Mobile Test Service Platform (Impact)

Rajeev Sebastian
Master of Science in Software Engineering
San Jose State University
San Jose, USA
rajeev.sebastian@sjsu.edu

Jacky Chen
Master of Science in Software Engineering
San Jose State University
San Jose, USA
jacky.chen@sjsu.edu

Pratik Baniya
Master of Science in Software Engineering
San Jose State University
San Jose, USA
pratik.baniya@sjsu.edu

Ruthwik Kadavakolanu
Master of Science in Computer Engineering
San Jose State University
San Jose, USA
ruthwik.s.kadavakolanu@sjsu.edu

***Abstract*– Impact enables users to test their mobile applications on physical phones and tablets. These are not emulators or simulators. This allows testing on both Android and iOS platforms where the application is native, hybrid or web. These devices are physically isolated from one another so these devices do not feed from each other or external devices via Wifi or bluetooth. These devices are dynamically tethered to the host machines that have the device plugged into over USB. This host machine is later torn down after usage such that the next test has no access over previous data whatsoever, fresh images. All the above features make this platform desirable to build a testing service which is more intuitive, easy to use and allows for provisioning of devices ahead of time.**

***Abstract*–Cloud, Test, AWS, Mtaas, Docker**

I. Introduction

The primary goal of our project is to provide an online platform allowing customers to test their software on mobile devices. Our objective is to provide these same customers with a seamless, interactive way to achieve their testing goals to make their software successful. In the end, our customer's success is our success because it speaks to the strength of our mTaaS.

In this report, we will walk you through our entire process starting from our system design all the way through to our end to end solution. By the end, you will have clarity on the benefits our solution provides to the market.

II. Design Methodology

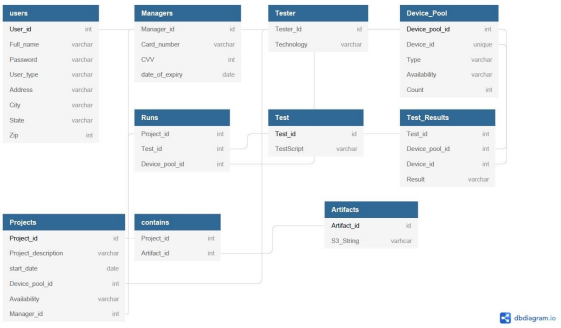

Figure 1: Database Design

The database design diagram shows all the components that we are managing. The users interact with the device farm through the web application for creating device pools, running tests and generating reports.The projects contain artifacts like the apks and the test cases , test plan etc.Each project is owned by a single manager.One device pool is associated with one

project other than the default device pool.Multiple testers work on multiple projects.A tester can run single test across the entire device pool.The results are given per device. The database closely mirrors the API created and the request responses generated.Since we are using mongo db for scalability of data the above RDBMS schema has been changed to document model of MongoDB.

Only 4 document types are present:

1. Manager

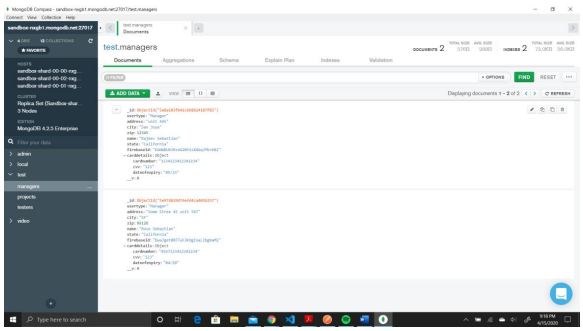

Figure 2: Manager Attributes

2. Tester

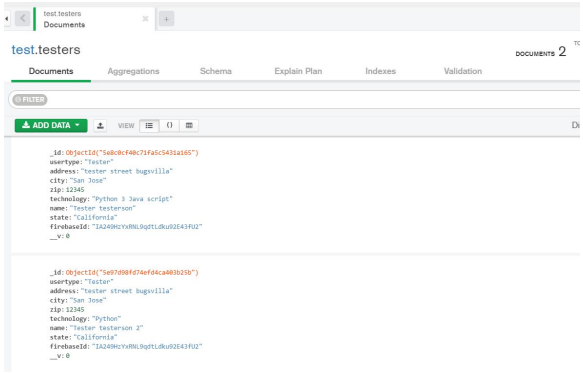

Figure 3: Tester Attributes

3. Project

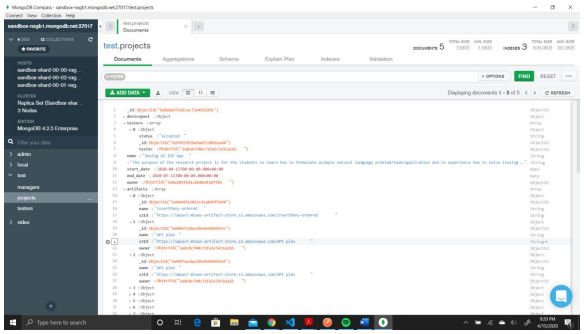

Figure 4: Project Attributes

4. Test Runs

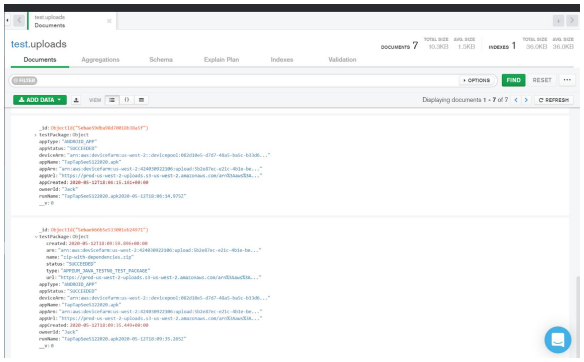

Figure 5: Test Run Attributes

III. Cloud-Based System Infrastructure and Components

A. AWS Device Farm

AWS device farm enables users to test their mobile applications on physical phones and tablets. The component Device Farm is fully managed and controlled via Amazon Web Services. They provide the developer SDKs for requesting resource pools on demand. The request coming to our application is something that can manage and control, so basically an abstraction layer. The plan of action is to develop a Device Farm service that receives and handles the request for our application.

B. User-Onboarding

The request from on-boarding and test stats dashboard will invoke API's to device farm service that we will be building.This component

is a web application which manages the creation of users and projects and is the middle ware sitting in between the backend and the Device Farm system. The various components are the frontend web application, the backend and the device farm from AWS. The web application will be hosted using dockerized containers on the ECS clusters. These provide auto scaling policies which will help in creating a scalable product. The front end is made using React running on Nodejs servers. For the backend we will be using the mongo DB database to store information of the various users like the managers and the testers. S3 servers will be used to store all the project artifacts like apks, test scripts. Additionally, there will be replication along the Mongo servers for high availability. Amazon device farm allows users to provision real devices to manipulate and test. We would be using the device farm API to handle all the allocation, deallocation and testing of the device.

B.1.1 Component Purpose & Objective

This shows the dependencies between the various components of the system which are Users, Web application, Device farm, Application servers and Database servers.The users are able to interact with underlying device farm through the web application. The requests are routed through a set of load balancers in order to scale well for a lot of users or requests.The application logic resides in EC2 instances which store the business logic for the functioning of the application with the MongoDB servers storing data related to the users, projects and results of the test runs. It is replicated for availability and fault tolerance. It interacts with the device farm on the basis of API requests.

