Technical Guide
Coding Test Platform

7th May 2021

Students:
Janesh Sharma - 17473124
Paul McNally - 17318221

Supervisor:
Dr. Stephen Blott

Table Of Contents

Overview

The main goal of this project was to build a coding test platform for companies to use as a means to assess potential new hires. We built a live coding environment allowing candidates to complete a coding test along with the option to complete video responses to questions proposed by an employer. In addition to this employers will have access to a results section where participants code solution and analytics about candidate's submissions are give(i.e. CPU time and memory used). This application was built using the MERN stack along with docker and is hosted on Firebase and google cloud.

Glossary

- **React** - Open-source, front end, JavaScript library for building user interfaces or UI components. It is maintained by Facebook and a community of individual developers and companies [2].
- **Node.js** - Open-source, cross-platform, back-end, JavaScript runtime environment that executes JavaScript code outside a web browser.
- **Express.js** - Back end web application framework for Node.js [3].
- **MERN** - Stands for MongoDB, Express, React, Node, after the four key technologies that make up the stack.
- **CI/CD** - Refers to the combined practices of continuous integration and either continuous delivery or continuous deployment. CI/CD bridges the gaps between

development and operation activities and teams by enforcing automation in building, testing and deployment of applications.

- **Cloud Function** - Serverless execution environment for building and connecting cloud services. With Cloud Functions you write simple, single-purpose functions that are attached to events emitted from your cloud infrastructure and services. Your function is triggered when an event being watched is fired.
- **WCAG 2.1** Standards - Web Content Accessibility Guidelines (WCAG) 2.1 defines how to make Web content more accessible to people with disabilities. Accessibility involves a wide range of disabilities, including visual, auditory, physical, speech, cognitive, language, learning, and neurological disabilities [1].
- **Firebase** - Platform developed by Google for creating mobile and web applications [5].
- **MongoDB** - Cross-platform document-oriented database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with optional schemas [6].
- **MongoDB Cloud Atlas** - Global cloud database service for modern applications. Deploy fully managed MongoDB across AWS, Google Cloud, and Azure with best-in-class automation and proven practices that guarantee availability, scalability, and compliance with the most demanding data security and privacy standards [5].
- **Judge0** - Robust, scalable, and open-source online code execution system that can be used to build a wide range of applications that need online code execution features [7].
- **Jest** - Jest is a JavaScript testing framework maintained by Facebook, Inc. designed and built by Christoph Nakazawa with a focus on simplicity and support for large web applications [8].
- **Docker -** Docker is a set of platform as a service products that use OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries and configuration files [9].
- **Cypress -** Cypress is a next generation front end testing tool built for the modern web. Cypress enables developers to write end to end tests [10].
- **Google Cloud -** Google Cloud Platform, offered by Google, is a suite of cloud computing services that runs on the same infrastructure that Google uses internally for its end-user products [11].
- **Nginx -** Nginx, stylized as NGINX, nginx or NginX, is a web server that can also be used as a reverse proxy, load balancer, mail proxy and HTTP cache. [12]
- **RCE -** This refers to our Remote Code Execution API. [13]
- **Redis** - Redis is an open source in-memory data structure store, used as a database, cache, and message broker. [14]
- **SendGrid** - SendGrid is an SMTP service provider that allows you to send email globally. [15]
- **Firecracker** - Firecracker is an open source virtualization technology that is purpose-built for creating and managing secure, multi-tenant container and function-based services. [16]
- **Kubernetes** - Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications. [17]
- **Postman** - Postman is a collaboration platform for API development. [18]

1. Motivation

Our main motivation for this project was having both project members used various coding test web applications such as hackerrank and codility as well as DCU's Einstein. We had an interest in how these applications work, in particular the remote code execution.

Another motivation was that we knew this project would involve a lot of technologies and self learning e.g how to implement docker and use containers. The many challenges that this project would present to us over the course of the academic year as well as the opportunity to build a useful and interesting application that is increasingly used by software companies hiring staff was enough to excite and interest us in carrying out this project.

2. Research

2.1 Frontend

For the frontend we used REACT. When doing research about react we learned about material ui. Material UI is a component library for React. Material UI allows us to import and use different components to create a user interface in our React applications, it is also beneficial that Material UI Components were built with WCAG 2.0 Level AA in mind.

2.2 Backend

The backend dynamic Express and Node.js application is hosted on a Firebase cloud functions in a microservices design pattern.

We chose to use microservices due to their simplicity and benefits as listed below:

- **Scalability** - The microservices are managed by the platform and can scale infinitely subject to the resources available to the platform, in this case Google.
- **Don't have to manage your own Server** - The microservices are fully managed by Google so you don't have to worry about setting up and managing your own server or handling load balancing and scaling.
- **Better isolation of errors which makes the application more resilien**t - in a traditional server one error could cause the entire server to stop but in microservices only that one service would fail and the rest would work fine assuming they do not depend on one another.
- **Easier to update or remove services.**
- **More secure**, interact with secure apis.
- When creating a microservices-based application, developers can connect **microservices programmed in any language**. They can also connect microservices running on any platform. This offers more flexibility to use the programming languages and technologies that best fit the needs of the project and the skillsets of your team.
- **Faster Time to Market** - Don't have to manage your own server or deal with configuration or setup of a server thus all those problems are eliminated, saving time and allowing you to deliver software faster.

2.3 Remote Code Execution

Originally before beginning this project we thought we would use an API (judge0) for the remote code execution, however after developing the main backend and frontend we decided to improve the scope of the entire project and implement the remote code execution ourselves, therefore we conducted research around docker containers and how they could be used for remote code execution.

2.4 Database

As we had decided to build this application using MERN stack we knew that we would be using MongoDB, so we were required to learn about and how to use MongoDB's query language. During our research we learned about MongoDB Atlas. MongoDB Atlas is a fully-managed cloud database developed by the same people that build MongoDB. Atlas handles all the complexity of deploying, managing, and healing your deployments on the cloud service provider of your choice (our choice was google cloud platform). We decided to use MongoDB Atlas as it's simple simplicity and service helped us to host the database online and speed the development process.

2.5 Creating a production Server

We wanted the entire application to be online, thus we hosted the remote code execution API on an Ubuntu virtual machine on Google Cloud Platform. This involved a lot of research in how to serve the application using Nginx as a web server and a reverse proxy. Using ssh to access the virtual machine and expose the necessary ports in the firewall to allow external access to the API. Also using SSl to encrypt traffic and set the necessary configuration in Nginx.

