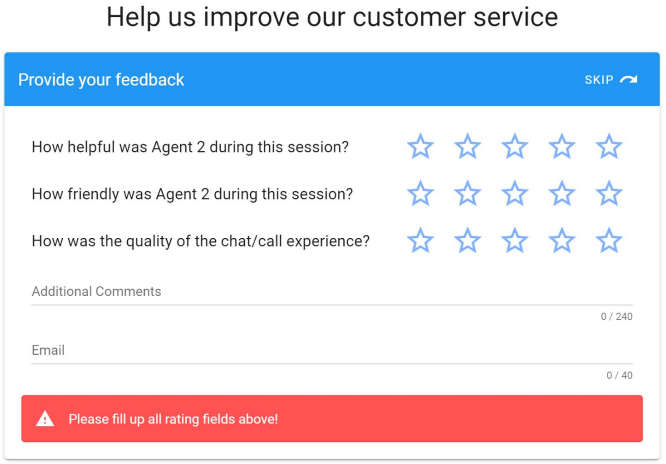
Inappropriate Chat

This test case will show how implementations are set to deal with spam messages which are sent to the agent when connecting via chat support. We consider 20> consecutive messages or any long message above 400 characters without any spacing to be spam messages sent to the agent by the customer. This will prompt an alert message and disconnect the customer from the agent immediately and he/she will be sent back to the Homepage.



Input Validation for Feedback page

This test case will show proper validation of input at our web application's Feedback page. If a customer decides to provide feedback for the agents, he/she must fill up all ratings. If the customer chooses to provide additional comments for the agents on top of the ratings, he/she must provide a valid email address. Not meeting these conditions will cause the application to prompt cue messages to change their inputs.



Backend Testing

Socket Server Testing

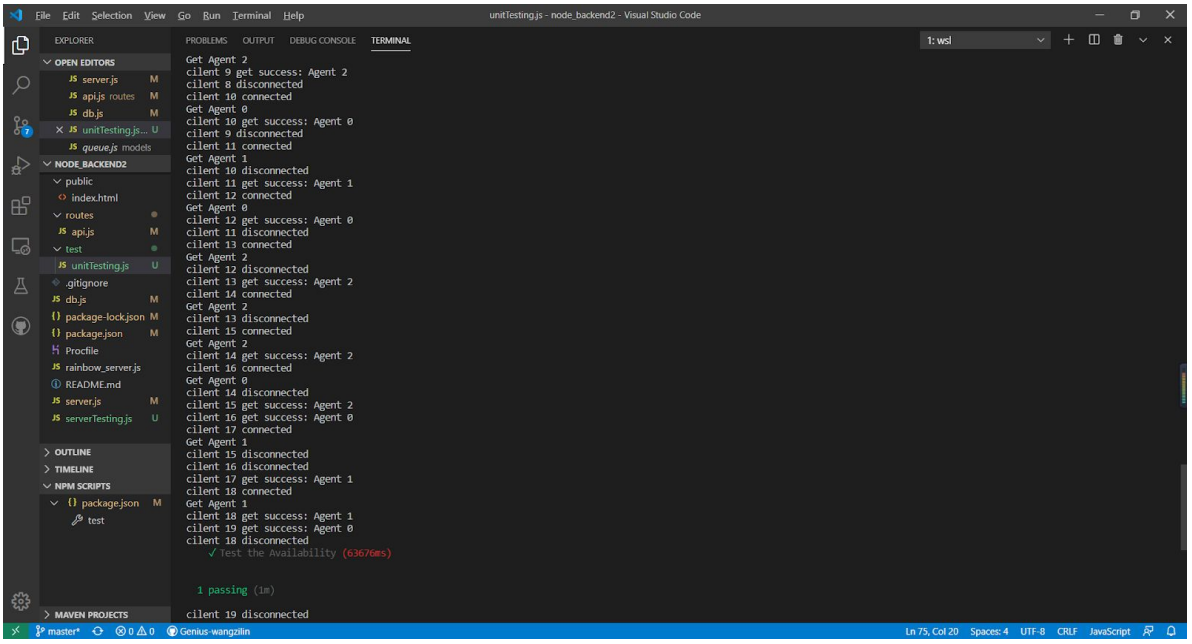
For the socket server, a testing module called Mocha is used to simplify the tests. Mocha provides a tidy testing format and enables diverse testing functionalities. We performed both automated unit and system testing, while doing manual integration testing with the website.

Robustness Testing

Test 1 connects 20 users to the socket server with a short time interval. Each user connected to the server requests for an available agent. When a user gets an agent, it will be disconnected after 2 seconds. Writes to the database are actually performed i.e. the server is forced to actually handle these requests as if they were clients. This test can be augmented to test specific categories, or increase the load if necessary.

```
it('Test the Availability', function(done){
    this.timeout(150000);
    console.log("1) Test the Availability");
    var i=0;
    var clients1=[];
    var count1=0;
    clients1[10] = io.connect(socketURL, options);
    clients1[10].on('connect', ()=> {
        console.log("cilent test connected");
    });
    clients1[10].on("CountingSuccess", ()=>{
        if(count1===10){
            clients1[10].disconnect();
            done();
        }
    });
    async function createClient(n){
        clients1[n] = io.connect(socketURL, options);
        clients1[n].on('connect', ()=> {
            console.log("cilent " + n + " connected");
            var number = Math.floor(Math.random()*3);
            console.log("Get Agent "+ number);
            clients1[n].emit('getAgent', {category:number, firstName:"Emma", lastName:"Watson"});
        });
        clients1[n].on('getAgentSuccess', (data)=>{
            count1++;
            console.log("cilent " + n + " get success: "+data.agentName);
            var ready = 0;
            var beforedisconnect = setInterval(function(){
                if(ready>1){
                    console.log("cilent " + n + " disconnected");
                    clients1[n].disconnect();
                    clearInterval(beforedisconnect);
                }
                ready++;
            }, 1000);
        });
    }
    var interval = setInterval(function(){
        createClient(i);
        i++;
        if(i>=10){
        clearInterval(interval);
        }
    }, 3000);
});
```

Testing code for Test 1



Result of Test 1

Test 2 connects 10 users to the server and gets the same agent with a short time interval. Each connected user requests for the same agent. When a user gets an agent, it will be disconnected after 2 seconds. This test aims to stress test a single channel connected to a single document on our database, and the load can be increased as necessary.

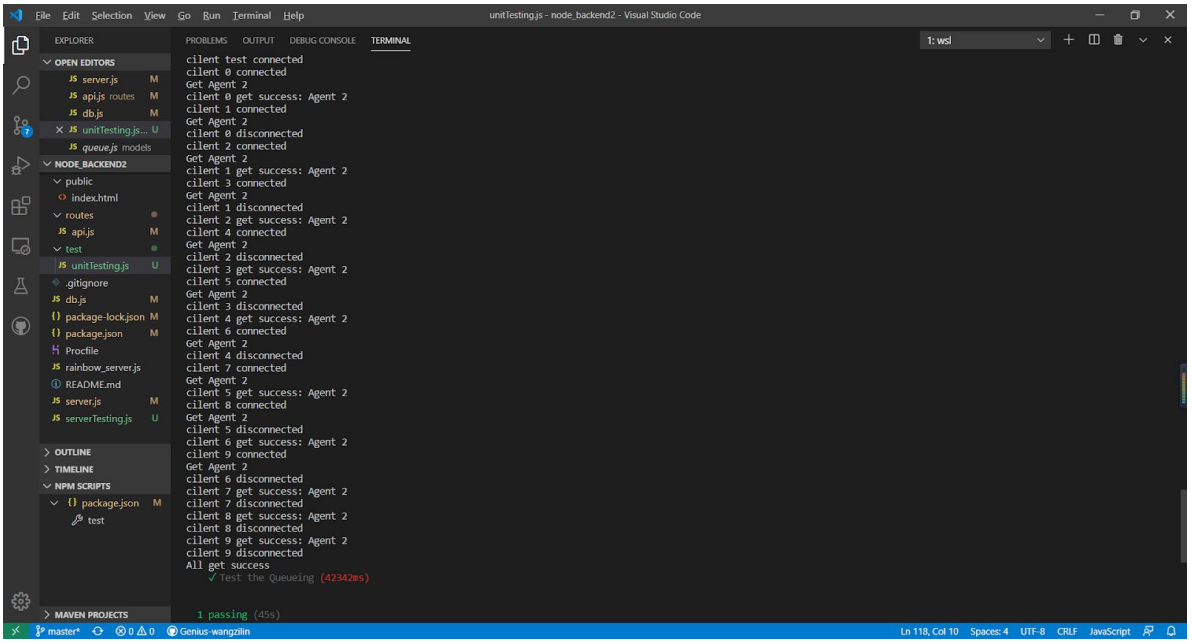
```
it('Test the Queueing', function(done){
    this.timeout(60000);
    console.log("2) Test the Queue");
    var count2 = 0;
    var i=0;
    var clients2=[];
    clients2[10] = io.connect(socketURL, options);
    clients2[10].on('connect', ()=> {
        console.log("cilent test connected");
    });
    clients2[10].on("CountingSuccess", ()=>{
        count2++;
        if(count2===10){
            console.log("All get success");
            clients2[10].disconnect();
            done();
        }
    });
    async function createClient(i){
        clients2[i] = io.connect(socketURL, options);
        clients2[i].on('connect', ()=> {
            console.log("cilent " + i + " connected");
            console.log("Get Agent 2");
            clients2[i].emit('getAgent', {category:2, firstName:"Emma", lastName:"Watson"});
        });
        clients2[i].on('getAgentSuccess', (data)=>{
                console.log("cilent " + i + " get success: "+data.agentName);
                var ready = 0;
                var beforedisconnect = setInterval(function(){
                    if(ready>1){
                        console.log("cilent " + i + " disconnected");
                        clients2[i].disconnect();
                        clearInterval(beforedisconnect);
                    }
                    ready++;
                }, 1000);

        });
    }

    var interval = setInterval(function(){
        createClient(i);
        i++;
        if(i>=10){
        clearInterval(interval);
        }
    }, 3000);

});
```