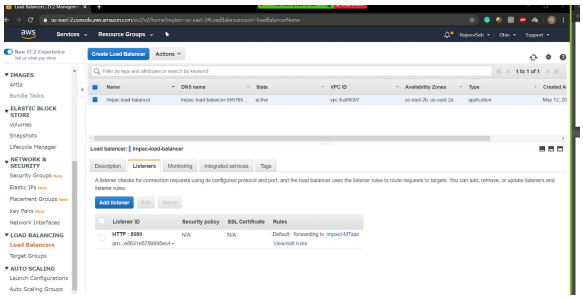

Figure: Load Balancer

B.2.1 Component Application Interface Design & Analysis

The following table shows the various apis used in the creation of the web app.We have made use of REST apis. Therefore each of these requests and responses are stateless.This reduces the load on the application server and we can create fat client applications with sessions.

URL	Method	Request	Response

/signup	**POST**	{
		"Fullname" : "Rajeev Sebastian",
		"Password": "Password",
		"Email" : "rajeev@gmail.com"
		"Type" : "Manager",
		"Address" : "Address",
		"City" : "San Jose",
		"State" : "California",
		"Zip" : 95126,
		"Card Number" : 12345678912345,
		"CVV" : 123,
		"Expiry" : "08/20"
		}
			{"Success": "User saved"}

/signin	**POST**	{ "Email" : "rajeev@gmail.com", "Password": "Password"	Made use of firebase to a handle authentication This allows us to focus on our product

Figure 6: REST APIS

		}	
/signout	**GET**	{}	{Success : "Done"}

/manager/:managerId/project/	**POST**	{
		"Project_Id" : "1",
		"Start_date": "03/10/2020",
		"End_date" :"03/12/2020",
		"Description" : "Information about the project"
		}	

/manager/:managerId/project/:projectId	**GET**		{
			"Project_Id" : "1",
			"Start_date": "03/10/2020",
			"End_date" :"03/12/2020",
			"Description" : "Information about the project"
			}

Figure 7: REST APIS

/manager/:managerId/project/:projectId/artifact/:artifactId	**GET**	{}	{ "Project_Id" : "1", "Artifact_id" : "S3 String" }
/manager/:managerId/project/:projectId/artifact/	**POST**	{ "Project_Id" : "1", "Artifact_id" : "S3 String" }	{ Success or Failure}
/manager/:managerId/project/:projectId/devicepool/	**POST**	{ "Device Pool Id" : "", "Devices" : [{ "Device_id" : "1", "Device_Name" :"Galaxy S3", "Device_type" : "Android", "Device_Availability" :"True", "No_of_devices" : 10 },{ "Device_id" : "3",

Figure 8: REST APIS

		"Device_Name" :"Galaxy S10", "Device_type" : "Android", "Device_Availability" :"True", "No_of_devices" : 12 },{ "Device_id" : "4", "Device_Name" :"Galaxy S20", "Device_type" : "Android", "Device_Availability" :"True", "No_of_devices" : 13 }] }	
/manager/:managerId/project/:projectId/devicepool/:devicepoolId	**GET**		{ "Device Pool Id" : "", "Devices" : [{ "Device_id" : "1", "Device_Name" :"Galaxy S3", "Device_type" :"Android", "Device_Availability":"True",

Figure 9: REST APIS

/manager/:managerId/project/:projectId/artifact/	**GET**		{ "Project_Id" : "1", "Artifact_id" : "S3 String" "Owner_id" : Refs manager }
/manager/:managerId/project/:projectId/test	**GET**		{"Tests" : [{"Testid" : 1,{"Testid" : 2}]}
/manager/:managerId/	**PUT**	{ "Fullname" : "Rajeev Sebastian", "Email" : "rajeev@gmail.com" "Password": "Password", "Type" : "Manager", "Address" : "Address", "City" : "San Jose", "State" : "California", "Zip" : 95126, "Card Number" : 12345678912345, "CVV" : 123, "Expiry" : "08/20"	{ Success or Failure}

Figure 10: REST APIS

/tester/:testerId/	**PUT**	{ "Fullname" : "Rajeev Sebastian", "Email" : "rajeev@gmail.com" "Password": "Password", "Type" : "Manager", "Address" : "Address", "City" : "San Jose", "State" : "California", "Zip" : 95126, "Card Number" : 12345678912345, "CVV" : 123, "Expiry" : "08/20" "Technology":"JAVA" }	{ Success or Failure}

Figure 11: REST APIS

B.3.1 Component Logic Design

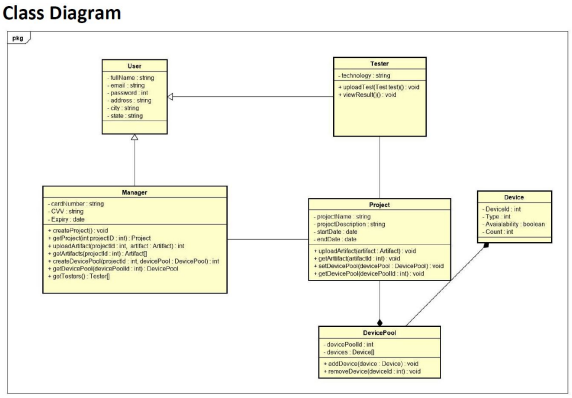

Figure 12: Class Diagram

The class diagram shows the structure of the components the attributes and the operations that the can performs on themselves and each other

User and manager are specialization of the base class users with additional attributes like Technology for the tester indication his preferred scripting language and the card details of the business manager who will handle the billing of the project.The manager can create a project and upload project files , acquire the device that are needed for the mobile application and see which testers are working in the given project. The project class handles the artifact and device management using the Device pool as a composed object which in turn has the device as a composed object. Therefore on deletion of projects these resources are automatically taken care of because of the way Composition works.

B.3.2 Component Function Description

1) User Login and Create Project functionality

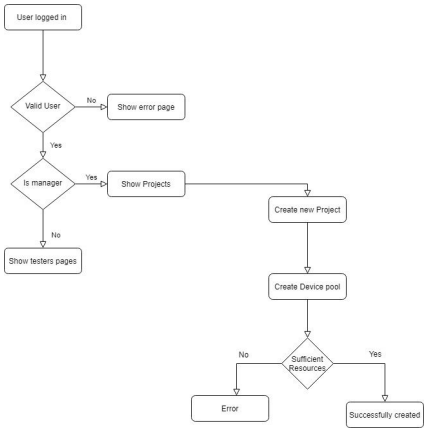

Figure 13: Component Functionality

2) Tester upload script and view results

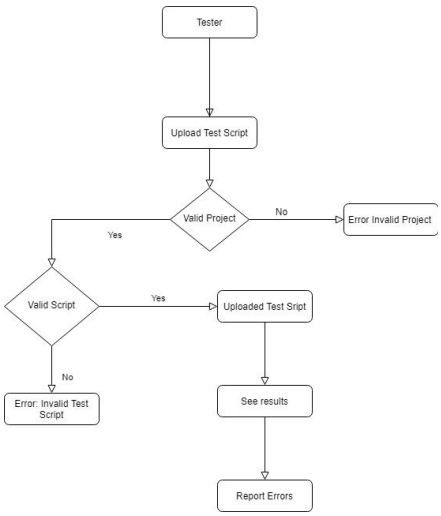

Figure 14: Component Functionality

3) Manager uploads Files

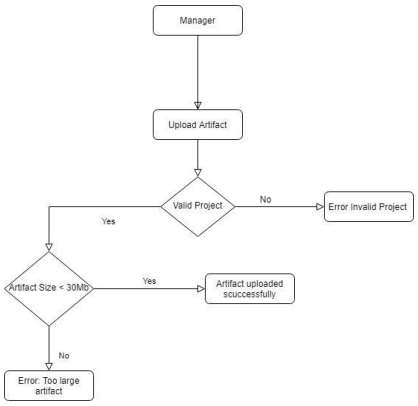

Figure 15: Component Functionality

B.3.3 Component Behaviour Analysis

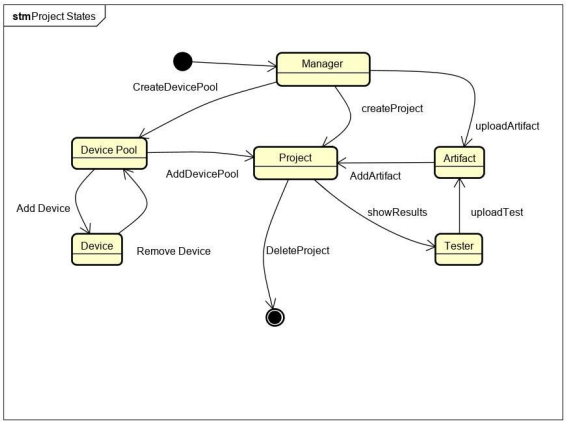

Figure 16: Component Behaviour

The above state diagram shows the different actors in the given component and how they interact with each other.All actors primarily interact with the project object which goes through different states during the course of the life of the component. Manager creates a project and assigns a device pool to it which causes the pool to allocate physical or virtual resources(mobile devices) to a project.Manager uploads artifacts like the application files associated configuration and Database files into a project whereas the tester uploads scripts.This triggers a test run whose results are then shown to the Tester Finally the project near the end of its lifecycle