2.6 Gitlab CI & Testing

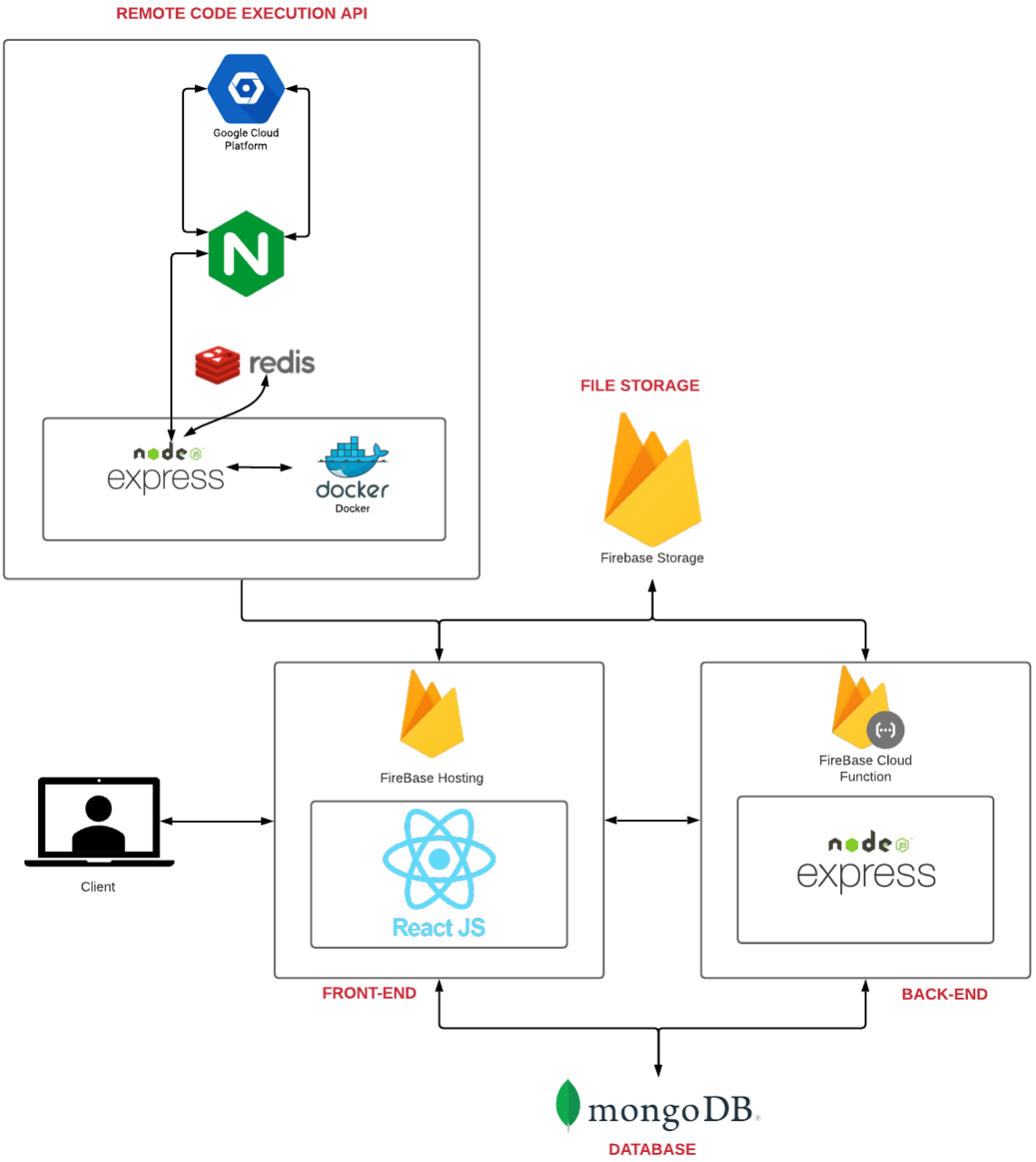
Significant research was conducted to best understand how to use the Gitlab pipeline to automate tasks such as build, test and deployment. Testing was also a considerable research effort to understand how to perform unit, integration and end to end tests on the system, the best practices in doing so and finally how to integrate the testing into the Gitlab pipeline.

3. Design

3.1 System Architecture Diagram

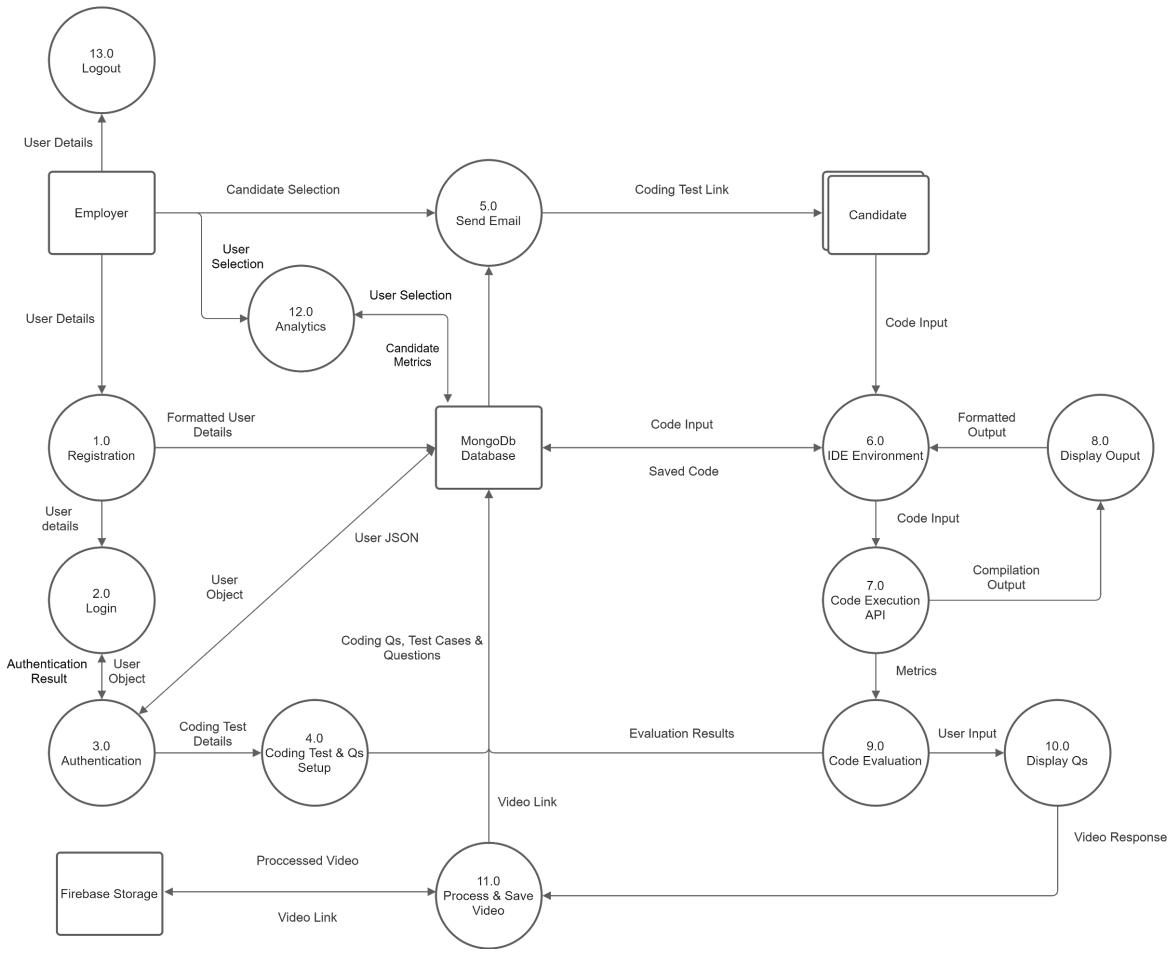
From looking at the diagram below you can see the overall architectural design of the system. We used the MERN stack architecture with Firebase for hosting, backend and storage, where we have the React static frontend application hosted on Firebase hosting and the backend dynamic Node.js application hosted on a Firebase cloud functions in a microservices design pattern. Additionally the database will be a NoSQL database provided by MongoDB Cloud Atlas.

