

# Test Document

## *Amazon Connect Supervisor Insights*

<b>Version</b>	1.5.0
<b>Revision Status</b>	Pending
<b>Last Update</b>	26.04.2024

---

**Safe Corp Jiva**

## Revision History

Name	Date	Reason For Changes	Version
First Draft	20/February/2024	Initial test document.	0.1
First Presentation Feedback	27/February/2024	Added missing sections (5,6,7,10).	0.2
Second Feedback Presentation	5/March/2024	Added many missing references, finished sections 7 and 10.	0.3
Final version before first official version	14/March/2024	Corrected section 11. Described deliverables, added description of section 9.	1.0
Updated comments	24/April/2024		1.5



# Table of Contents

<b>Revision History</b>	<b>1</b>
<b>1. Objective</b>	<b>4</b>
<b>2. Scope</b>	<b>4</b>
2.1. Authentication	4
2.2. Backend	4
2.3. Frontend	4
<b>3. Test Methodology</b>	<b>5</b>
<b>4. Assumptions</b>	<b>5</b>
<b>5. Risks</b>	<b>6</b>
<b>6. Mitigation Plan</b>	<b>6</b>
<b>7. Roles and Responsibilities</b>	<b>8</b>
<b>8. Schedule</b>	<b>9</b>
<b>9. Defect Tracking</b>	<b>10</b>
<b>10. Test Environment</b>	<b>10</b>
10.1. Front-end	10
10.2. Backend	11
10.3. Local Testing Environment	11
<b>11. Entry/Exit Criteria</b>	<b>11</b>
<b>12. Deliverables</b>	<b>14</b>
<b>References</b>	<b>15</b>

# Test Document

## 1. Objective

Supervisor Insights is a project that tries to simplify the day to day work of call center supervisors by centralizing metrics, actions, and real time monitoring in a single application. Due to the importance this software could have for businesses and individuals, we have designed tests that ensure security, reliability, and the overall availability of the resources.

We shall adopt *Unit Testing* and *Integration Testing* strategies to ensure the correctness of each module as well as when these are used in combination. Moreover, we considered *Load Testing* and *Performance Testing* to determine the capabilities of the system, identify bottlenecks, and strive towards real time flows of data.

## 2. Scope

### 2.1. Authentication

One crucial element of modern web applications is authentication. Managing user credentials both in transit and at rest is of utmost importance, therefore, we decided to use Amazon Cognito with Amplify (and the UI library) which follow industry standards. Amplify implements both Cognito and IAM as microservices within their authentication solution, and therefore our authentication solution will expand to those microservices. Consequently, we shall limit our authentication tests to the frontend to guarantee the correct use of the interface provided by Amplify.

### 2.2. Backend

We will also have to implement a GraphQL API, following all of our requirements for the backend that may be broken down within our Software Requirement Specification. This will include any CRUD operations necessary, along with any authentication middleware we might need. Furthermore, the application uses a stream at one point of the connection with Amazon Connect, which will have to be another endpoint to keep in mind. Some of the specific backend functionalities include the following but are not limited to:

- Real time call updates: assert that call transcripts are performed on an ongoing call, that sentiment analysis is performed, that a request for help by an agent is sent to a supervisor, among many of the rest of the requirements specified in our SRS.

- Correct implementation of copilot-agent chat: assert that messages are sent and received in real time, meaning that they are created in the DB, and that a live web socket connection is correctly established between two peers.

### 2.3. Frontend


The frontend of our application consists of both connections to the backend, and connections to authentication. Thus, the application will involve quite a robust frontend development experience, which is eased through the usage of *NextJS* as our frontend framework. The architecture of *Next*, which divides components into server and client components, will allow us to provide faster and more efficient reactivity wherever it may be needed within our application, and render what is not reactive, on the server directly.