C. Billing

This component supports book keeping and account management for billing for all of user groups. It supports a pay-as-you-go billing model based on the predefined cost model and metrics supported by our mTaaS platform. The billing microservice leverages the "costExplorer.getCostAndUsage" AWS service to come up with the cost an end user will incur for using our services.

A service to deliver insightful information to understand resource spending and predict future

customer resource deployment requirements. We need a mechanism to find resource usage and resource allocated per customer along with its cost.

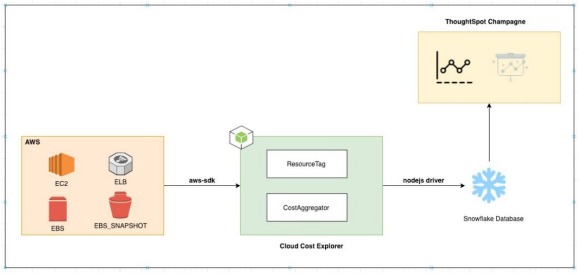

Figure 17: Cost Explorer Component

The idea behind writing cost explorer as a service is that it would be cloud agnostic.

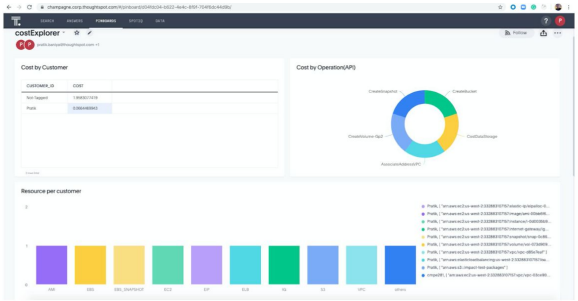

Figure 18: Cost Explorer UI

First part would be to scrape all the data based on tags then aggregate the data and push it to snowflake. Then make a connection between snowflake and ThoughtSpot, which is a BI tool, to display the data.

D. Test Runner

The test runner component handles file uploads, test executions, and test results. We also leverage Report Portal to manage bug tracking and use this as an additional feature displayed during test results.

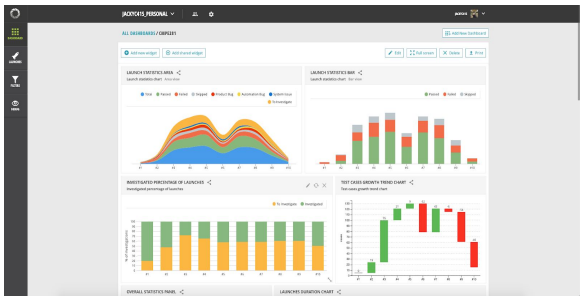

Figure 19: Test Runner UI

Test artifacts are streamed directly from AWS S3 and then integrated with the Report Portal with the use of AWS Lambda functions. We used this approach because it is highly scalable and highly available.

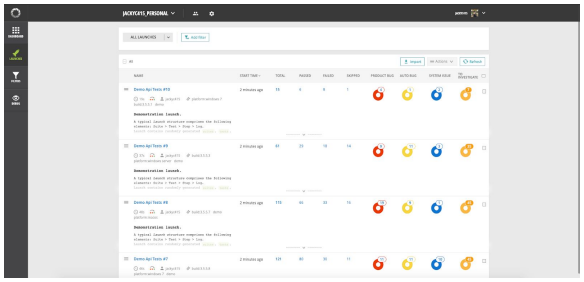

Figure 20: Test Runner UI

Our test runner component can also provide real time analytics and visualization of automated test results. With the bug tracking from Report Portal, the end user can establish fast traceability as well as accelerate their routine with our test result analysis.

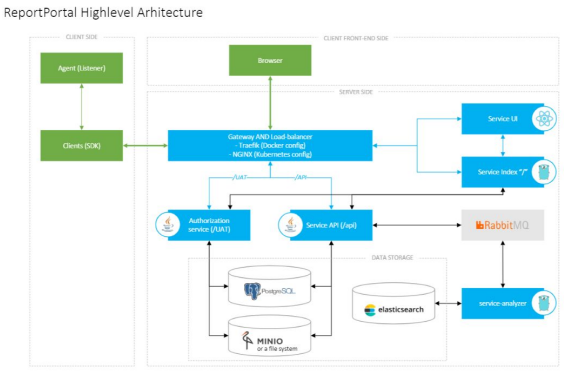

Figure 21: Report Portal Architecture

D.1.1 Component Purpose & Objective

The main objective of the test runner component is to interact with tester requests; to essentially accept an upload of an apk file which represents the mobile application that needs to be tested and any custom test scripts provided by the user to be able to execute such testing scenarios on the AWS device farm platform.

As such, users will upload these files through a web user interface portal upon successfully logging on to the application with SSO and then select the appropriate user authorization for testing. The server will receive the uploaded files as a request to be processed by AWS device farm platform which then have a callback function that renders the result back to the user interface and that will consists of various graphs and/or charts to illustrate the testing outcome, such as the total number of passed, failed, skipped, stopped, and so on of test cases.

In addition to the testing environment hosted by the device farm platform, an open source bug tracking management tool (ReportPortal) will be integrated onto the application to manage test automation and provide further analysis of the test outcome presented in a way that will enable the user to fully understand the result, manage bug tracking issues and accelerate project manager's routine analysis.

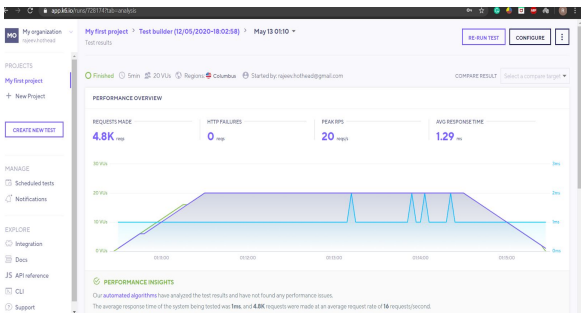

Figure: Performance Analytics

We have stress tested our applications using k6.io made 5K requests and 20 virtual users at a time And ensured that the response time is less than 2ms and on average 1ms.This is due to the autoscaling and ALB that we integrated into the application.