The Remote code execution API component of our system has a nodejs API which executes code remotely in a docker container which is destroyed upon completion on return of code output. The design of the API incorporates a concurrent producer consumer pattern in which incoming requests to the server add valid submission to a redis queue, this redis queue is pulled from by a pool of workers in a first in first out order (FIFO). This allows the server to run effectively even under heavy load as submissions are queued and only executed once the pool of workers are finished their current task. The workers can easily be scaled to support higher concurrency by simply changing an environment variable to define the number of workers in the pool. The API is hosted on an ubuntu virtual machine on Google Cloud Platform with nginx as a web server and reverse proxy. The API is also ssl certified using a self signed certificate.



3.2 Data Flow Diagram

A data flow diagram or DFD represents the flow of data of a process or system. The DFD provides information about the outputs and inputs of each entity and the process itself.



External Entities

From looking at the diagram above you can see there are four external entities, namely:

1. **Employer** - This represents a person from the company reviewing candidates.
2. **Candidate** - This represents a person chosen to perform a coding test for the Employer.
3. **MongoDb Database** - This represents the MongoDB database which will be used to store various details such as user data, analytics etc.
4. **Firebase Storage** - This represents the Firebase file storage bucket within which candidate video responses will be stored for future viewing by the Employer.

Processes

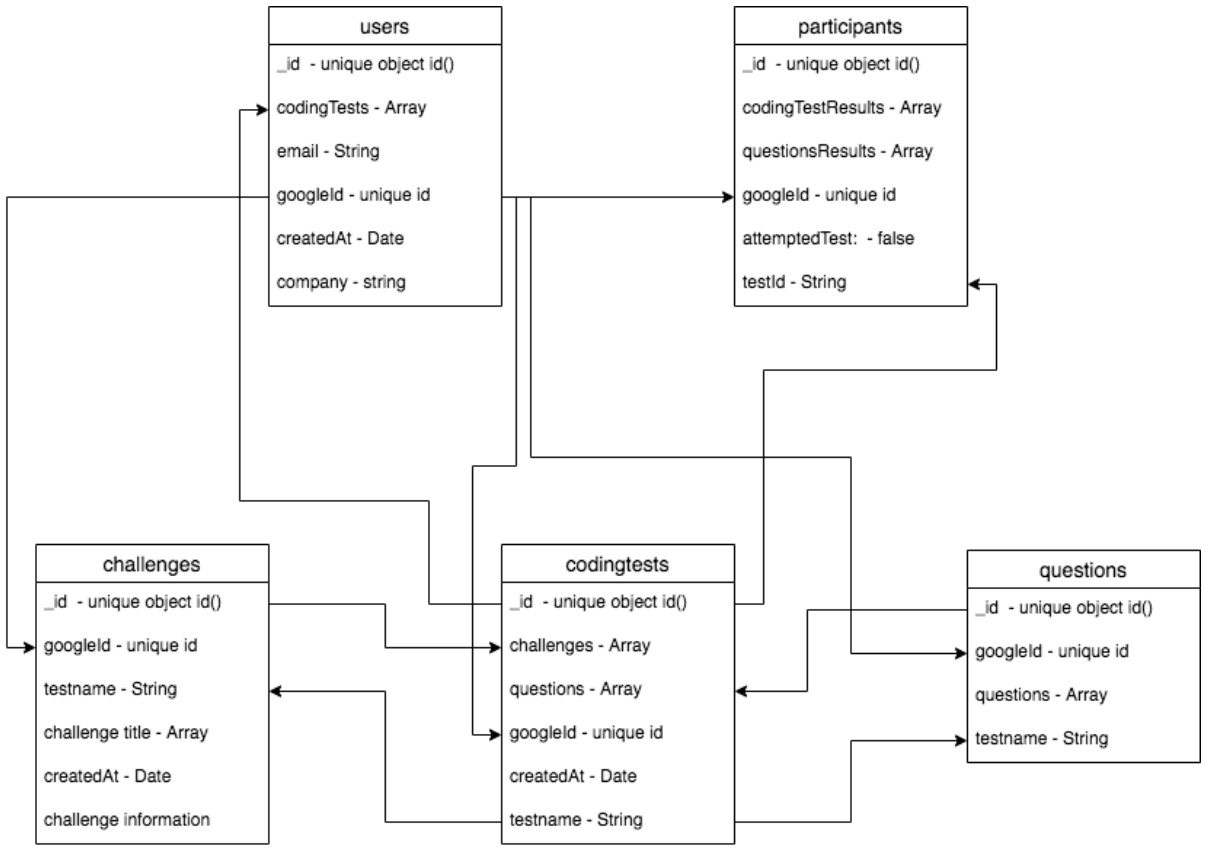
1. **Registration** - The user enters details which are then stored in the database.
2. **Login** - The user enters user details and sends the data for authentication.
3. **Authentication** - The user details are authenticated against the details stored in the database and the user is authenticated depending on the result.
4. **Coding Test & Qs Setup** - The user inputs the coding question in text format along with test cases and any questions for the candidate.
5. **Send Email** - The user selects chosen candidates and provides them with a link to the coding assessment in an email.
6. **IDE Environment** - The candidate is given a link which brings them to the coding test IDE with syntax highlighting.