Testing code for Test 2



Result of Test 2

Authentication Unit Testing

Besides the tests above, we also test the authentication of the connection to avoid unauthorized connections which could flood the server and deny services to customers.

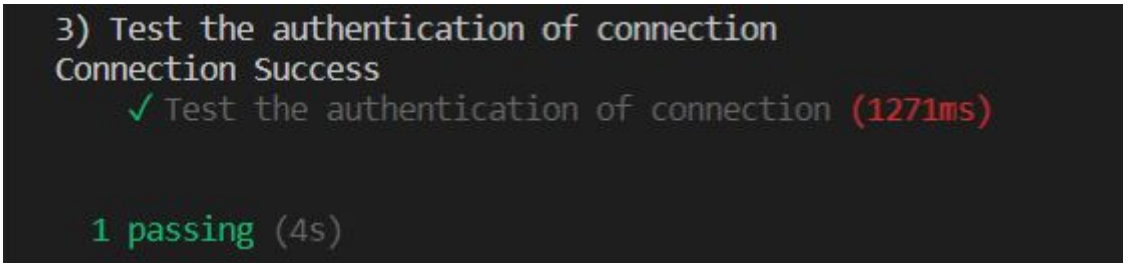
Test 3 connects a user with a valid authorization key. The test passes when the user is connected to the server successfully.

```
it("Test the authentication of connection", function(done){
    this.timeout(100000);
    console.log("3) Test the authentication of connection");
    const socketKey= "BBO5e7IVtK9TeSAQ3RTYGsQOWOZ0QAe8k9jbvomydoOUEjK1lwTLIkK4J3yu";
    var options ={
    transports: ['websocket'],
    'forceNew': true,
    query: { key: socketKey }
    };
    clientAuth = io.connect(socketURL, options);

    clientAuth.on('connect', function(){
        var ready = 0;
        var beforedisconnect = setInterval(function(){
            if(ready>1){
                console.log("Connection Success");
                clearInterval(beforedisconnect);
                clientAuth.disconnect();
                done();
            }
            ready++;
        })

    })
});
```

Testing code for Test 3

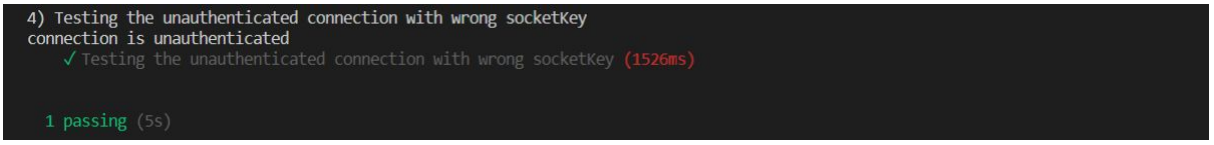


Result of Test 3

Test 4 connects a user with a wrong authorization query during handshaking. The test passes when the user receives an Authentication Error.

```
it("Testing the unauthenticated connection with wrong socketKey", function(done){
    this.timeout(5000);
    console.log("4) Testing the unauthenticated connection with wrong socketKey");
    const socketKey= "wrongkey1";
     (property) transports: string[]
    transports: ['websocket'],
    'forceNew': true,
    query: { key: socketKey }
    };
    var client = io.connect(socketURL, options);
    client.on('error', function(error){
        expect(error).to.eql('Authentication error');
        console.log("connection is unauthenticated");
        done();
    });
});
```

Testing code for Test 4

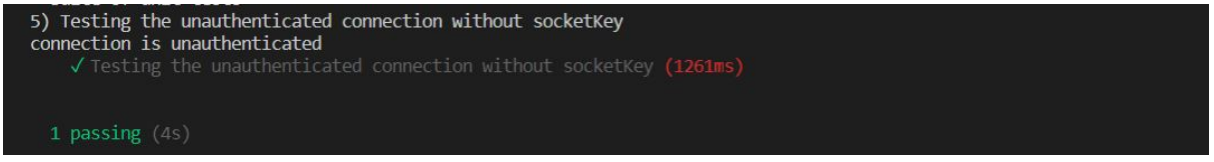


Result for Test 4

Test 5 connects a user without a query field. The test passes when the user receives an Authentication Error.

```
it("Testing the unauthenticated connection without socketKey", function(done){
    this.timeout(5000);
    console.log("5) Testing the unauthenticated connection without socketKey");
    var options ={
    transports: ['websocket'],
    'forceNew': true,
    };
    var client = io.connect(socketURL, options);
    client.on('error', function(error){
        expect(error).to.eql('Authentication error');
        console.log("connection is unauthenticated");
        done();
    });
});
```

Testing code for Test 5



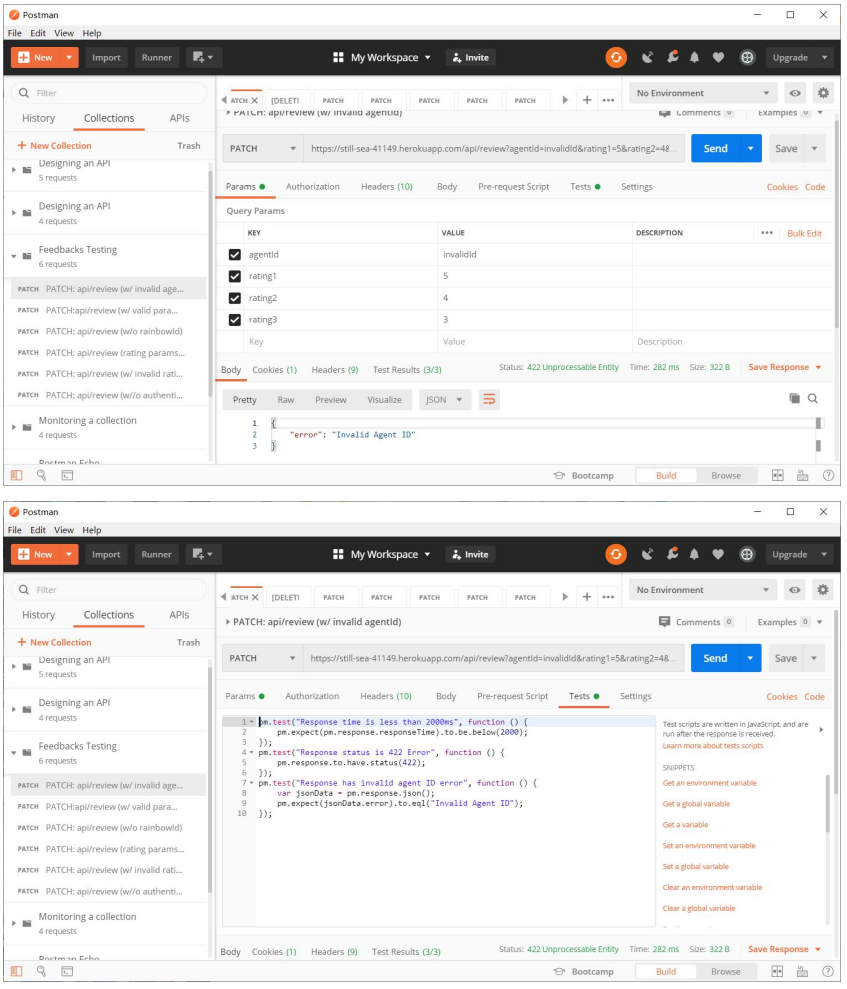
Result of Test 5

Feedback Server RESTful API Testing

We used RESTful API for the backend feedback server as it does not necessarily require real-time updates, and also because of time constraints to integrate it with the admin dashboard. We created several unit tests to show that the endpoint can handle invalid inputs and authentication issues. The tests are mainly on the PATCH function below:

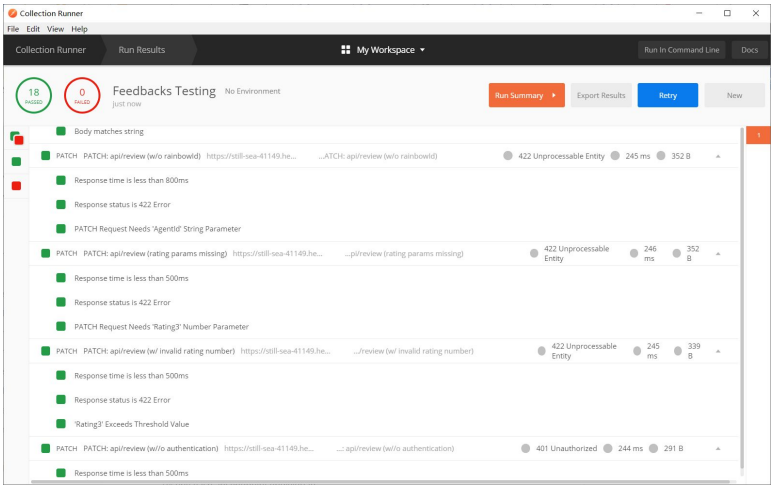
PATCH with path parameter '/review': Update the feedbacks to an agent and update the average ratings to the agent

We used Postman to run the tests. The tests include valid parameter inputs, invalid parameter inputs, missing the necessary parameters, and authentication check.



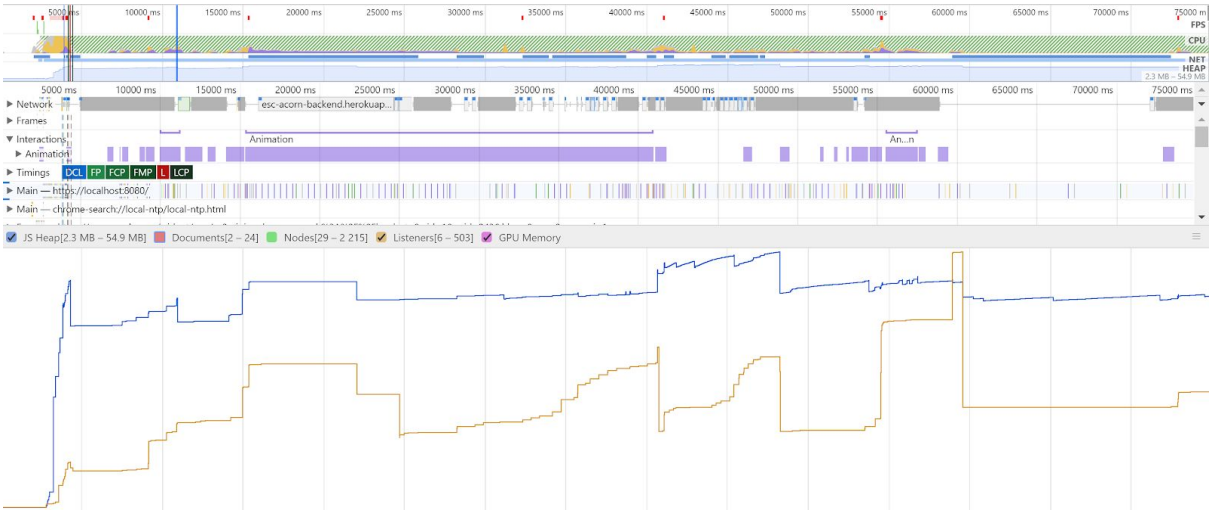
Sample Test Example: PATCH with invalid agent ID

Postman also provides Collection Runner and makes running the tests more convenient.



All the tests run by Collection Runner for the feedback system

Integration Testing



Overall, our website seems to have no major performance issues. There are no memory leaks throughout the user journey even while switching between chat and call support (meaning no unforeseen crashes are likely), and the largest amount of memory seen for our JS heap is well within the expected range for a web application, allowing users to access it through both desktop and mobile easily.

Lessons Learnt

Technical Skills

Technical skills learnt: CSS/HTML/JS, Selenium, Mocha, Postman, MongoDB, JSON file format, HTTP protocols and API, Web Sockets

Firstly, this project allowed us to experience and implement each phase of the Software Development Lifecycle. We conduct this project by applying the Agile iteration model . At the same time, We generated the use case diagrams and the UML diagrams under the 'Design' stage to help us describe, visualize, construct, and document the artifacts of a software system.UML diagram also is an easy way to consider system behavior,detect errors and omissions early in the life cycle , present the proposed designs and communicate with stakeholders ,help us to understand the requirements and drive implementation.

In terms of technical skills,in this project, there are many new things that we haven't learned before. Hence, we need to pick up these new skills through different tutorials of these applications. For example, I need to learn HTML/JS/CSS language , understand the framework of vue and think of a way to link each component together.In order to conduct frontend work easily, I also try to use Vue UI components to build , manage and develop our web application directly. In addition, Ziling and Aaron are responsible for backend stuff. They picked up MongoDB Atlas as a relational database. MongoDB Atlas is a fully-managed cloud database. It handles all the complexity of deploying, managing, and healing your deployments on the cloud service provider of your choice (AWS, Azure, and GCP).Postman is another tool that we used for building an API . It is easier for us to create, share, test and document APIs.

Testing framework and principles is another aspect of what we learnt in this project.For Frontend testing, we conduct a testing about Graphical User Interface (GUI), functionality and usability of websites. The automatic testing tool selenium helps us to conduct basic frontend testing. We also knew something about JavaScript Unit Testing,Integration Testing,Performance/Stress Testing,Automated Browser Testing,Visual Regression Testing,Accessibility Testing and User Testing. Based on our requirements, we implement basic testing for web performance ,integration testing ,robustness testing and error handling(such as connection issues(eg. no internet connection, connection to the server errors) , invalid input issues(eg. Haven't filled all empty input boxes or exceed maximum words) and verification issues(eg.recaptcha verify)). For backend testing, we conduct socket server testing by Mocha, system testing, authentication unit testing . We also used Postman to do the feedback server restful API testing .

Apart from the technical skills, we also realised the importance of having a modular code especially when working on a big project with multiple team members. It proved to be very useful when keeping track of everything. Proper documentation of code plays a

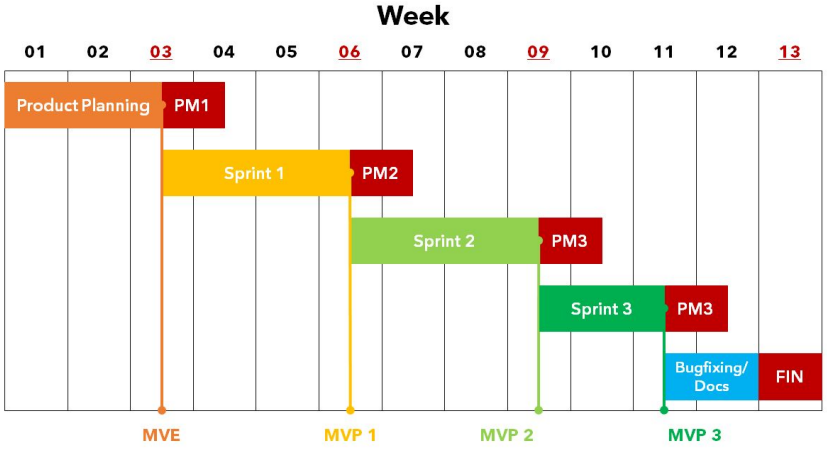
crucial part in understanding code which is written by the various members in the group which will result in a smoother integration of the project and also easier troubleshooting if anything goes wrong.