On the frontend, our tests will mainly consist of testing rendering of the web application, and ensuring that components work as expected when new changes are introduced. Naturally, we would make “selective tests”, that is, write tests for components or functionalities that we deem somewhat or very likely to fail upon changes, and the reason for this could mainly be how much a component interacts with others and with external services. That said, the following would be tests that could be performed on the frontend, but they may vary and be adjusted to fit the system’s evolving shape:

- Correct fetching of the user’s information, by asserting that the user’s information is displayed on screen.
- Appropriate functionality for message sending buttons, ask for help button, join call button, among others.
- Assert that certain interactions are explicit to the user, for example, when help is requested, ensure that the user is notified in the defined manner.

## 3. Test Methodology

As we have opted into using some of the serverless solutions that AWS provides such as Amplify, some resources will be coupled together in production. However, since each service should be only responsible for its own performance and reliability, we shall mock external systems for unit tests and create development environments for integration tests. Thanks to the command *amplify pull* we will be able to replicate all our microservices within every machine, thus creating the ease of use across the team.



Furthermore, our frontend is running on the *NextJS* framework, which hosts a server in *Node* and facilitates once more the replication of the environment within each machine owned by the team. Along these same lines, Amplify exposes a *Command Line Interface (CLI)*, which helps us in running mock versions of our *Lambda* functions, using the *amplify mock* command. It is thus important to note that the *amplify mock* command also works for more technologies than the *Lambda* functions only; we are able to mock [virtually all of our microservices that are hosted within Amplify](#).

## 4. Assumptions

We will continue forward with the following assumptions, as our testing document evolves:

1. We will have a team consisting of 15 people, all able to work in a capable manner, who are also willing to learn the new technologies and navigate documentation as they see fit.
2. Our team will be working on machines either from Microsoft, or Apple, along with operating systems including *Windows*, *MacOS* and *Linux*.
3. The team will have 10 weeks to complete the development process for the project, including testing.
4. Each “man day” from the team implies a work time of 5 hours.

## 5. Risks

### 1. *SafeTest*

As it will be highlighted in [\*\*Section 10\*\*](#), we have opted to use *Netflix*’s cutting edge library *SafeTest* for the frontend. Since we are adopting a new technology we might face challenges due to the lack of information sources or even potential internal errors found within the library as this is more common in projects with less maturity.

### 2. Resources

Tight deadlines, budget constraints, or the size of our team may lead to rushed or incomplete testing suits, increasing the risk of undetected bugs and quality issues.

### 3. Environments

Inconsistent settings across testing and production stages, such as different hardware specifications, software versions, or data sets, leading to inaccurate or unreliable test outcomes.

#### 4. Scope Changes

Changes in requirements or functionality mid-development can disrupt testing plans and require adjustments.

#### 5. Technology Adoption

Using new technologies can introduce unforeseen challenges and require additional expertise for testing.

## 6. Mitigation Plan

To reduce the impact from the previously mentioned risks we have considered the following mitigation plans for each:

#### 1. *SafeTest*

The library used for testing is a very bleeding edge, and as with most bleeding edge technologies, we have to be wary of potential bugs and breaking changes as it is something that is being worked on consistently. That being said, the main reason we have chosen to use *SafeTest* as our testing platform is because of its tests' similarity to the *Jest* testing environment, thus allowing us to switch libraries on the fly if necessary.


#### 2. Resources

We will focus on mitigating mainly the implementation of buggy, and lower quality features through our development operations systems (e.g. Github Actions, no Amplify Pushing from developers other than the designated leads, etc). Furthermore, we can ensure that the size of our team will not be an issue, by creating sub-teams who are able to more easily communicate with one another, through the usage of the team leads, which in turn also facilitates the communication within the teams themselves.

#### 3. Environments

We will have to ensure that versioning of our libraries is properly understood by everyone on the team. When it comes to the backend, the version control will generally not be an issue, given that the hosting service we are using does not explicitly expose different versions, and thus through the use of the *amplify pull* command, everyone will be in sync with each other.

#### 4. Scope Changes



The easiest way to mitigate the risk for scope changes within a software project would be to thoroughly develop the requirements. We also possess a ranking system for these requirements, which allows the team to prioritize those features that might be imperative to the functioning of the system.