7. **Code Execution Environment** - This process compiles code and returns the output, and metrics such as cpu time and memory used.
8. **Display Output** - The code compilation output is displayed for the candidate to show the output of the code they executed.
9. **Code Evaluation** - The code is evaluated against all the test cases and the results are stored in the MongoDB database.
10. **Display Qs** - The questions set by the employer are displayed to the candidate who must provide a video response to each question.
11. **Process & Save Video** - The video is pre-processed to reduce size and required storage space and sent to Firebase Storage to be saved, then a link is stored in MongoDB to reference it at a later time.
12. **Analytics** - The candidates for a particular coding test will be displayed and ranked according to performance metrics such as cpu time and memory used which tells us the overall efficiency of a solution.
13. **Logout** - The employer will be logged out of their account.

3.3 Database modelling

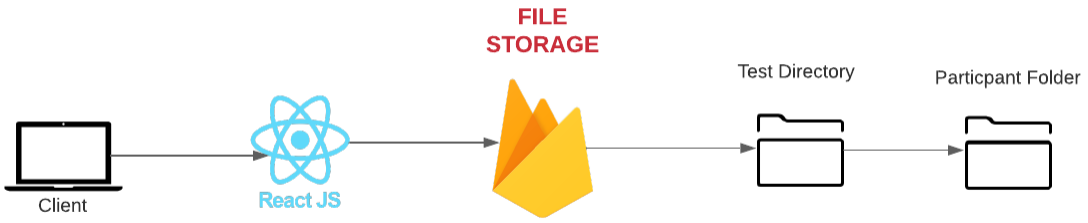
3.3.1 MongoDB

Our MongoDB cluster consists of 5 collections that can be communicated with via our backend API. The Entity relationship diagram below shows the relationship between the documents of each collection in the cluster. For each collection we decided to use the googleId of the user as the foreign key. The user collection stores the googleId, company name, as well as an array of all the object ids of the coding test documents created by that user. The coding test collection stores the test name, object ids of the individual coding challenges and question documents as well as the googleId of the user, this was important for the company ui so the API could get all the coding tests created by a particular user with a single find many query. The challenge document contained all the information about each challenge(i.e. Title, problem description, test cases) as well as the googleId and the name of the coding test it belonged to, this was again so all challenges could be returned to the api with a single find many query. The questions collection model was modelled the same way as the challenge collection with the googleId and test name and an array containing the questions to be asked in the video interview. A participant collection is modelled so that each document contains the googleId of the user, testId of the test (this is important if the user wants to reset a test a delete all current results), email of the participants as well as the the participants coding test results when they submit their code.



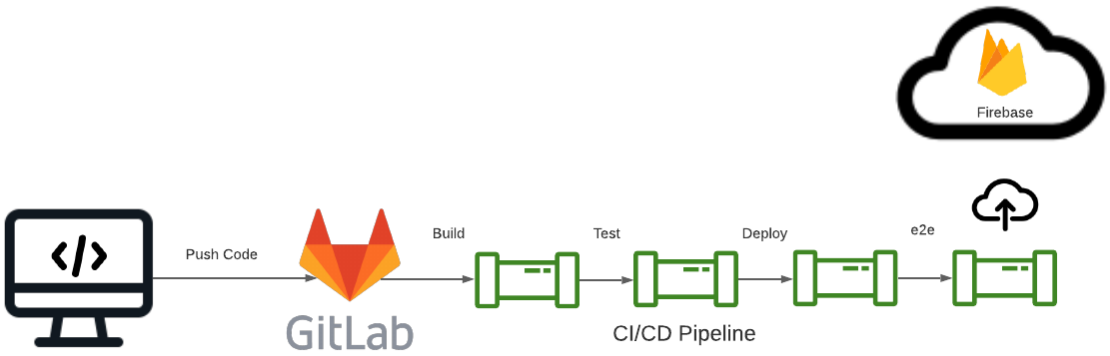
3.3.2 Firebase Storage

As MongoDB cannot store videos, we decided to use firebase storage to store the video recordings from the video interview. In Firebase storage each coding test has a directory with the mongoDB coding test object id of the document being the name of the directory to ensure uniqueness. Within each directory there are folders again with the mongoDB participant object id of each of the participant documents being the name of the folders to ensure uniqueness again. And within each of these folders are the video files which are later retrieved to be displayed in the results page of the frontend.



4. Implementation

4.1 Continuous Integration / Continuous Delivery



We utilised devops practices to incorporate automation in the development lifecycle to shorten the development process and provide continuous delivery with high software quality. As seen above, all stages of the application including build, test, deployment and production e2e tests were automated in a CI/CD pipeline in GitLab upon commit or merge to the repository.

4.2 Building the Database

Implementation of the database involved creating a mongoDB cluster and linking it to the backend of our application.

```
const mongoose = require('mongoose');
const functions = require('firebase-functions');

class ConnectMongo {
 static init() {
   const uri = functions.config().app.atlas_uri;
   mongoose.connect(uri, {
     useNewUrlParser: true,
     useCreateIndex: true,
     useUnifiedTopology: true,
   });
   const connection = mongoose.connection;
   connection.once('open', () => {
     console.log('MongoDB connection established');
   });
 }
}
```

The URI for MongoDB is stored in firebase as a private configuration key. Once this was set up we began modelling the database, below you can see the model of documents for the users collection:

```
const Schema = mongoose.Schema;

const userSchema = new Schema({
 company: { type: String, required: false },
 email: { type: String, required: true },
 createdAt: {
   type: Date,
   default: Date.now,
 },
 googleId: { type: String, required: true },
 codingTests: {type: Array, required: false}
});

const NewUserDB = mongoose.model('NewUserDB', userSchema);
```

Above we can see that when creating a new document for a user a company, email, createdAt and google id are all required to successfully create the document, values for the coding tests array are not required as on creation obviously the user would not have set up any tests yet. Values for the coding tests are pushed onto the array later when the user creates a coding test and the id from the coding test document is pushed onto the coding tests arrays. In total we have 5 models for the 5 collections in the cluster(Users, Coding tests, coding challenges, questions, participants).

4.3 User Interface

The frontend of the application was built using react. We used material ui to design the ui. Material provided us with very useful pre styled components as well as the tools to customize these components. For example the Grid component gave us the ability to create the layout for a section of the DOM, we used the grid component when designing the layout for the coding test UI, the grid consists of the header, problem description, code editor and terminal. Material ui provides good documentation which assisted us through the design process. The material UI design consistency enabled us to make the ui simple and easily accessible for non technical users.