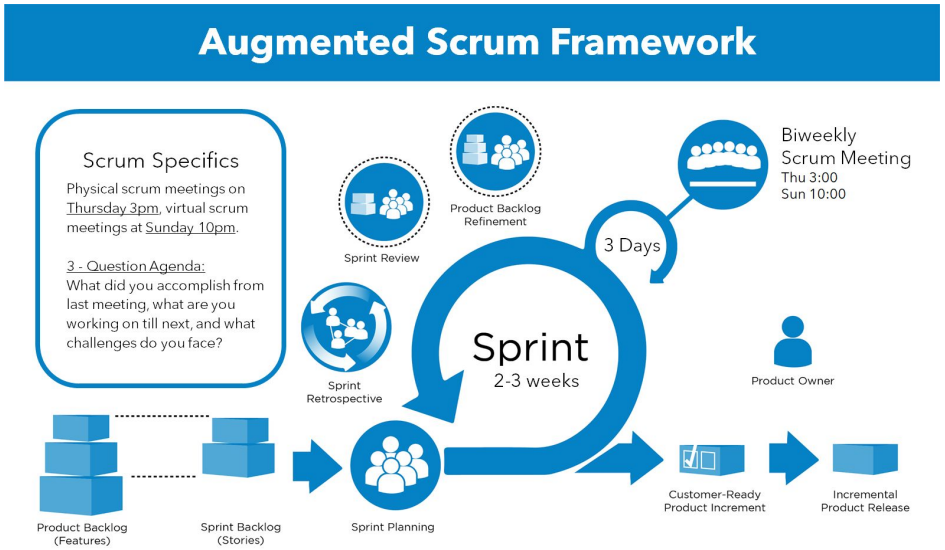
In general, It is a positive and happy experience for everyone in our team in this project, everyone contributed to the project and all of us learned a lot.

Project Development

In terms of project development, our team utilised agile software development methodology, planning a total of 3 sprints. This realistically started from week 2, and our team stuck quite faithfully to the schedule.

This predetermined schedule along with our project backlog enabled our team not to lose inertia despite the COVID outbreak, and ensuring progress was made from each project meeting to the next.In addition, putting in adequate time beforehand allowed us to catch most bugs and present a fairly polished MVP at each checkpoint.





Below is our product backlog, which our team used to set goals and check them off as we completed each task.
ESC Product Full Backlog:
https://docs.google.com/spreadsheets/d/1Nj8H4YL0QQe8Vw2GI-_nJxt-ogssKFX_9_QOfArjeAU/edit?usp=sharing

PROJECT NAME	SCRUM MASTER	START DATE	END DATE	OVERALL PROGRESS	PROJECT DELIVERABLE	Full Stack Web Application
Alcatel ESC Project	Aaron	1/27/2020	4/27/2020	100.00%	SCOPE STATEMENT	Building a Customer Service Platform

AT RISK	TASK NAME	IMPORTANCE	RESPONSIBLE	STORY POINTS (1 - 5)	START	FINISH	DURATION (DAYS)	BURNDOWN STATUS	FRAMEWORK/LIBRARIES
	Product Planning				1/27/2020	2/12/2020	16	Complete	
	High Level Planning	Must Have	All	1	2/6/2020	2/13/2020	7	Complete	N.A.
	Wireframing	Must Have	All	1	2/13/2020	2/14/2020	1	Complete	N.A.
	Sprint 1				2/13/2020	3/4/2020	20	Complete	
	Front End Boilerplate Setup	Must Have	Jeremy	1	3/2/2020	3/3/2020	1	Complete	VueJS
	Basic Home Page	Must Have	Chen Yan	1	3/2/2020	3/7/2020	5	Complete	VueJS
	Chat Page Elements	Must Have	Chen Yan	2	3/4/2020	3/7/2020	3	Complete	VueJS
	Rainbow Chat API Integration	Must Have	Jeremy	2	3/3/2020	3/10/2020	7	Complete	VueJS, Axios, Rainbow Web SDK
	Backend REST API	Must Have	Aaron, Zilin	2	2/13/2020	2/20/2020	7	Complete	Mongo/Express/Node Framework
	Agent Assignment Mechanism	Must Have	Aaron, Zilin	2	2/20/2020	2/25/2020	5	Complete	MongoDB
	Sprint 2				3/5/2020	3/25/2020	20	Complete	
	Individual Product Pages	Could Have	Chen Yan	1	3/6/2020	3/11/2020	5	Complete	VueJS
	Call Page	Should Have	Chen Yan	2	3/14/2020	3/21/2020	7	Complete	VueJS, Rainbow Web SDK
	FAQ Page	Should Have	Chen Yan	1	3/7/2020	3/14/2020	7	Complete	VueJS
	Establishing HTTPS connection for Webpage	Must Have	Jeremy	2	3/17/2020	3/19/2020	2	Complete	SSL, VueJS
	Rainbow Call API Integration	Should Have	Jeremy	3	3/18/2020	3/20/2020	2	Complete	VueJS, Rainbow Web SDK
	User Interface Automated Testing	Must Have	Chen Yan, Jeremy	2	3/23/2020	3/25/2020	3	Complete	Selenium, Chromedriver, Java
	Guest Account Creation	Must Have	Zilin	2	3/6/2020	3/8/2020	2	Complete	Rainbow Node SDK
	Queue Pushing/Pulling	Must Have	Aaron, Zilin	2	3/8/2020	3/15/2020	7	Complete	Express, MongoDB Queries
	Queue Polling Mechanism	Must Have	Aaron	2	3/15/2020	3/19/2020	4	Complete	VueJS, Axios
	Queue Leaving/Patching	Must Have	Aaron, Zilin	3	3/19/2020	3/23/2020	4	Complete	Express, MongoDB Queries
	Node Backend Automated Testing	Must Have	Aaron, Zilin	2	3/23/2020	3/26/2020	3	Complete	Postman
	Sprint 3				3/26/2020	4/8/2020	13	Complete	
	Review Page Integration	Could Have	Jeremy	3	3/28/2020	3/28/2020	1	Complete	VueJS, Axios
	Front End Testing (User Journey)	Must Have	Chen Yan	2	4/11/2020	4/18/2020	7	Complete	Selenium, Chromedriver, Java
	Front End Testing (Functionality Testing)	Must Have	Jeremy	2	4/11/2020	4/18/2020	7	Complete	Selenium, Chromedriver, Java
	Admin Panel	Should Have	Aaron	4	4/8/2020	4/11/2020	3	Complete	CRUD Operations for Agents
	Web Socketing for Queue Management	Could Have	Aaron	3	3/28/2020	4/3/2020	6	Complete	Express, MongoDB Queries, WebSockets
	Multi-Queue Model (Optimisation)	Should Have	Zilin	3	3/28/2020	4/2/2020	5	Complete	MongoDB
	Bugfixing and Documentation				4/9/2020	4/19/2020	10	Complete	
	Hotfixes	Must Have	Everyone	3	4/13/2020	4/27/2020	14	Complete	
	Hosting	Must Have	Aaron	2	4/12/2020	4/14/2020	2	Complete	Netlify
	Authentication	Should Have	Aaron, Zilin	2	4/16/2020	4/18/2020	2	Complete	
	Interface Robustness	Must Have	Jeremy	2	4/16/2020	4/18/2020	2	Complete	VueJs
	Interface Polishing	Could Have	Chen Yan, Jeremy	3	4/16/2020	4/18/2020	2	Complete	VueJs

Overall, we made an effort to embody the principles of the agile manifesto, and here are some examples from our team's journey which we believe display the principles listed:

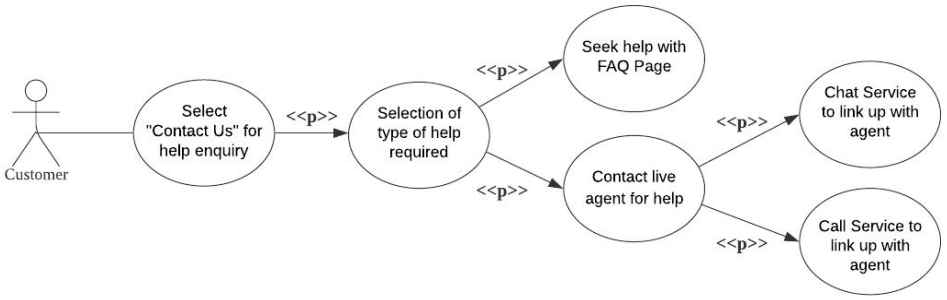
1. Prioritising to build a robust but feature-rich ecosystem (*Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*)
2. Upgrading our queueing system from RESTful API protocol to Web Sockets (*Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.*)
3. Kept to a planned sprint cycle consisting of 2-3 weeks each with a working MVP at each point *(Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.*)
4. Maximised time spent on robustness and system testing, while using manual methods for simple unit testing *(Working software is the primary measure of progress.)*
5. We maintained a relatively steady rate of output deliverables against our product backlog for the entirety of the project *(The sponsors, developers, and users should be able to maintain a constant pace indefinitely.)*
6. Choosing a flexible stack and considering available tools like MongoDB Atlas instead of adopting no-invented-here (NIH) syndrome *(Simplicity--the art of maximizing the amount of work not done--is essential.)*