D.1.2 Component Scope & Usage

The scope of this component may be divided into simply three main aspects. To interact with the AWS device farm server, it is essential to first establish a web server that hosts the API gateway between device farm and the application server itself. Moreover, the application server shall also link to a web service interface through proxies to render the content on the client side.

```
var devicefarm = new
AWS.DeviceFarm();
devicefarm.createUpload(params,
function(err, data) {
  if (err) console.log(err,
err.stack); // an error occurred
  else     console.log(data);
// successful response
});
```

Figure 22: Device Farm Code

Above code snippet illustrates creating an instance of AWS Device Farm in the application server. An instance variable, device farm, is created to gain access to the API provided by AWS SDK. From here, the instance will call a method, createUpload, passing in certain parameters and have a callback function to retrieve the response data, which essentially acts as a constructor to initialize an infrastructure to enable testing on a pool of mobile devices.

```
var params = {
  arn: 'STRING_VALUE' /* required
*/
};
devicefarm.getUpload(params,functi
on(err,data) {
if (err){
 console.log(err,err.stack);// an
error occurred
}else{
console.log(data);// successful
response
}

});
```

Figure 23: Device Farm Code

Once the initial process is set up, device farms may request the user to upload certain files to the server and this may include .apk (android application package) file and/or custom test scripts. Apk files are essentially the mobile application itself that requires testing and test scripts are responsible for generating a set of conditions to be tested. Similarly, devicefarm instance will call another method, getUpload, also provided by AWS SDK, to implement this feature.

```
var params = {
  arn: 'STRING_VALUE' /* required
*/
};
devicefarm.getDevicePool(params,
function(err, data) {
  if (err) console.log(err,
err.stack); // an error occurred
  else     console.log(data);
// successful response
});
```

Figure 24: Device Farm Code

Once we get both application and test package file uploads, we will proceed to get the device pool. By default, test cases will run against Top Devices that are compatible. To obtain the device arn, we will implement getDevicePool API and provide it with the project arn as its parameters.

```
var params = {
  name: "MyRun",
devicePoolArn:
"arn:aws:devicefarm:us-west-2:1234
56789101:pool:EXAMPLE-GUID-123-456
", // You can get the Amazon
Resource Name (ARN) of the device
pool by using the list-pools CLI
command.
projectArn:
"arn:aws:devicefarm:us-west-2:1234
56789101:project:EXAMPLE-GUID-123-
456", // You can get the Amazon
Resource Name (ARN) of the project
by using the list-projects CLI
command.
  test: {
   type: "APPIUM_JAVA_JUNIT",
testPackageArn:
"arn:aws:devicefarm:us-west-2:1234
56789101:test:EXAMPLE-GUID-123-456
"
  }
 };
```

Figure 25: Device Farm Code

```
devicefarm.scheduleRun(params,
function(err, data) {
   if (err) console.log(err,
err.stack); // an error occurred
   else     console.log(data);
// successful response
var params = {
```

```
  arn: 'STRING_VALUE' /* required
*/
};
devicefarm.getRun(params,
function(err, data) {
  if (err) console.log(err,
err.stack); // an error occurred
  else     console.log(data);
// successful response
});
```

Figure 26: Device Farm Code

Once we obtain the four essential resource names: project, upload, testpackage and device arns, we will be able to schedule the official test run for the uploaded application and test files. Upon completion of AWS Device Farm testing, users will be able to retrieve the test results for each test case(s) provided to the platform. The callback function of the method, getRun, will have a condition to check if the test operation succeeded and thus retrieves the appropriate response data or throws an error exception upon failure and prints out the stack trace.

D.2.1 Component Application Interface Design & Analysis

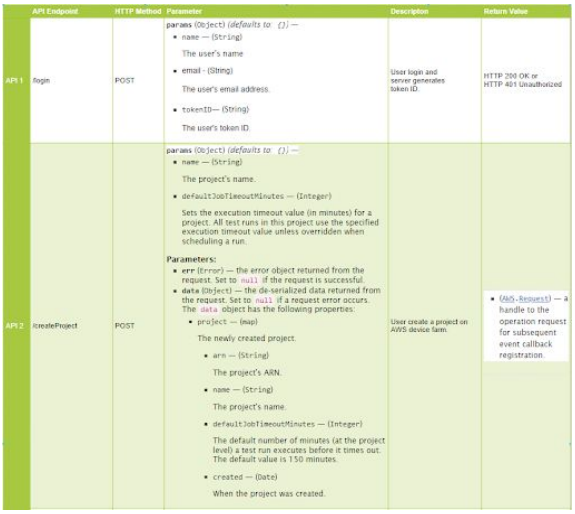

Figure 27: API Component List

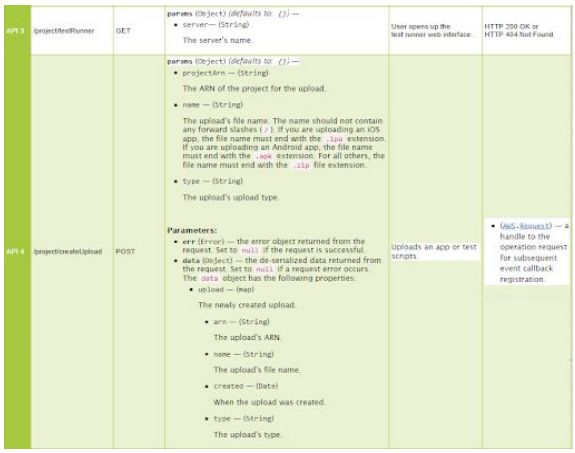

Figure 28: API Component List

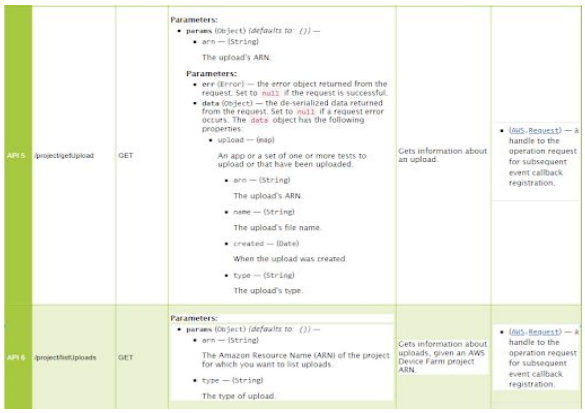

Figure 29: API Component List

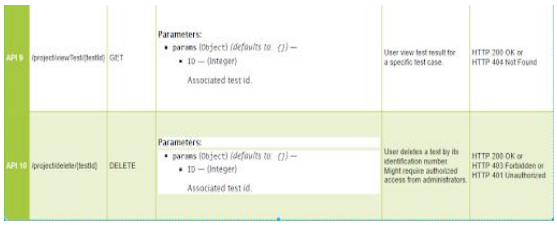

Figure 30: API Component List

D.3.1 Component Logic Design

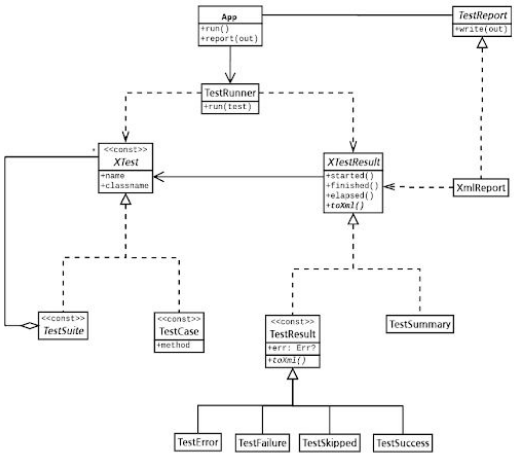

Figure 31: Component Logic

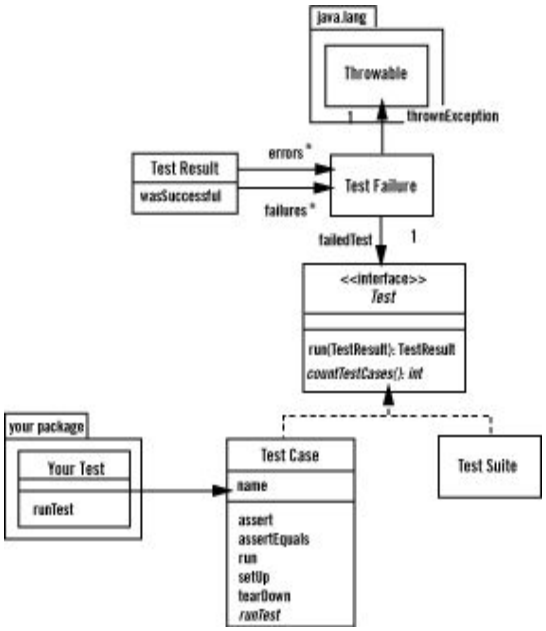

Figure 32: Component Logic

Logic design diagrams shown above illustrate the I/O process of the test runner. Application initiates the test runner class which triggers the Test class which is dependent on a test suite that consists of files, such as test scripts from the tester. Test Result class relies on the data supplemented by the Test class and these result data will be structured in various factors, such as test error, test failure, test skipped, and test success. These data will be formatted (parsed from JSON) and delivered (HTTP GET method) to the Test Report class where the web service interface will render the test outcome (dashboard) for each particular test case on the client side.