4.4 Coding Test

The employer sets a coding challenge and sends an email to a candidate with a unique link allowing them to access the coding test. Upon accessing the link, a call to the backend is initiated and the ids in the link are verified to ensure the user is authorized to access the test. Upon passing that check, another check is conducted to ensure the user cannot reattempt a coding test, there is one submission per link.

Once a candidate is authorized they can proceed to attempt the coding challenge. As the user types code it is saved to local storage in case the user accidentally refreshes or closes

the browser. Upon a second visit to the link any code that is saved to localstorage is restored, ensuring no code loss during the test.

When a User hits the run code button, a container is started by the remote code execution API on a google cloud vm. The code is saved in the container as a file and it is executed, returning stdout and stderr. Metrics such as time execution and memory usage are also obtained. This is then returned to the user and the results of each test case and output is displayed to the user.

4.5 API Documentation And Testing

We used Postman to document and test our API manually throughout the lifecycle of the project. API documentation can be found in the 'docs' folder of the repository or it is available online at https://documenter.getpostman.com/view/10756408/TzK14tzh.

5. Sample Code

5.1 Gitlab CI/CD

The pipeline in Gitlab is defined in a file called .gitlab-ci.yml. Here we enter commands to specify commands to execute at various stages within the pipeline. Below you can see the image and various stages we have in our pipeline:

1. **Image**: Here we define the docker image to execute the commands within
2. **Stages**:
    a. **Build**: Here the code is built to ensure it can be compiled without error
    b. **Test**: Here the unit and integration tests are executed.
    c. **Deploy**: Here the code is deployed to a production environment.
    d. **E2e**: Here the e2e tests are executed against the production environment.

```
image: node:lts

stages:
  - build
  - test
  - deploy
  - e2e
```

Below you can see our backend integration test job.

- **Services**: Here you can see we use mongoDB as a service which pulls the official container from docker and makes the mongo container accessible in our pipeline.
- **Cache**: Here we use job specific cache and only cache the node modules so as to avoid redownloading them upon every job trigger. We also define a key so it can be reused by other jobs that require the same cache so as to avoid duplication.

- **Only**: Here the job is only triggered if there are changes made within the src/functions/ directory, without this the job would run unnecessarily on every commit.
- **Retry**: The job is retried a maximum of 2 times in the instance of a runner failure or timeout/stuck error. This avoids manual rerunning of the pipeline for errors that are specific to gitlab or can be likely be solved by rerunning.

```
test-backend-integration:
  stage: test
  services:
    - mongo
  cache:
    key: backend-cache
    paths:
      - ./src/functions/node_modules/
  script:
    - cd src/functions
    - npm i
    - npm run test:integration
  only:
    changes:
      - src/functions/**/*
  retry:
    max: 2
    when:
      - runner_system_failure
      - stuck_or_timeout_failure
```

5.2 Remote Code Execution with Docker

Below you can see the main service that executes the users inputted code in a docker container and deletes it afterwards. To obtain the stdout and stderr from the container we attach our own stdout and stderr to listen and capture any logs outputted. We also set a memory limit of 200mb to ensure no container uses an unnecessarily large amount of memory. Time usage and memory usage are also obtained from the process executed in the docker container. Afterwards, the container is deleted and the outputs and statistics are returned.

```
// executes code submitted in docker container, deleting container afterwards.
class ExecutorService {
  async execute(code, input, language, maxTimeLimit) {
    var stdout = new streams.WritableStream();
    var stderr = new streams.WritableStream();

    const context = this.createContext(code, input, language, maxTimeLimit);
```

```
    const data = await docker.run(
      context.image,
      ['/bin/sh', '-c', context.cmd],
      [stdout, stderr],
      { Tty: false, memory: '200m' }
    );
    const container = data[1];

    const stats = await container.inspect();
    const runTime = getRunTime(stats.State.StartedAt, stats.State.FinishedAt);
    await container.remove();
    stdout = stdout.toString().trim();
    stderr = stderr.toString().trim();
    const output = extractMemory(stderr);

    if (this.isTimeoutError(stderr)) {
      output.stderr = 'Timeout Error: Maximum time limit exceeded.';
    }

    return {
      time: runTime,
      memory: output.memory,
      stdout: Base64.encode(stdout),
      stderr: Base64.encode(output.stderr),
    };
  }
}
```

Below you can see the function which determines which docker image and command to execute within the container.

As you can see we first decode the code and input as they are encoded in Base64, this is to ensure that a larger character set is available to avoid issues with unsupported characters when they are sent over the web in a JSON format.

Next we ensure the given timeout is above 15, as below 15 is a limit put in place to ensure code is given enough time to execute within the container. Timeouts are handled by using the ‘timeout’ keyword in linux within the container, this command will terminate the process, once execution time is over the given timeout.

If you look closer at the ‘context.cmd’ lines below you will see the full command that is executed within the container. Here you can see that the code is echoed into a file and then run using the standard command for a given programming language e.g for python the command is ‘python3 test.py <args>’, where args is the input passed in as command line arguments. If an unsupported language is given an error is thrown and returned.

```
// returns correct docker image name and command to execute
  createContext(code, input, language, maxTimeLimit) {
    code = Base64.decode(code);
    input = Base64.decode(input);
    code = escapeQuotes(code);
    const getMem = "time -f 'MEM: %M'";
    if (!maxTimeLimit) {
      maxTimeLimit = 15;
    } else {
      maxTimeLimit = parseInt(maxTimeLimit);
      maxTimeLimit < 15 ? 15 : maxTimeLimit;
    }
    const timeout = `timeout ${maxTimeLimit}`;

    var context = {};
    language = language.toLowerCase();
    switch (language) {
      case 'python':
        context.image = 'python:3-alpine';
        context.cmd = `echo "${code}" > test.py && ${getMem} ${timeout}
python3 test.py ${input}`;
        break;
      case 'java':
        context.image = 'openjdk:8-alpine';
        context.cmd = `echo "${code}" > Main.java && javac Main.java &&
${getMem} ${timeout} java Main ${input}`;
        break;
      default:
        throw new Error(`'${language}' is not a supported language.`);
    }
    return context;
  }
```

5.3 Remote Code Execution Producer Consumer with Redis

The remote code execution API incorporates a concurrent producer consumer pattern in which incoming requests to the server add valid submission to a redis queue, this redis queue is pulled from by a pool of workers in a first in first out order (FIFO).