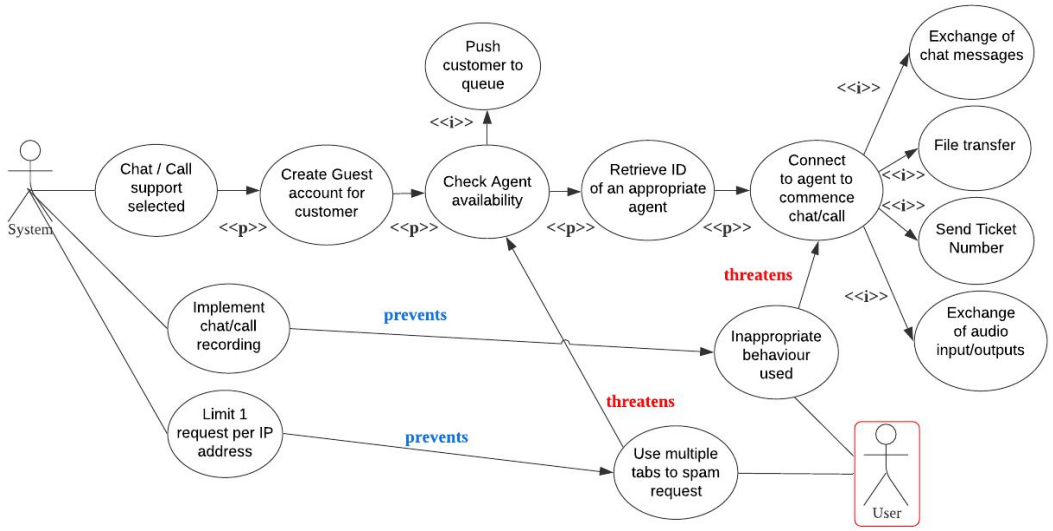
Appendix

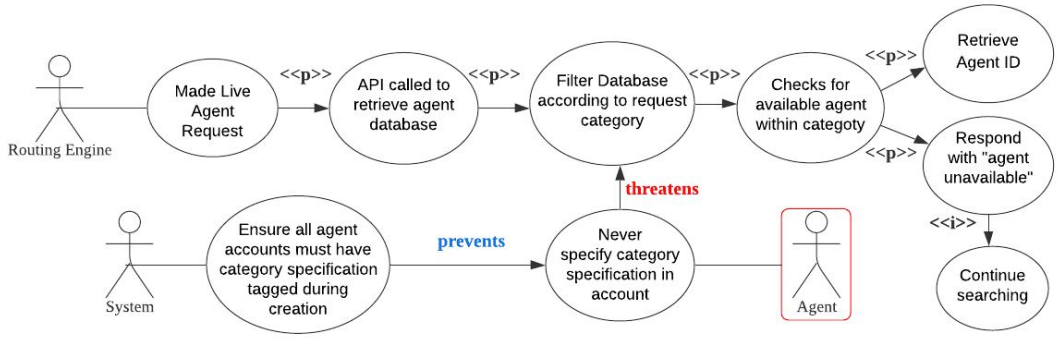
Initial Design of the Project

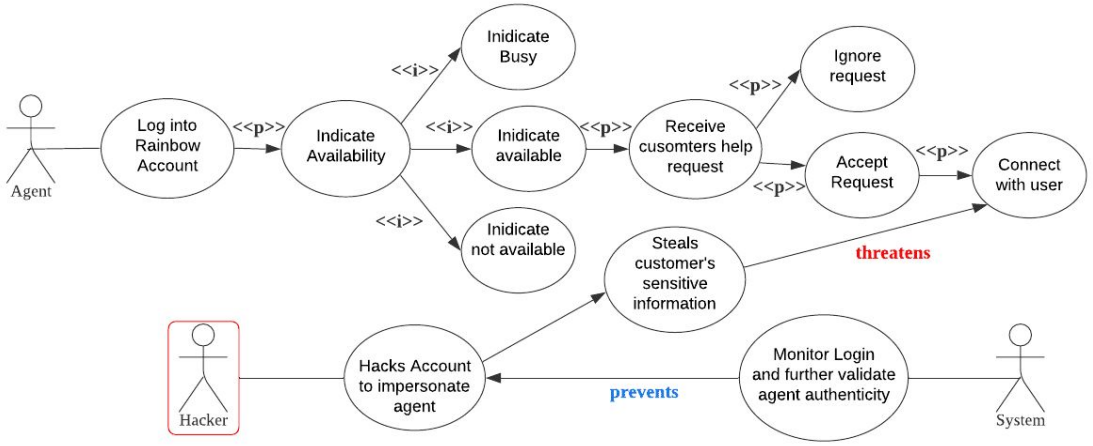
The following designs were made during the initial stages of the design phase of the Project which shows the amount of progression our team has made over the weeks. These diagrams have been updated in the main report.

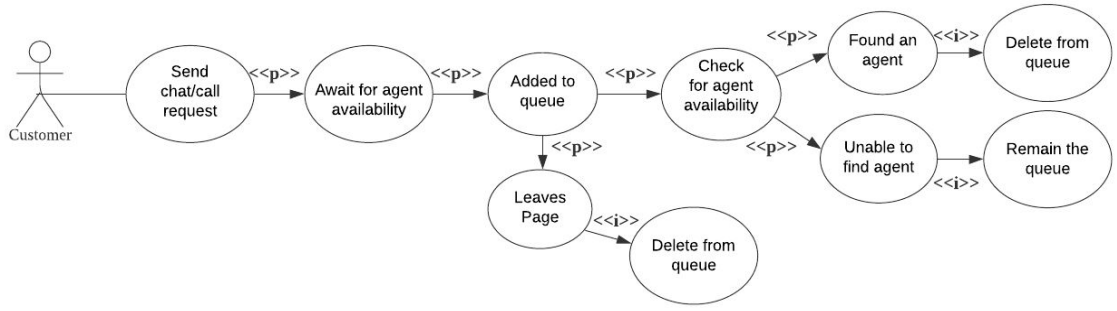
Use Case Diagrams





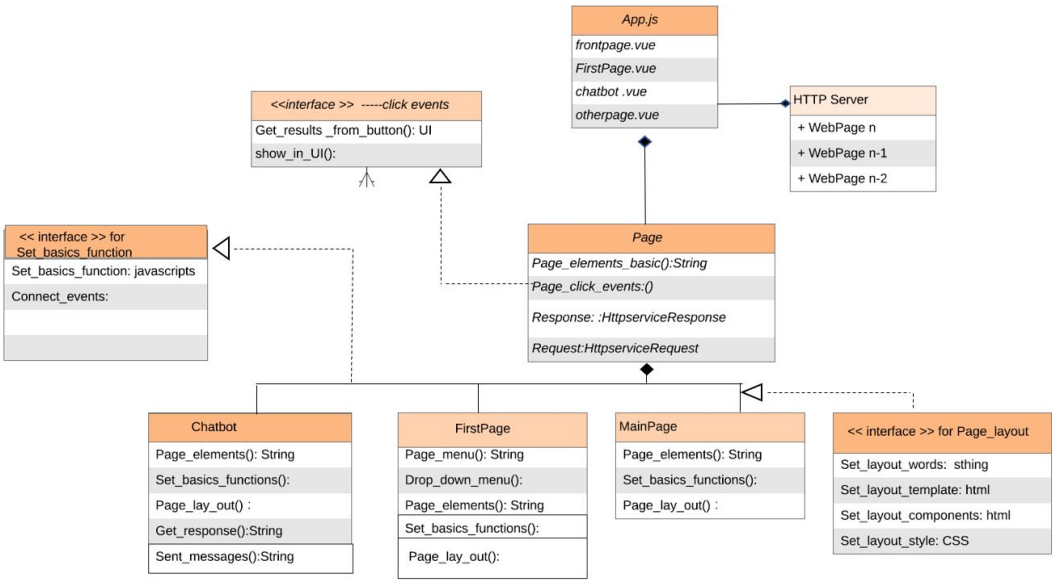



Agent
Log into Rainbow Account
<<p>>
Indicate Availability
<<i>>
Inidicate Busy
<<i>>
Inidicate available
<<i>>
Inidicate not available
<<p>>
Receive cusomters help request
<<p>>
Ignore request
<<p>>
Accept Request
<<p>>
Connect with user
Steals customer's sensitive information
threatens
Hacker
Hacks Account to impersonate agent
prevents
Monitor Login and further validate agent authenticity
System