D.3.2 Component Function Description

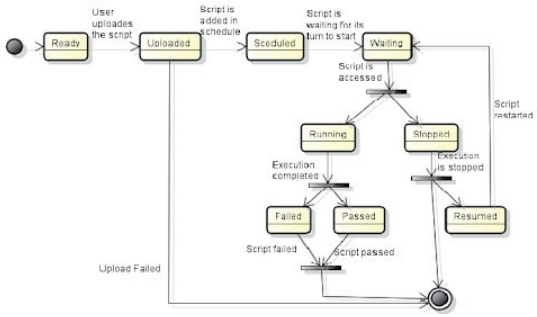

Figure 33: Component Flowchart

Following flowchart diagram illustrates the process of I/O and interactions between components in the test runner. The user starts by uploading a test file in the format accepted by AWS Device Farm; such as, an apk zipped with dependencies. This will put the test files in a queue, assuming the user has the ability to upload multiple test files for different projects (scalability).

As each test file gets prepared for execution, its status will either change to “RUNNING” or “STOPPED” based on certain criteria and/or exceptions. Suppose the script is terminated for a critical reason, say wrong format submission, the user will be alerted by the system requesting a re-upload to attempt the script processing again. Otherwise, as the script executes in the running stage, it will conduct various APIs to the AWS Device Farm server and fetch the necessary information back to the client.

Upon completion, there will be two possible scenarios for test results - “Failed” or “Passed” with different entities that demonstrate in-detail analysis of the outcome. To keep track of these bugs in the system, we will be relying on a bug tracker management tool, such as the open source bug tracking tool, Bugzilla. By utilizing this tool, it will trace through each possible test case and report defects by providing an updated status of the bug. Despite there being many open source tools for bug tracking management, there is a possibility we will develop our own which will be more suitable for overall design and architecture of our existing application.

D.3.3 Component Behaviour Analysis

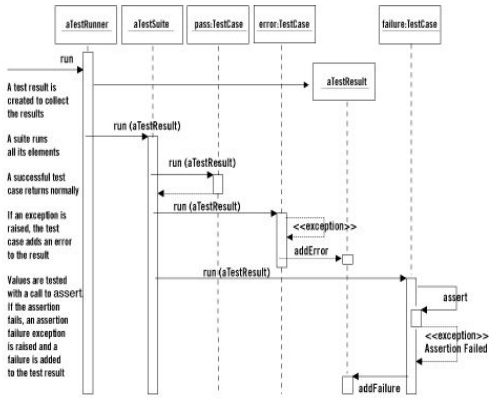

Figure 34: Behaviour Analysis Flowchart

D.4.1 Component Business Logic

The Decision table shown below illustrates some of the most common API requests that will be made on behalf of the user and produces various responses based on the validity of exceptions.

	Rules						
Conditions	1	2	3	4	5	6	7
C1. Server is running?	N	Y	Y	Y	Y	Y	Y
C2. Upload interface is present?		N	Y	Y	Y	Y	Y
C3. Upload request is processing?			N	Y	Y	Y	Y
C4. Request only accepts .apk files?				N	Y	Y	Y
C5. Response shows valid test cases?					N	Y	Y
C6. Response alert user if errors?						N	Y
Actions	1	2	3	4	5	6	7
A1. 500 Internal Server Error	X						
A2. 403 Forbidden		X					
A3. 409 Conflict			X				
A4. 406 Not Acceptable				X			
A5. 404 Not Found					X		
A6. 408 Request Timeout						X	
A7. 400 Bad Request							X

Figure 35: Business Logic Chart

There are six specific conditions that will most likely appear upon interacting with the test runner. **First condition** is to check whether the server is running. Without such a condition, it is impossible to even upload any files onto the web portal for processing. Suppose this condition is invalid, the most appropriate action is to throw a 500 Internal Server Error, which is a http error response that indicates the server has encountered certain problems that prevents a request from being fulfilled.

Second condition focuses on validating the client side implementation, specifically, the web interface, that exposes an action the user could take to interact with the server. It is to ensure an upload interface is present, meaning the user may see an upload functionality where the user can take a file from a local directory and push that file to the web portal. To access such an interface, more commonly known as “protected route”, a user must be authenticated by the server upon login to the application. Suppose this user tries to access the upload page without authorization, an http response error “403 Forbidden” will be thrown to prevent this action.

Third condition checks if an upload request is already under processing. Suppose a user uploads a file and then tries to re-upload the same or a different file, the server will likely

throw “409 Conflict” to prevent overloading of requests simultaneously.

Fourth condition prevents a user from uploading any extension files that are not associated with .apk. Since the test runner solely focuses on testing mobile devices on the Cloud, we must adhere to the following structure, an android application package. Therefore, if a user tries to upload, for instance, a PDF file, a response code of “406 Not Acceptable” will be thrown given that the request only accepts .apk files.

Fifth condition requires the response, or callback function from the server to present test cases outcome upon completion of AWS device farm testing. These test cases will be fetched from the server and sent as a response back to the client side for rendering. For example, when the device farm processes the request, it will generate a list of test outcomes that state the number of passed, failed, error, or stopped test cases. These JSON data will be present on the server where we could implement a REST API with the GET method to retrieve the data and format them prior to rendering on the web interface. For any reason this method fails, it will throw a response code, “404 Not Found” to indicate the data cannot be found.

Finally, the last but not least, **Sixth condition**, is to ensure the user or the “tester”, who uploads an.apk file and custom test scripts (if any), is being alerted for any error situations when something critically wrong is occurring in the back-end. Now, these alerts should be done in real time and that could be implemented with call-back functions. However, there are only so many alert messages a specific user could receive when it simply becomes overwhelming. Thus, to prevent the user from seeing all these alert messages, we will terminate the testing at a certain point (up to say 5 alerts) before we throw a “408 Bad Request” timeout response to pause any actions temporarily.