Below is the code which handles a submission, There are two ways in which this endpoint can be used:

1. You can skip the redis queue by adding the ‘wait:true’ key value pair to the JSON, this if faster than waiting for the worker to pull the job from the queue but it limits the concurrency of the system as under heavy user load multiple errors will be thrown due to memory limits being exceeded or over utilisation of the cpu.
2. You can use the redis queue to handle heavy user load which is slightly slower as you have to wait for a worker to be ready and pull the job from the queue but it ensures maximum concurrency of the system.

```
// add submission to queue
exports.addSubmission = async (req, res) => {
  const data = req.body.data;
  if (data?.wait === true) {
    const code = data.code;
    const input = data.input;
    const language = data.language;
    const maxTimeLimit = data.timeout;

    const output = await new ExecutorService().execute(
      code,
      input,
      language,
      maxTimeLimit
    );
    res.status(200).json({ data: output });
  } else {
    const job = await submissionQueue.add(data);
    res.status(201).json({ id: job.id });
  }
};
```

Below you can see the code for the worker where you can define the number of workers using environment variables. Jobs are distributed amongst the pool of workers, which pull from the redis queue in a FIFO manner and execute the same ‘ExecutorService().execute()’ as explained above. Results are then saved back to redis and available to be obtained through another endpoint.

```
const workers = process.env.WEB_CONCURRENCY || 2;
```

```
const maxJobsPerWorker = process.env.MAX_JOBS_PER_WORKER || 50;

const start = async () => {
  submissionQueue.process(maxJobsPerWorker, async (job) => {
    try {
      const code = job.data.code;
      const input = job.data.input;
      const language = job.data.language;
      const maxTimeLimit = job.data.timeout;

      const output = await new ExecutorService().execute(
        code,
        input,
        language,
        maxTimeLimit
      );
      await job.update(output);
      return { data: output };
    } catch (err) {
      console.error(err);
      return { data: err };
    }
  });
};

// Initialize workers
throng({ workers, start });
```

5.4 Email

Below you can see the code that sends an email via a simple write to a database, this write triggers a firebase function which sends an email via the SMTP service provider SendGrid to the intended recipient. Details of why we took this approach are explained in the ‘Problems Solved’ section below.

```
exports.sendEmail = async (req, res) => {
  const email = req.body.data.email;
  const TestId = req.body.data._id;
  const googleId = req.body.data.googleId;
  const attemptedTest = req.body.data.attemptedTest;

  const newParticipantsEntry = new ParticipantsDB({
    email,
```

```
    TestId,
    googleId,
    attemptedTest,
  });

  const result = await newParticipantsEntry.save();
  const participantsId = result._id;

  await CodingTestDB.updateOne(
    { _id: TestId },
    { $push: { participants: participantsId } }
  );

  await db.collection('mail').add({
    to: email,
    message: {
      subject: 'Coding Test Invitation',
      text:
        'You have been invited to attempt a coding test, you can access the
test by clicking the following link: \n
https://coding-test-platform.web.app/codingtest/' +
        TestId +
        '/' +
        participantsId,
    },
  });

  return res.status(200).json({
    data: null,
  });
};
```

5.5 Delete Users

In the user interface there is an option for a user to delete their account, when the user confirms that they wish to delete their account, it sends a request to the delete account endpoint of the API and also calls the firebase delete user function which removes the user from firebase authentication.

```
const handleSubmitDelete = async (e) => {
   try {
     e.preventDefault();
     deleteUser();
```

```
    await DeleteUserDetails();
    history.push('/login');
  } catch {
    console.log('error');
  }
};
```

As the googleID is the foreign key in all collections, we only need one query to delete many documents from all collections in the Database as shown in the code snippet below. This means that a user can delete all data related to them when they choose to delete their account.

```
exports.deleteUserData = async (req, res) => {
 const user = {
   googleId: req.body.data.googleId,
 };

 var query = {
   googleId: user.googleId,
 };

 await NewUserDB.deleteOne(query);

 await CodingTestDB.deleteMany(query);

 await CodingChallengeDB.deleteMany(query);

 await QuestionsDB.deleteMany(query);

 await ParticipantDB.deleteMany(query);

 return res.status(200).json({
   data: null,
 });
};
```

6. Problems Solved

6.1 Remote Code Execution

Problem

We needed to remotely execute code on a server which is inputted by a user. This is inherently dangerous as the code can be malicious, which could potentially allow a user to gain access to and control the server, it can have infinite loops and high memory usage which would drain server resources and cost a lot of money.

Solution

Our solution was to use Docker. This allowed us to set memory and timeout limits which prevents high memory usage and infinite loops from occurring. It also allowed us to isolate a user's code in a container which prevents malicious code from getting out of the container and causing harm to the system. Each time a user hits the “Run Code” button a container is spun up using the official container for the language e.g. for Java we use the official published openJDK docker image publically available on docker hub. The users code is inserted into the container as a file and the program is executed within the container. Program output is read from the container by reading stdout and stderr. Upon completion the docker container is deleted afterwards.

6.2 Remote Code Execution API Minimal Resources

Problem

The remote code execution API is hosted on google cloud platform on a small instance with ~600mb of ram and 1 cpu core. This posed an issue as for example if 1000 people were to use our application at the same time and hit the “Run code” button simultaneously the RCE server would be overloaded and run out of ram, thus causing multiple errors.

Solution

Our solution to this was to use a concurrent producer consumer pattern using a FIFO redis queue to queue the workload amongst a pool of workers. This allows the server to run effectively even under heavy load as submissions are queued and only executed once the pool of workers are finished their current task. This limits the amount of cpu and ram used at any given time. The workers can easily be scaled to support higher concurrency by simply changing an environment variable to define the number of workers in the pool. We also added swap to the virtual machine to give the machine some more flexibility, even if using that extra memory would be slow.

6.3 Remote Code Execution Non Encrypted Traffic

Problem

The remote code execution API originally served as a standard http server. This posed a problem when our react app tried to communicate with the API in production as Firebase would throw errors in the console stating that there was mixed content. All the microservices and hosting were secured using https but the RCE API was served using http.

Solution

The solution was to generate a self signed certificate on the RCE API which allowed the server to send traffic safely between the server and clients without the possibility of the messages being intercepted by outside parties. The certificate system also allows clients to verify the identity of the sites that they are connecting with. For encryption we generated a private key using the Rivest–Shamir–Adleman algorithm (RSA) that is 2048 bits long. This key was configured to be used by Nginx, and all http traffic was redirected to https.

6.4 Remote Code Execution Non Encrypted Traffic Part 2

Problem

Previously we had configured our server to use https via a self signed certificate, which worked at the time, but in the last week of the project Firebase seemed to have updated their security rules and blocked all outgoing traffic to https sites that are self signed.