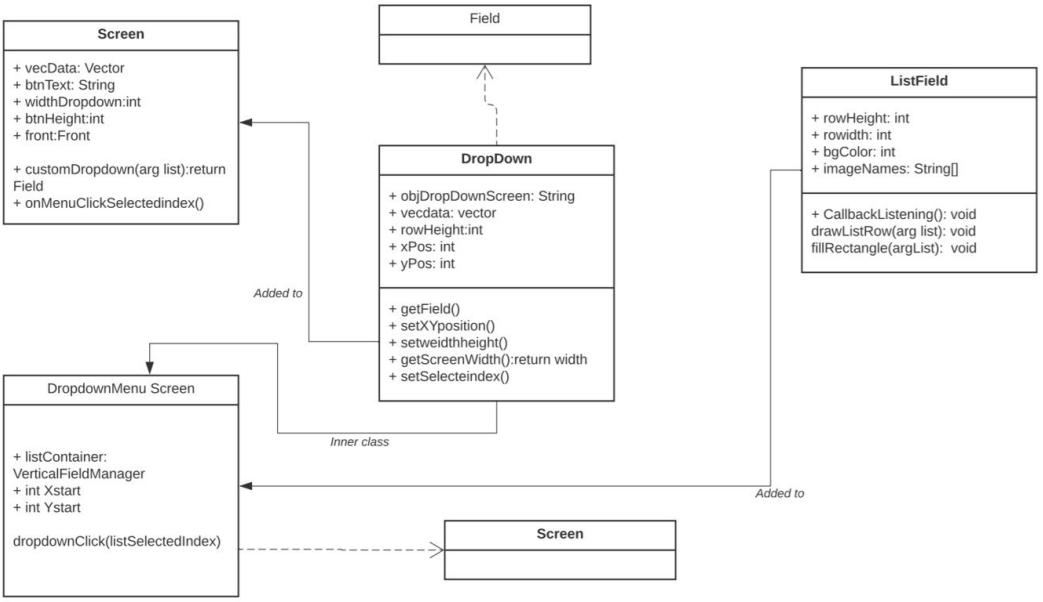

Customer
Send chat/call request
<<p>>
Await for agent availability
<<p>>
Added to queue
<<p>>
Check for agent availability
<<p>>
Found an agent
<<i>>
Delete from queue
<<p>>
Unable to find agent
<<i>>
Remain the queue
<<p>>
Leaves Page
<<i>>
Delete from queue

Frontend Class Diagrams

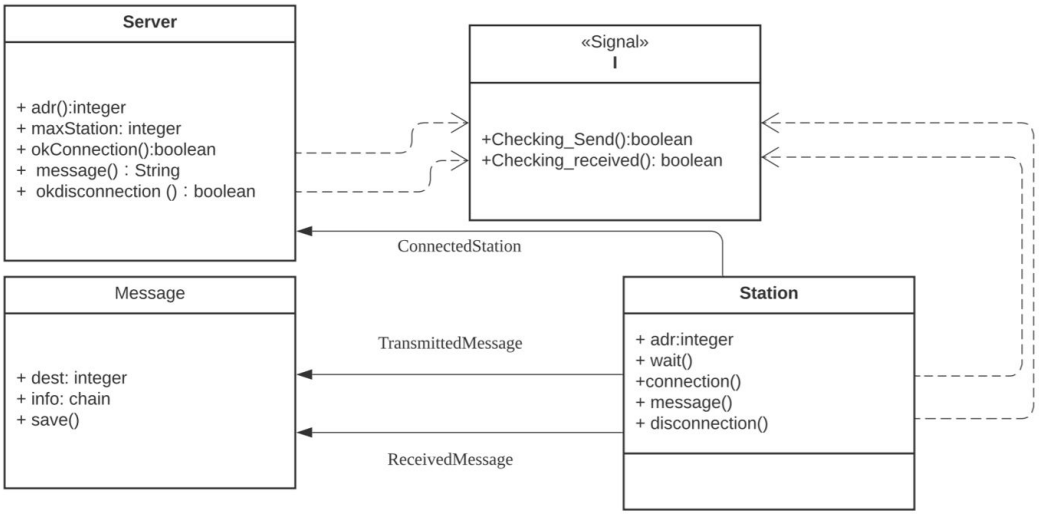
Main class diagram for web application structure