#### 5. Technology Adoption

These new technologies nowadays often come bundled with good documentation. As is the case with [SafeTest](#). The same library also allows us to mock certain integration tests in case we have an issue within a unit of that component, further diminishing the potential risk for technology adoption issues. We will also be using GitHub actions for our automated testing and CI/CD workflows, and this in turn will expose us to be working with its YAML syntax, and to understand how tests are run in the cloud (how to set virtual a environment for this and replicate the project's to assert that all tests are performed in similar conditions).





## 7. Roles and Responsibilities

Team Lead
The team lead is responsible for overseeing the frontend development, backend development, and quality assurance teams. They coordinate tasks, set goals, assign roles, ensure communication, monitor progress, and ensure successful delivery of high-quality products.
Joaquín Badillo

Table 7.1. Team lead and role

Frontend Development				
The main responsibilities in this team will be the creation of UI/UX, as well as implementing user-facing features and ensuring a seamless and engaging experience for our users.				
Lead Iker Garcia	Alina Rosas	Fernanda Cantú	Cristina González	Patricio Bosque

Table 7.2. Frontend development members and roles

Backend Development			
This team will focus on building and maintaining the server-side logic, databases, and APIs that power our applications, ensuring scalability, reliability, and performance.			
Lead Andrés Tarazona	Valeria Martinez	Samantha Covarrubias	Sebastian Moncada
	Joaquín Badillo	Alejandro Arouesty	Samuel Acevedo

Table 7.3. Backend development members and roles

Quality Assurance		
This team will be responsible for testing the functionality, performance, and usability of our software products to ensure they meet the highest standards of quality before they are released to our users.		
Lead Rodrigo Núñez	Enrique Cabrera	Omar Rivera

Table 7.4. Quality assurance members and roles

## 8. Schedule

### 1. Scoping Phase (3 Weeks)

#### Week 1 to 3:

Roles & Responsibilities: Assign team members to project roles and define the decision-making process.

Definition of Requirements: Gather and analyze requirements to understand project scope and deliverables fully.

Budgeting: Develop a financial plan that covers all necessary resources, tools, and potential risks.

Technology Scouting: Research and select the best technologies and tools for the project based on requirements, budget, and team expertise.

### 2. Design Phase (2 Weeks)

#### Week 4 to 5:

Architecture Design: Outline the overall system architecture, including data flow, integration points, and external dependencies.

UI/UX Design: Design the user interface and experience, focusing on ease of use, efficiency, and accessibility.

Database Design: Structure the database to support scalability, performance, and security requirements.

Microservices and Interface Design: Define and design the microservices architecture and interfaces for internal and external communication.

Tests Design: Plan and design test cases for unit, integration, and user acceptance testing phases to ensure coverage and effectiveness.

### 3. Build Phase (9 Weeks)

#### Week 6 to 14:

Create Tests : Develop tests based on the Test-Driven Development methodology.

Create CI/CD Pipeline: Set up Continuous Integration/Continuous Deployment pipelines for automated testing and deployment.

Initial Frontend Development: Begin developing the frontend, ensuring that UI/UX designs are accurately implemented.

Initial REST API Development: Develop the RESTful API that will serve as the backend for the frontend, ensuring it meets specified requirements.

Microservice Development: Start the development of individual microservices according to the designed architecture.

Frontend and Backend Integration: Integrate the frontend with the backend services to ensure seamless interaction and functionality.

#### 4. Validation Phase (1 Week)

##### Week 15:

Unit Testing: Execute all unit tests to validate individual components for correctness.

Integration Testing: Perform integration testing to ensure that different components of the application work together as expected.

User Acceptance Testing: Conduct user acceptance testing with AWS to ensure the application meets business requirements and user expectations.

## 9. Defect Tracking

The Defect Tracking table serves to keep a ledger of past failed tests, recount which of them have been correctly fixed, and which of them still need fixing, while the testing is going on. For each test, we will document the *Finder*, *Fixer*, *Date*, and the *Name* of the test to check.