Solution

The solution was to acquire a domain and create an official ssl certificate, this involved dealing with nameservers and dns servers to point the registered domain to the IP address of the google cloud virtual machine and configuring nginx to handle the ssl certificate and encrypt all traffic.

6.5 Firebase Redirects

Problem

Firebase had a built-in functionality in which you would need to define where to send a particular request using regex patterns, e.g to the backend or frontend. This caused many problems as the redirects did not seem to function properly after several variations, unless for every new request or API endpoint you defined the route in the redirects file. This was obviously not a good solution as errors would occur if individuals forgot to define all new endpoints and router, also there is significant time wastage in doing so.

Solution

The solution was to not use the firebase package which would handle sending the request to the server by invoking the redirects file in the background. Instead we used standard http

requests to communicate with the server thereby skipping the unnecessary redirection built into firebase. This solution made the code much more flexible by reducing the coupling of the application with the Firebase platform, which is important for maintainability of the application by reducing unnecessary dependencies and improving overall flexibility and allows us to quickly switch platforms if needed.

6.6 Cross Origin Errors

Problem

Upon first developing our react application and then proceeding to attempt to communicate with our backend AP, the console logged numerous errors each relating to cross origin requests.

Solution

It was preventing a call to an API endpoint from a different machine, as it knew nothing about it. This is favourable functionality as it guards against a cross-site scripting attack. The solution was to simply add Cross-Origin Resource Sharing (CORS) headers to the requests which allowed our API and frontend application to communicate from different origins.

6.7 Email Errors In Production

Problem

Originally we used a library called nodemailer to send emails via a gmail account. This worked locally in a development environment but upon deployment to a production environment in a different location, errors occurred with gmail detecting the emails as spam or rejecting them from even being delivered into spam. This caused email to be lost and errors to be thrown on the client side.

Solution

The solution was to use an SMTP service provider which is updated and maintained to meet the standards necessary for gmail to not mark the email as spam. For this we used the SendGrid SMTP service provider coupled with a firebase function which would be triggered upon writing an “email” in json format to a specified collection in a database. This function trigger would then send the email using the SMTP service provider and allow us to successfully deliver email globally.

6.8 JEST for components wrapped in firebase authentication

Problem

For the private routes in our coding test platform, the components were all wrapped in the firebase authentication context. However when it came to testing these components using JEST. JEST was unable to render these components. The error was always that it could not destructure the property of the firebase auth functions as it is defined. I.e. “TypeError: Cannot destructure property 'currentUser' of '((cov_5mwatn2cf(...).s[0]++) , (0 , _AuthContext.useAuth)(...))' as it is undefined.” This problem made it impossible to test any of these components and elements in them.

Solution

The solution to this problem was to define a custom render method that included authentication context providers by creating a utility file(test-utils.js) that re-exports everything from the react testing library. Instead of creating a new auth context this file had to still use the AuthContext from AuthContext.jst for <AuthContext.Provider>, as that's the context that the hook uses. We were then able to replace all our react testing library imports in our test files with imports from our test.utils.js file.

6.9 Gitlab Pipeline timeouts

Problem

Originally all cache was defined in the .gitlab-ci.yaml file as global cache. This worked fine in the beginning as there were only a few stages but as the project progressed more and more stages were added. This meant for every job that pulled cache, it would pull all the cache even though it only required a single folder in the cache. This was the main issue as most of the time in the pipeline was spent downloading and uploading cache, which would then be hit with a timeout error if exceeding 20 minutes.

Solution

The solution was to use job specific cache and keys so that each job only downloads and uploads the cache they need and the key allows jobs that need the same cache to be reused within that job, avoiding duplication.

7. Results

We successfully completed all the functional requirements we originally defined in our functional specification as seen below, thus this project can be deemed a success.

7.1 Gantt Chart

	Start Date	End Date	Timeline	Status
Final Year Project	9-11-2020	28-05-2021		Active
Get project approved by approval panel	9-11-2020	09-11-2020		Complete
Functional Specification	10-11-2020	05-12-2020		Complete
Setup Databse, Firebase and CI/CD	7-12-2020	20-12-2020		Complete
MongoDB NoSQL Data Modelling	14-12-2020	20-12-2020		Complete
Setup Jest Testing (unit, integration, e2e)	14-12-2020	20-12-2020		Complete
Build Register/Login/Logout Authentication	11-1-2021	17-01-2021		Complete
Setup Simple Code Execution with IDE	11-1-2021	31-1-2021		Complete
Develop Employer Coding Test Setup UI	18-1-2021	28-2-2021		Complete
Email Candidates Link To Coding Test	1-3-2021	7-3-2021		Complete
Fully Integrate Code Execution system with IDE and sync code incase of disconnect	8-3-2021	4-4-2021		Complete
Analytics - Evaluate Candidate Solutions	5-4-2021	11-4-2021		Complete
Candidate Video Responses - Save to Firebase	12-4-2021	18-4-2021		Complete
Refactoring and Finish Code	19-4-2021	7-5-2021		Complete
Final Documentation and video walkthrough	19-4-2021	7-5-2021		Active
Submission	7-5-2021	07-05-2021		Upcoming
Project Expo	17-5-2021	28-5-2021		Upcoming
Project Demonstration	17-5-2021	28-5-2021		Upcoming
		Overall Progres		

7.2 Functional Requirements:

7.2.1 Online Compiling of code

Requirement

The code needs to be compiled/interpreted upon submission and when the participant runs their code in order for them to see the results of the test cases. Code compilation is the most critical component of the system as it's output is used to evaluate the code against test cases for candidate evaluation

Result

For this we used Docker. This allowed us to set memory and timeout limits when running the candidate's code against the test cases which prevents high memory usage and infinite loops from occurring. It also allowed us to isolate a user's code in a container which prevents malicious code from getting out of the container and causing harm to the system.

7.2.2 Code Editor

Requirement

The System must allow a user to write code online as they would offline in an IDE with syntax highlighting.

Result

The editor in the application works just as you expect an offline text editor to work with a syntax highlighter for multiple languages and line numbers to assist users for debugging their code.

7.2.3 Register and Login

Requirement

The user will have to register first and will then be securely logged in using authentication. Only the employer is required to login as they are required to set up their custom coding test.

An employer needs to be able to register/login in order to set up the coding test, sending emails to applicants and to view results.

Result

When registering users have the option to register with an email and password or to make things easier, they also have the option to sign up with google. Only employers are required to register as they will be creating the coding tests and viewing the results. When the user logs in with either email and password or signs in with google, they are authenticated through firebase authentication.

7.2.4 Database

Requirement

Saving employer details, coding tests, interview questions, participant information, participant code and results, video responses is an important requirement.