D.5.1 Component Graphical User Interface Design

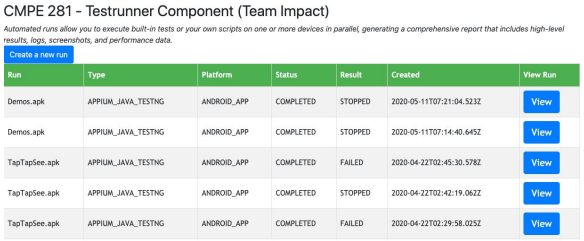

Figure 36: Test Runner GUI

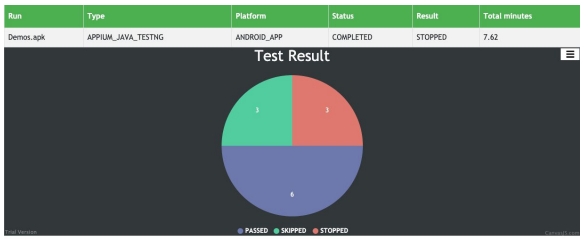

Figure 37: Test Runner GUI

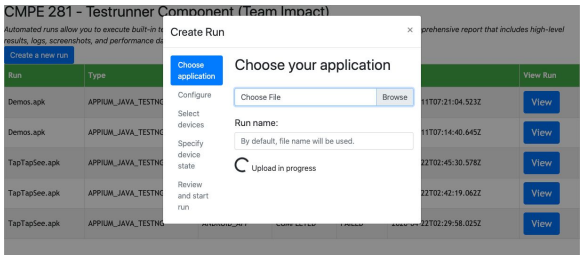

Figure 38: Test Runner GUI

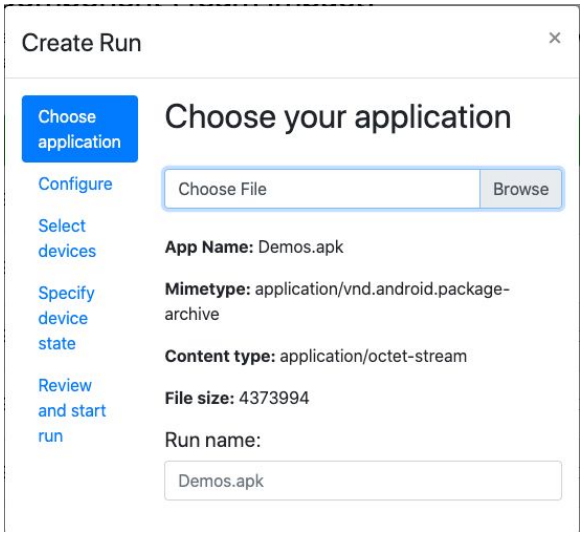

Figure 39: Test Runner GUI

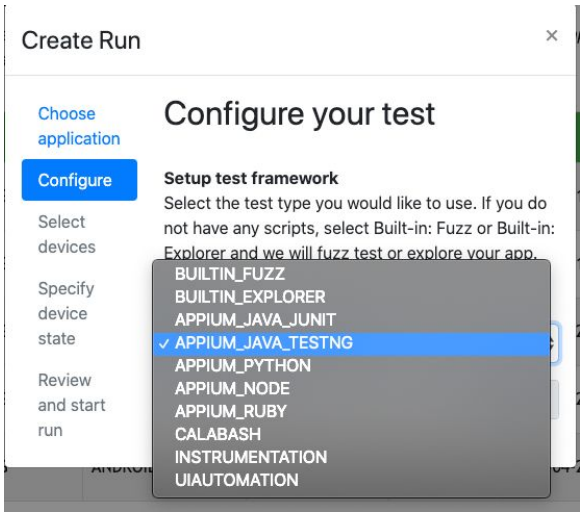

Figure 40: Test Runner GUI

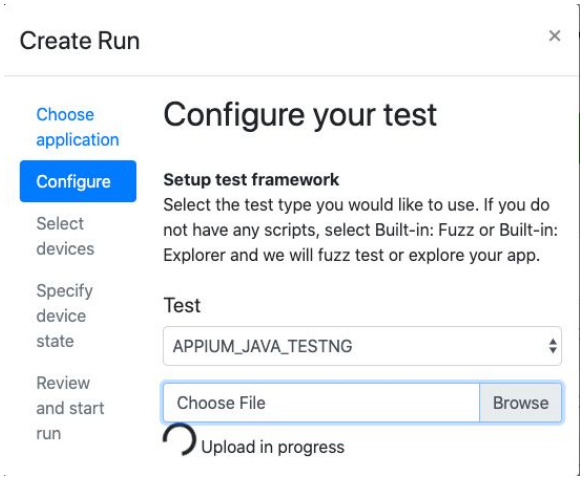

Figure 41: Test Runner GUI

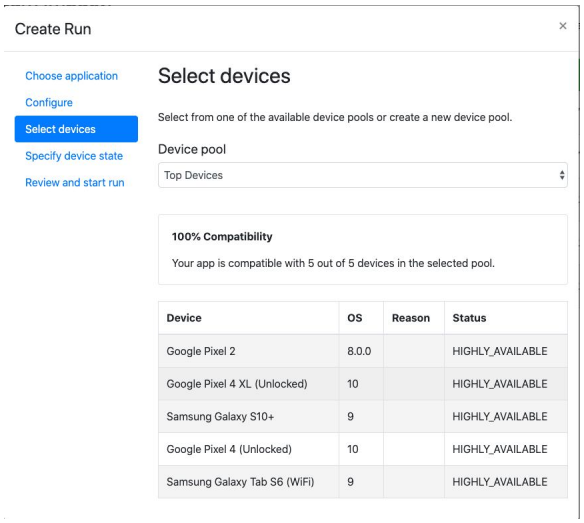

Figure 42: Test Runner GUI

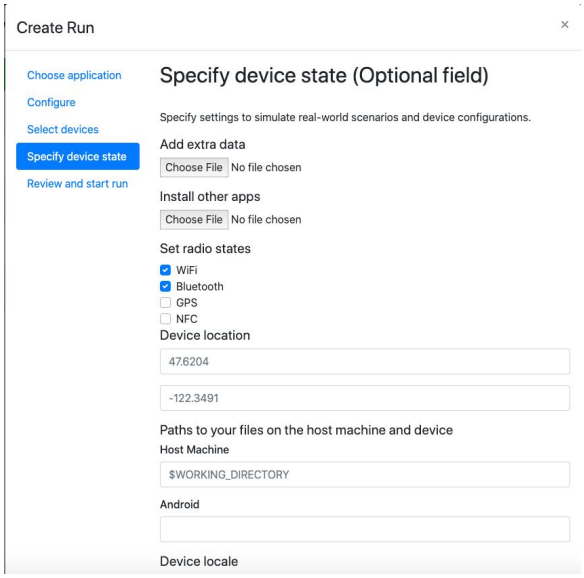

Figure 43: Test Runner GUI

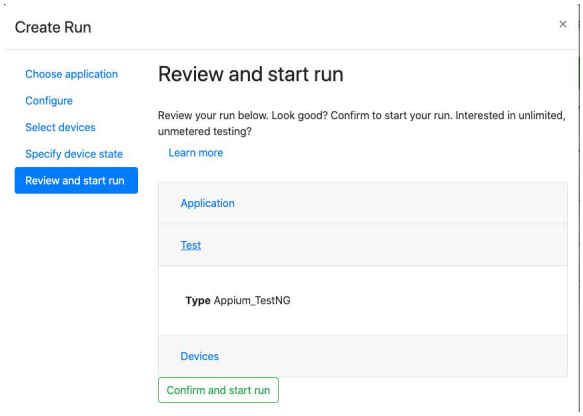

Figure 44: Test Runner GUI

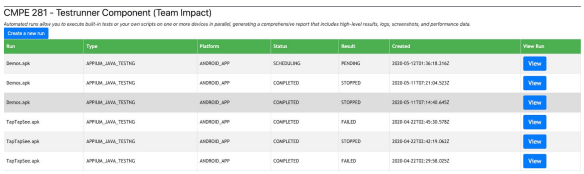

Figure 45: Test Runner GUI

III. System Architecture and Design

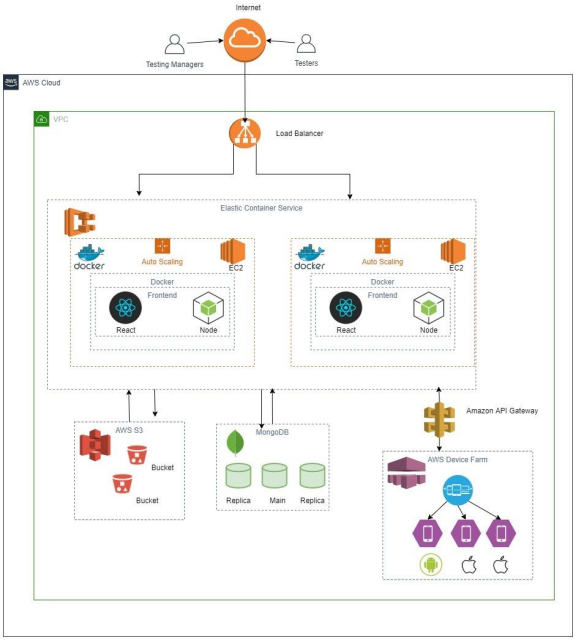

Figure 46: System Architecture Diagram

IV. Deployment

For production we are using kubernetes to scale our application as needed. We have also created kubernetes persistent volume to store the data on disc. We do not want it to be changed every time we re-deploy. Each of these Deployments are independent and thus can scale when there is high traffic.