We will write them down within the following table:


Name of test	Finder	Fixer	Date of fixing

**Table 9.1.** Defect Tracking

## 10. Test Environment

### 10.1. Front-end

To test the desired behaviors of our frontend we have decided to use [SafeTest](#), a testing library used by *Netflix* that tries to simplify the implementation of tests. It can be used on top of [Vitest](#) and integrates interesting features from other libraries, for instance it includes [jest-image-snapshot](#) which allows tests to pass or fail if they match previous snapshots. It also simplifies the integration with context and providers, which might be cumbersome in



other environments like *React Testing Library* and it also simplifies authentication tests. For the sake of documentation, *SafeTest* also provides an easy and standardized way to show results through reports, which we expect to boost the efficiency of the Quality Assurance department. (Dinh, Ferret, Kolodny, et al. 2024).

## 10.2. Backend

Since we are building a backend with multiple microservices and serverless solutions in AWS, we will only be concerned with testing our own codebases or custom resources; these could be cloud functions (lambdas) or custom services (which shall be managed as containers). Depending on the underlying technologies, different testing environments shall be adopted, for instance if we were to solve an endpoint using the Go programming language we would create tests using the standard library; however a lambda written in the Node JS environment would be tested using Jest and a service developed in Rust would use cargo and “test functions”.

Notice that testing our own codebases is not equivalent to avoiding testing the rest of the services, as we will perform integration tests that test the interaction between our resources and the serverless solutions we have configured when relevant. For unit testing (and even local testing) we are aware of the importance of mocking the resources to better understand the origin of errors if these were to arise.

## 10.3. Local Testing Environment

As we are developing the project using a wide range of hardware, with varying amounts of RAM, bandwidth, CPU cores and even different CPU architectures, which we have set up with different operating systems and development environments, it is relevant to create a standardized way of testing, which we shall achieve using containers, in particular the Docker Engine which allows us to build, run and pull the same image from a repository and even limit the resources assigned to the container.

## 11. Entry/Exit Criteria

Testing	Entry criteria	Exit criteria
Unit test: Login of users	<ul style="list-style-type: none"><li>• A working computer.</li><li>• User credentials.</li></ul>	<ul style="list-style-type: none"><li>• A message response from the server</li></ul>

	<ul style="list-style-type: none"> <li>• A user pool or database system in a development environment.</li> <li>• Tester with access to the testing environment and relevant permissions.</li> </ul>	<p>presented on the frontend depending on the correctness of credentials.</p> <ul style="list-style-type: none"> <li>• A status code in the API response that is read by the frontend client to handle success or error states.</li> <li>• A message displayed in the frontend client or a redirect event to inform the user about a successful login or error.</li> <li>• Test results and any issues encountered are documented in the test report.</li> </ul>
Integration test: Test the interaction between a button component and a notification service in order to be able to intervene in a call so the supervisor is able to enter agents calls	<ul style="list-style-type: none"> <li>• A working computer.</li> <li>• Button component is implemented and integrated into the user interface.</li> <li>• Notification service is set up and running.</li> <li>• Supervisor's access privileges are defined and configured within the system.</li> <li>• Agents are actively making or receiving calls within the system.</li> <li>• Test environment is prepared with necessary dependencies and configurations.</li> <li>• Tester with access to the testing environment and relevant permissions.</li> </ul>	<ul style="list-style-type: none"> <li>• Successful notification sent to the supervisor upon clicking the intervention button.</li> <li>• Supervisor receives timely and accurate notifications for agent calls.</li> <li>• Supervisor is able to intervene in the call effectively through the provided options.</li> <li>• Error handling mechanisms are in place and functioning correctly.</li> <li>• Integration test report documenting the test steps, results, and any issues encountered.</li> </ul>
Unit Test: Management of permissions and access to	<ul style="list-style-type: none"> <li>• A working computer.</li> <li>• User roles are</li> </ul>	<ul style="list-style-type: none"> <li>• Supervisor users have unrestricted</li> </ul>