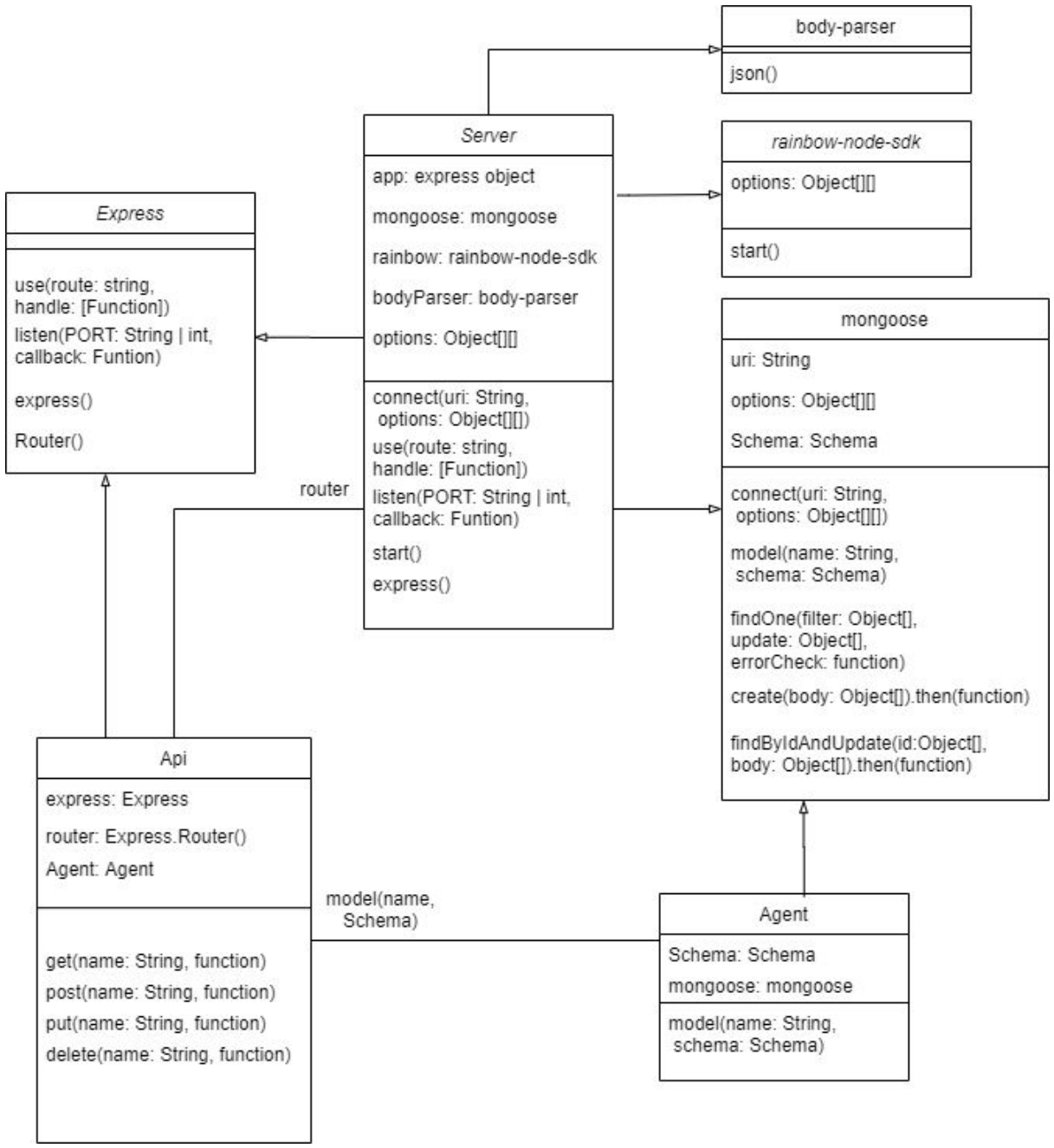
Class Diagram for Dropdown Menu feature



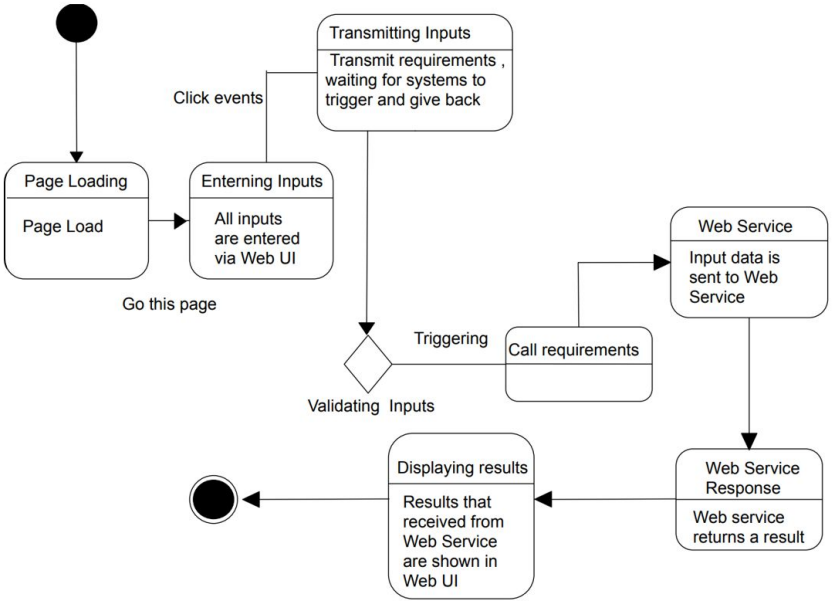
Class Diagram for Chat service with agent