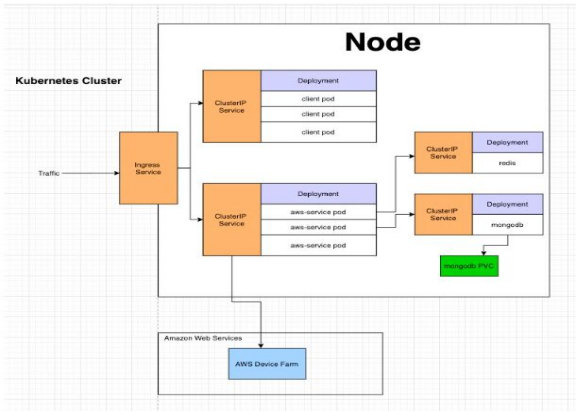

Figure 47: Kubernetes Node

The whole application can also be brought up using docker compose yaml file. This has been used for a local development environment.

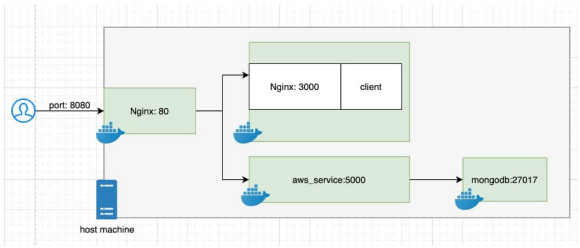

Figure 48: NginX Diagram

Load Balancing:

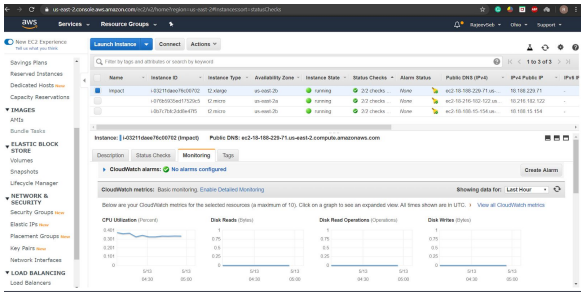

Figure: AWS Load Balancing

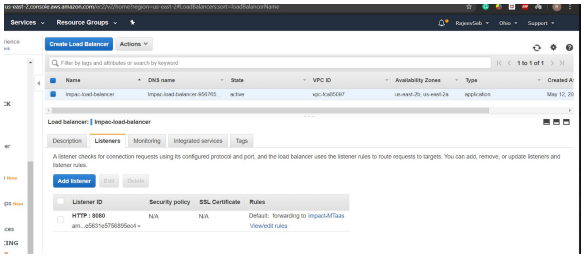

Figure: AWS Load Balancing

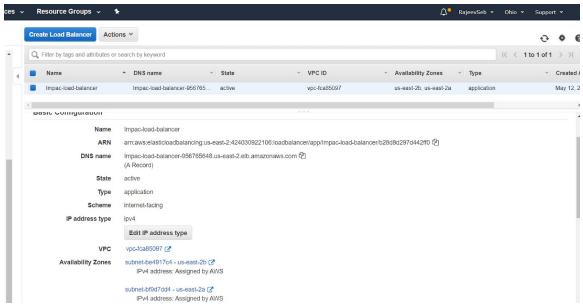

Figure: AWS Load Balancing

We have also used an application load balancer to ensure each server is being used efficiently. The load balancer redirects to the Nginx within the docker container.

Scalability:

We have used EC2 auto scaling groups to scale out our services.

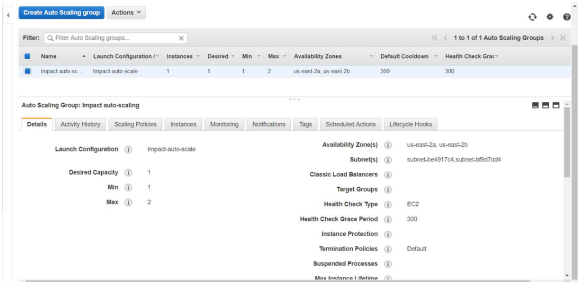

Figure: AWS Scaling

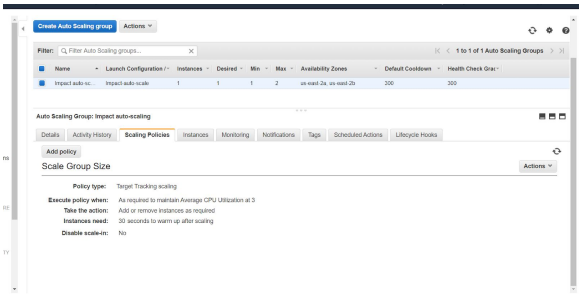

Figure: AWS Scaling

Multi-Tenancy:

We combine the following to achieve multi-tenancy. To handle large user session loads, we leverage Firebase. We also use NginX to handle network load balancing. We vertically scale using ECS. Finally, we use MongoDB for vertical and horizontal scaling. Combining everything mentioned so far, we use Security Group to ensure that we are able to have multi-tenancy with our mTaaS platform.

V. System GUI Design and Implementation

For our system GUI, we chose to design and implement with React and Bootstrap 4. React JS is a market forerunner when it comes to Front End development and we wanted our GUI to be as seamless as possible for the end user. React also allows for quick integration of additional features so that we can quickly meet the demands of our end users should they require modifications to our system GUI.

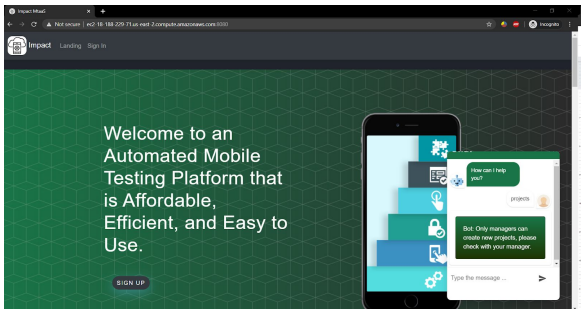

Figure 49: Landing Site GUI

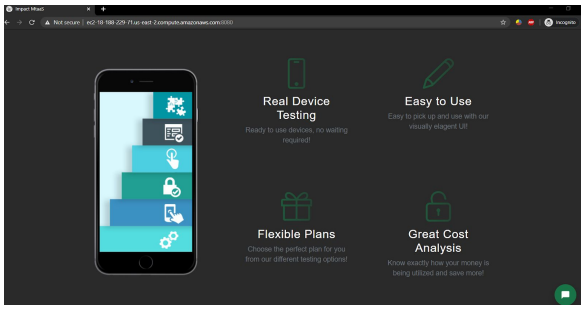

Figure 50: Landing Site GUI

The above is the landing page for our website. We made stylistic choices mimicking popular and successful websites and integrated those components to make sure our users felt in their natural habitat while working with a new service. Just because our end user has not used an mTaaS platform before does not mean they need to start from scratch. We integrated all the popular and appropriate features of most testing services while also keeping in mind which type of user had access to which types of services.