Result

The MongDB collections are modelled so they contain all the information required for the user, coding tests, challenges, questions and the coding tests participants. And as explained in detail in the design, the googleId of the user is used as the foreign key to link documents together.

7.2.5 Generating Link And Sending Email

Requirement

The application should be able to generate a custom link for each selected applicant and send it to them via email. This is critical because if the link does not work and/or is not sent to the applicant, they cannot attempt the coding test.

Result

When an employer enters an participant email to send an invitation to. The backend api creates a new document in the MongoDB participant collection and uses the object id of that document together with the object id of the coding test document to create the link to send in an email to the participant. By using the MongoDB ids, it ensures that all links to a coding test are unique and no two participants will have the same link to a coding test.

7.2.6 Coding Test Setup

Requirement

The employer must be able to set up a coding test with multiple coding challenges along with test cases to allow the system to evaluate candidates.

Result

The user interface makes it easy for the user to create coding tests, for setting up coding challenges they fill out a simple form (there is an example provided to assist them). The user also has the option to edit challenges if they wish i.e. at more test cases. There is an option to delete individual challenges or the entire test. There is also a feature to reset a test, this means all current results for that test are removed and all current invitations are made invalid.

7.2.7 Video Responses

Requirement

Candidates should be able to record a video response to questions proposed by the employer and these videos should then be uploaded to storage.

Result

This was successfully implemented, candidates click a button to start recording and they can click a button to stop recording or else wait for the 60 second timer to run out. When the video is being uploaded to firebase storage there is a loading screen that ensures the window is not closed before the video is finished being uploaded to firebase storage. The employer is then able to watch the recordings in the results page of the UI.

7.2.8 Analytics

Requirement

The system must be able to assess an applicants' code based on predefined test cases set by the employer and metrics such as cpu time and memory used so as to allow the employer rank of candidates.

Result

Once a candidate has completed and submitted their coding test. The employer can view the results in the results section. The results show the candidates code solution, the number of test cases passed and the runtime and memory used for each test case. This helps the employer to determine which candidate had the most efficient solution.

7.2.9 User Interface

Requirement

The UI should be designed with simplicity and accessibility in mind so as to allow non technical users to operate the employer section and allow various individuals with disabilities to operate the site easily.

Result

When developing the UI we designed it with simplicity and accessibility in mind so as to allow non technical users to operate the employer section, we achieved this by having

things helper text when filling out the coding challenge form and examples. Material UI Components were built with WCAG 2.0 Level AA in mind.

7.3 What we learned

In terms of what we learned, we both gained a vast amount of knowledge of technologies we did not have much experience with before undertaking this project.
These technologies include:

- Docker
- MongoDB
- React
- Node.js
- Express js
- Firebase
- Google vm

8. Future Work

8.1 Remote Code Execution With MicroVMs

While the implementation of remote code execution works, due to Docker it has a high cpu and memory usage. A better solution would be to utilise amazons open source software called firecracker which they use in amazon web services like AWS Lambda. The Micro VMs have significantly greater startup time and much less memory footprint while also being considerably more secure and isolated.

8.2 Expand application to encompass entire job application process

It is possible that this application could be expanded to encompass the entire job application process. Starting with enabling employers to post jobs on the platform and allowing applicants to apply directly through the application. Then along with the coding test and video interview components we would also have a live video chat feature for employers to conduct interviews remotely. We could also allow job seekers to create accounts so when they apply for jobs they can monitor their application status when they apply for various jobs.

8.3 Live Coding Test

During the application process for a software engineer role there tends to typically be a live coding test in which an interviewer gives you a problem and you have to solve it on the spot through code. This could be achieved by building off the existing coding test system and adding a new option on coding test setup to make the challenge a live coding test. The interviewer and the candidate need to be able to see the same screen and see the code being written by the candidate in real time. This can be achieved by using websockets to stream the code from the candidate computer to the interviewers computer and vice versa.

8.4 Remote Code Execution Audit

While the use of docker should isolate the user code and make it much harder for malicious code to escape the container, there is still a possibility that a user could escape the container by nefarious means. To prevent this and stop a process that escapes a container even after being deleted we can monitor the process ids and ensure no bad actors are in execution and kill any processes that do not pass the audit.

8.5 Saas Company

We could make this product an actual business as a software as a service offering by which companies can sign up and pay a monthly subscription fee to use our services. Some additional work to achieve commercial readiness would be to dockerize the remote code execution API and use Kubernetes to distribute the load between containers to improve the scalability and concurrency of the API, also we would need to improve the frontend UI to be of a standard similar to professional businesses.

9. Appendices

1. (WAI), W3C Web Accessibility Initiative. “WCAG 2.1 at a Glance.” Web Accessibility Initiative (WAI), www.w3.org/WAI/standards-guidelines/wcag/glance/. Accessed 25th Nov. 2020.
2. “React – A JavaScript Library for Building User Interfaces.” React, reactjs.org. Accessed 4 Dec. 2020.
3. “Express - Node.Js Web Application Framework.” Express, expressjs.com. Accessed 4 Dec. 2020.
4. MongoDB. “The Most Popular Database for Modern Apps.” MongoDB, www.mongodb.com. Accessed 4 Dec. 2020.
5. Firebase, Google, firebase.google.com/. Accessed 4 Dec. 2020.
6. “Managed MongoDB Hosting | Database-as-a-Service.” MongoDB, www.mongodb.com/cloud/atlas. Accessed 4 Dec. 2020.
7. “Judge0.” GitHub, github.com/judge0/judge0. Accessed 4 Dec. 2020.
8. “Jest · Delightful JavaScript Testing.” Jest, jestjs.io. Accessed 4 Dec. 2020.
9. Docker - Docker makes development efficient and predictable, https://www.docker.com. Accessed May 1 2021.
10. Cypress, https://docs.cypress.io/guides/overview/why-cypress#In-a-nutshell. Accessed May 1 2020.
11. Google Cloud, https://cloud.google.com/. Accessed May 1 2021.
12. Nginx, https://www.nginx.com/. Accessed May 1 2021.
13. RCE, https://en.wikipedia.org/wiki/Arbitrary_code_execution. Accessed May 5th 2021.
14. Redis, https://redis.io/, Accessed May 5th 2021.
15. SendGrid, https://sendgrid.com/, Accessed May 5th 2021.
16. Firecracker, https://firecracker-microvm.github.io/, Accessed May 5th 2021.
17. Kubernetes, https://kubernetes.io/, Accessed May 5th 2021.
18. Postman, https://www.postman.com/, Accessed May 6th 2021.