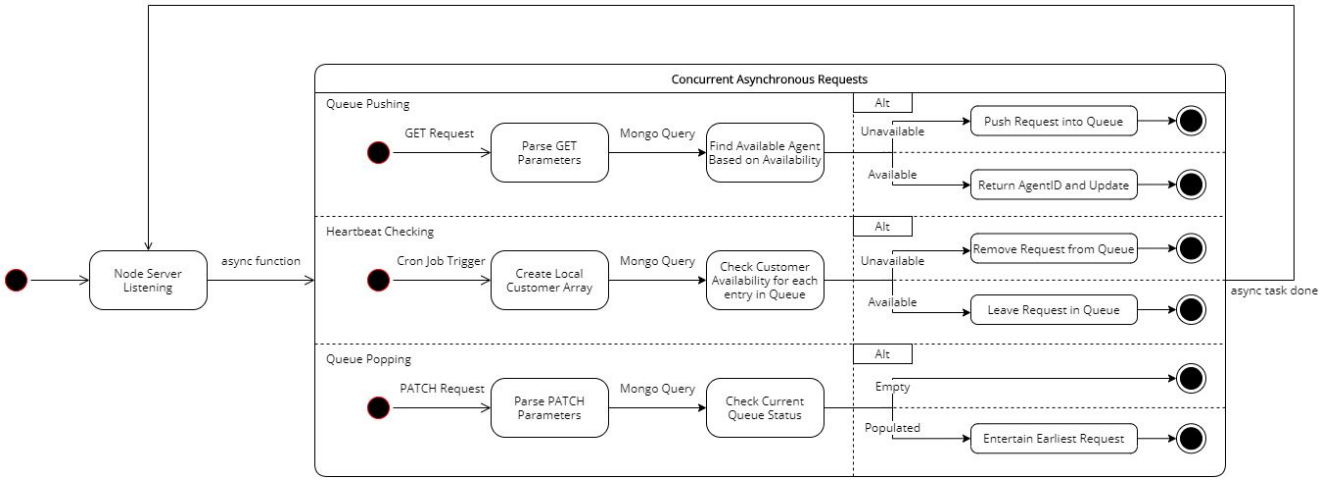
Backend Class Diagram



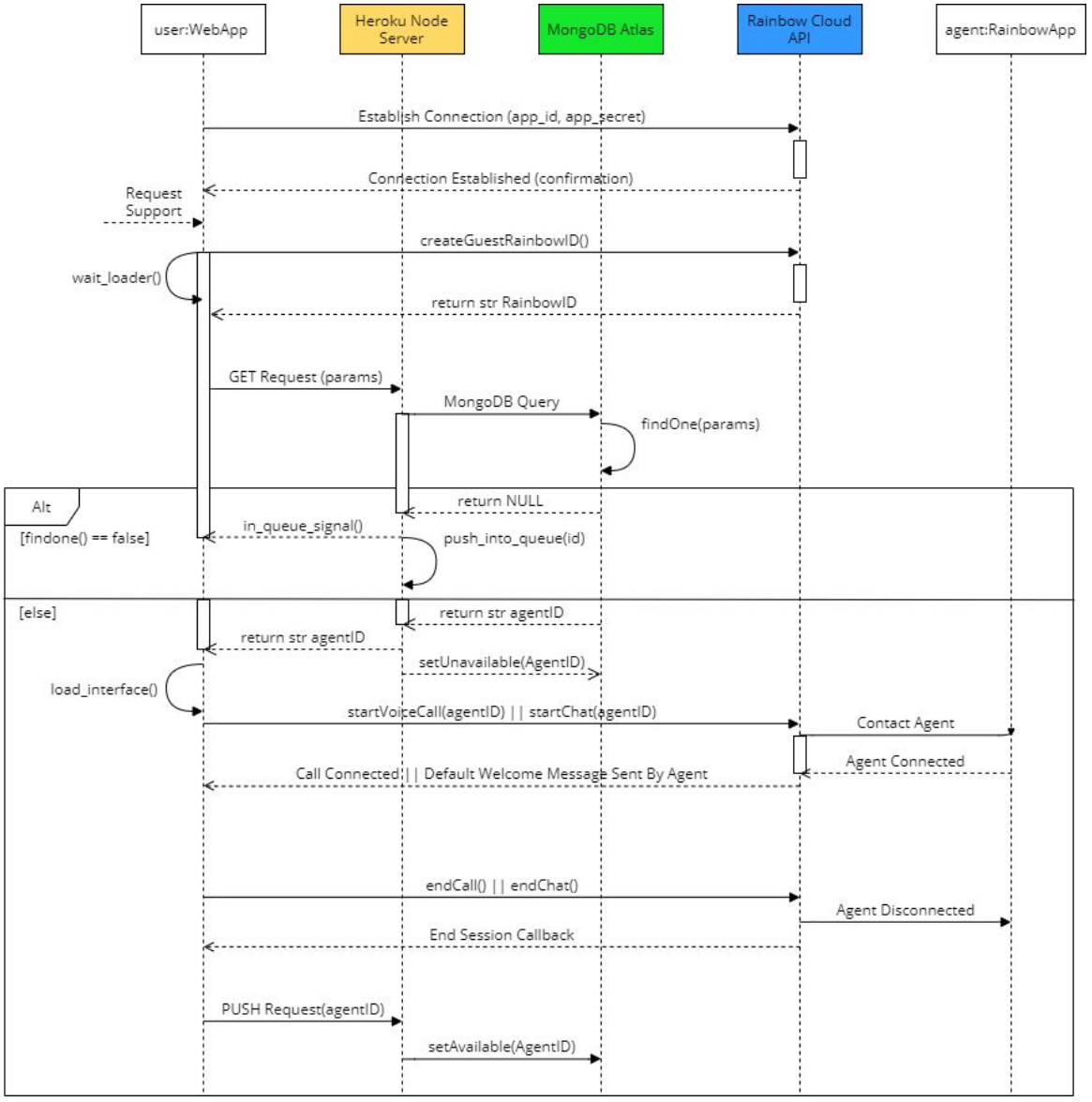
Frontend State Machine Diagram



Backend State Machine Diagram

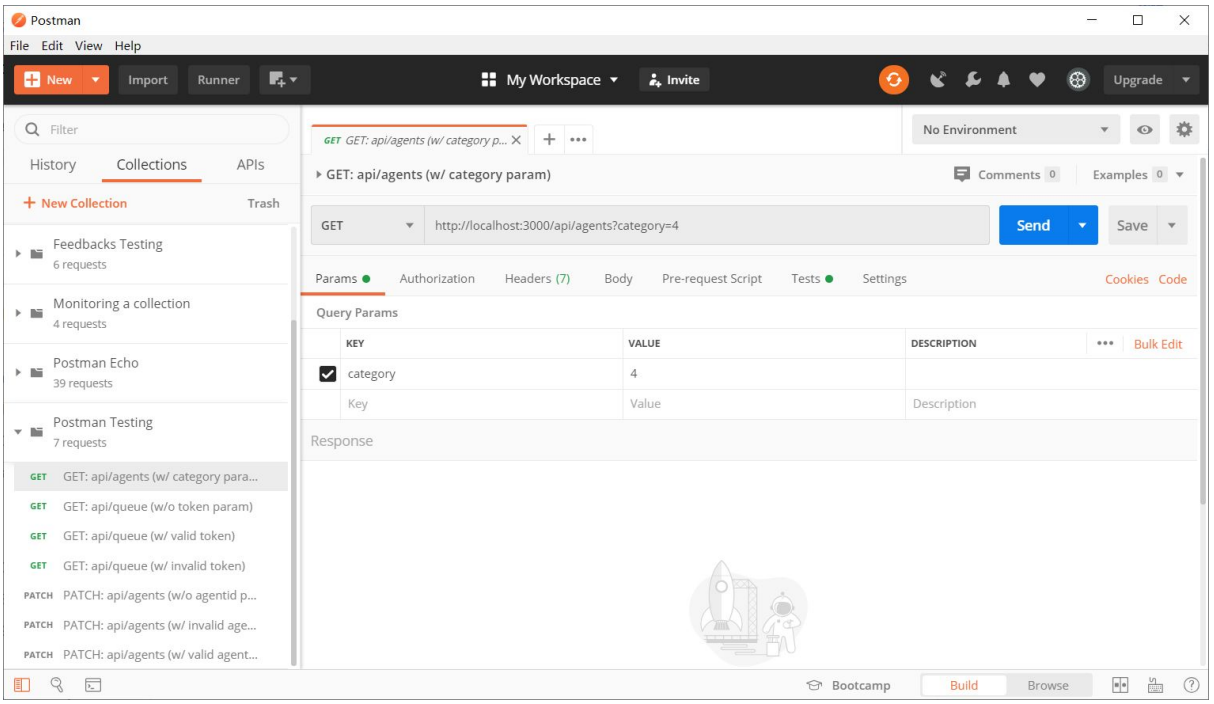


Sequence Diagram

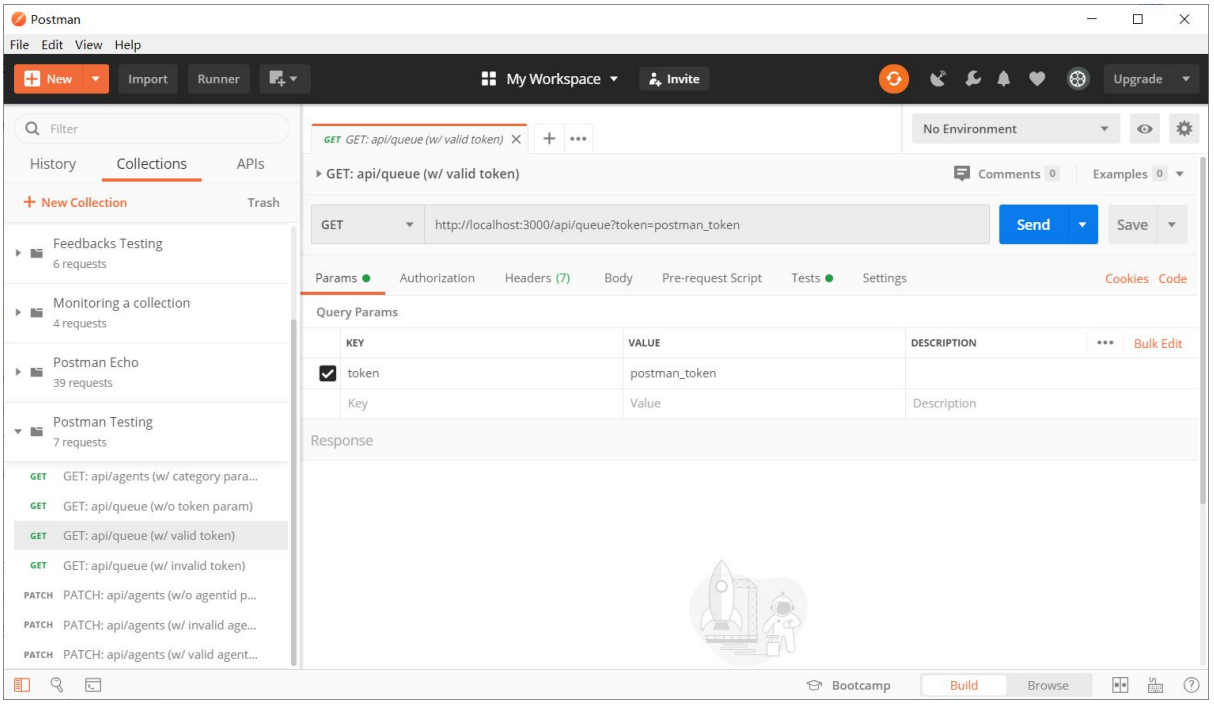


Backend Unit Testing (old test suite)

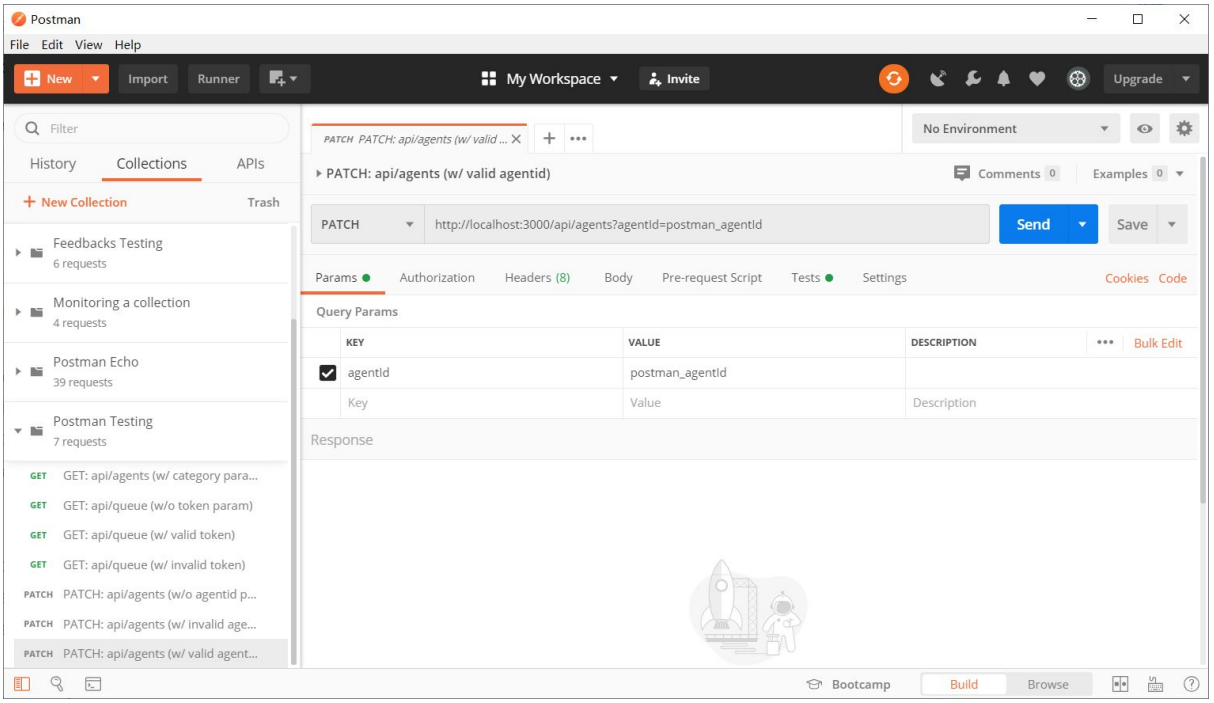
1. GET /agents



2. GET queue



3. PATCH queue



Collection Runner