Manager Views:

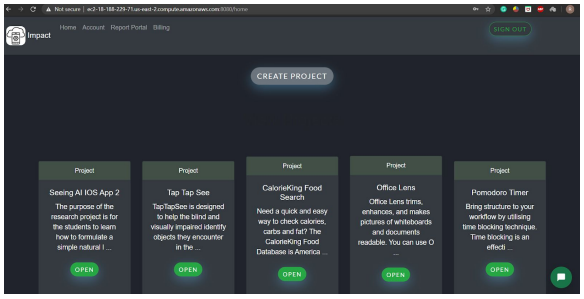

Figure 51: Manager Portal GUI(Create Project)

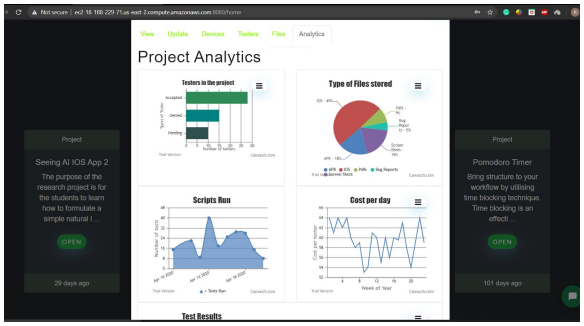

Figure 52: Manager Portal GUI(Analytics)

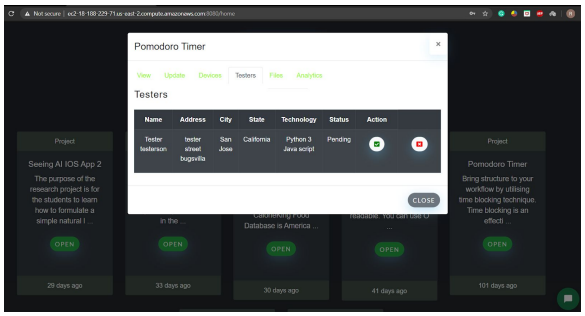

Figure 53: Manager Portal GUI(Tester)

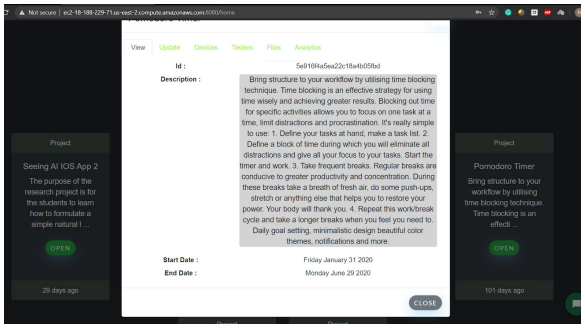

Figure 54: Manager Portal GUI(Project details)

Tester Views:

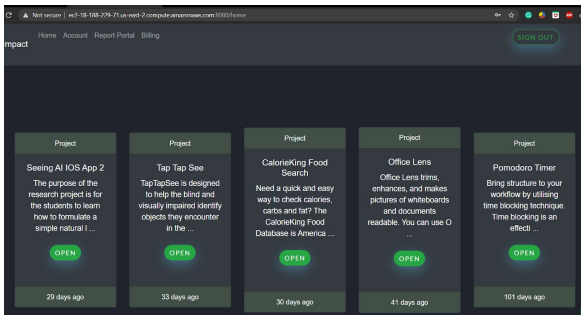

Figure 55: Tester Portal GUI (Projects)

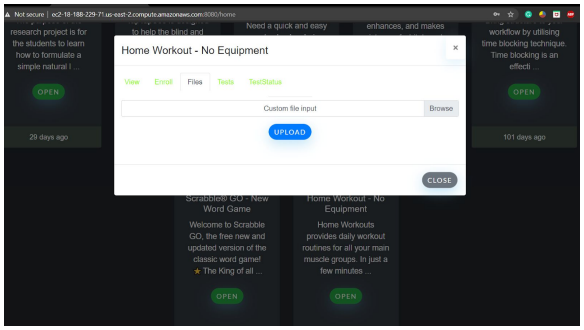

Figure 56: Tester Portal GUI (Add files)

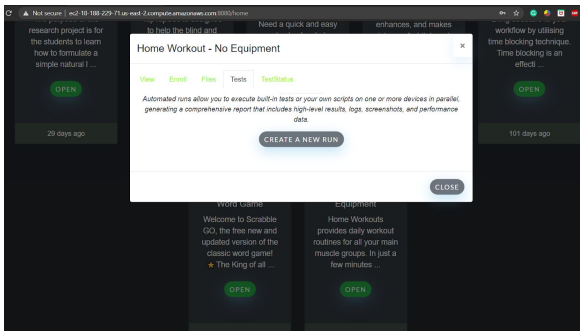

Figure 57: Tester Portal GUI (Create Run)

Figure 58: Tester Portal GUI (Run Status)

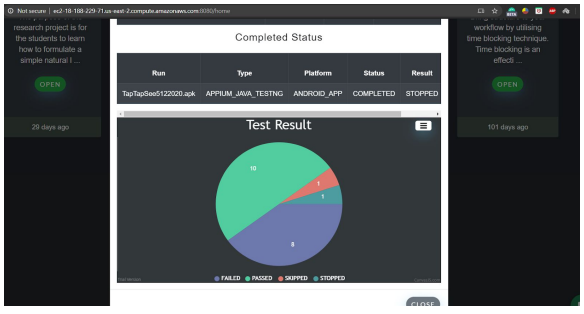

Figure 59: Tester Portal GUI (Test results)

As you can see above, the design flows of both the tester and the manager follow industry leading examples and allow our end users to leverage their existing knowledge in aiding themselves to successfully using our platform. This allows them to meet their needs quickly and effortlessly.

VI. Conclusion

Mobile testing as a service (mTaaS) is a quickly emerging feature that will be a mainstream technology in the very near future. Our service recognizes this importance and not only provides an end to end solution, it also allows for the quick expansion of new components and features thanks to our design choices. We have essentially engineered a fluid and elegant solution that will not only allow our end user achieve success in their business goals, it also allows for companies to leverage our platform and customize it to fit their needs so that their employees (the end users) can conduct their jobs effortlessly.

REFERENCES

1. https://d1.awsstatic.com/whitepapers/microservices-on-aws.pdf
2. https://blog.zhaw.ch/icclab/dynamic-rating-charging-billing-for-cloud-a-micro-service-perspective/
3. https://github.com/icclab/cyclops-billing
4. https://d2908q01vomqb2.cloudfront.net/7719a1c782a1ba91c031a682a0a2f8658209adbf/2017/08/10/devsecops2_1.jpg
5. https://docs.aws.amazon.com/devicefarm/latest/APIReference/devicefarm-api.pdf
6. https://docs.aws.amazon.com/awsaccountbilling/latest/aboutv2/billing-example-policies.html#example-policy-pe-api
7. https://par.nsf.gov/servlets/purl/10092500
8. https://www.semanticscholar.org/paper/A-cloud-based-TaaS-infrastructure-with-tools-for-Gao-Manjula/31a27ac16f74b0d8fb74db0edc252033418bb5d1
9. https://docs.aws.amazon.com/devicefarm/latest/developerguide/how-to-create-test-run.html
10. https://docs.aws.amazon.com/devicefarm/latest/developerguide/how-to-use-reports.html
11. https://dzone.com/articles/introduction-to-running-android-automation-applica
12. https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DeviceFarm.html#listJobs-property
13. https://aws.amazon.com/blogs/devops/ui-testing-at-scale-with-aws-lambda/