the dashboard according to the user's role	<p>defined and implemented in the system.</p> <ul style="list-style-type: none"> <li>• Dashboard interface is accessible and functional.</li> <li>• Permissions system is integrated into the dashboard.</li> <li>• Test environment is set up with necessary dependencies and configurations.</li> <li>• Tester with access to the testing environment and relevant permissions.</li> </ul>	<p>access to all dashboard features.</p> <ul style="list-style-type: none"> <li>• Regular users have access only to permitted features based on their role.</li> <li>• Access to restricted areas or actions is appropriately restricted for both supervisor and regular users.</li> <li>• Error handling mechanisms are in place and functioning correctly.</li> <li>• Unit test report documenting the test steps, results, and any issues encountered.</li> </ul>
Local test: Manages tests within a controlled environment among different operating systems before deployment.	<ul style="list-style-type: none"> <li>• Multiple working computers.</li> <li>• Test environment is set up with multiple operating systems</li> <li>• Application or software to be tested is installed and configured on each operating system.</li> <li>• Test data and environment configurations are consistent across all operating systems..</li> <li>• Tester with access to the testing environment and necessary permissions.</li> </ul>	<ul style="list-style-type: none"> <li>• Test cases have been executed successfully on all targeted operating systems.</li> <li>• Application behaves consistently across different operating systems, meeting functional and performance requirements.</li> <li>• Integration with system-level components or dependencies is functional across all operating systems.</li> <li>• System resource utilization remains within acceptable limits during testing.</li> <li>• Issues or discrepancies identified during testing are</li> </ul>

		<p>documented for further investigation and resolution.</p> <ul style="list-style-type: none"> <li>● Local test report documenting the test steps, results, and any issues encountered is prepared for review.</li> </ul>
<p>White box integration test: handle possible ask for help errors and notify accordingly</p>	<ul style="list-style-type: none"> <li>● A working computer</li> <li>● A set of defined errors that could occur upon asking for help</li> <li>● A mock for these responses to emulate behavior and test appropriate notification</li> <li>● A calculation for the cyclomatic complexity to evaluate against</li> </ul>	<ul style="list-style-type: none"> <li>● A set of assertions to handle error cases, and tests for the notifications</li> <li>● A report of the count of operations, looking to match the cyclomatic complexity</li> </ul>

**Table 11.1.** Entry/Exit Criteria

## 12. Deliverables

### - **Vision document**

Defines the scope and purpose of the project, which helps establish expectations and reduce risks.

### - **Communication plan**

Defines the information that should be communicated, as well as the frequency of communication, who will receive and deliver said information, and the channel of communication.


### - **Development strategy**

Action plan where the architecture of the system and the services that will be used are defined and described.

### - **Test document**

Documentation that describes the process, objectives, and results of software testing.

### - **Work breakdown structure**



Planning tool where work is divided into smaller tasks to make the workload more manageable, as well as aligning the plans for the scope, cost and schedule.

- **Gantt Chart**

Visual representation of the schedule and time allocated for each of the tasks in a project.

- **SRS (Software Requirement Specification)**

Document that rigorously describes the requirements of a software project, what it will do, and the expected results regarding performance and functionality.



## References

- Dinh, A. [andrewtdinh], Ferret, M. [andrewtdinh], Kolodny, M. [kolodny], Mitchell, V. [vmitchell] & Uminer. M. [mosheduminer]. (2024). *SafeTest*.  
<https://github.com/kolodny/safetest/tree/main?tab=readme-ov-file>
- Amazon Web Services [AWS]. (2024). *Mocking and testing*.  
<https://docs.amplify.aws/javascript/tools/cli/usage/mock/>
- American Express [GitHub organization] et al. [GitHub contributors]. (2023) [last commit to *main* 72ee1a0 December 11, 2023]. *Jest Image Snapshot*.  
<https://github.com/americanexpress/jest-image-snapshot>
- Kent C. Dodds et al. [GitHub contributors]. (2024) [last commit to *main* 0e8a058 March 6, 2024]. *React Testing Library*.  
<https://testing-library.com/docs/react-testing-library/intro/>
- Fu, A., Capeletto, M., Vitest Contributors. (2021). *Vitest*. <https://vitest.dev/>