SENG3011 21T1

API Design Report

Original Deliverable 1 + Deliverable 2 Updates

Group 200OK

Abhyudit Gupta	z5196145
Jiahui Luo (Lacey)	z5158415
Haoran Xu (Matthew)	z5134675
Yueru Duan (Ellen)	z5210986
Ayaan Adil	z5213315

Table of Content

Deliverable 1 API DESIGN/DESIGN DETAILS

Development of API module and Web App

Objective :
The objective of the project is to extract information from the CDC (Centre for Disease Control) Website using a web scraper and then make it available on the web in an organised manner using an API. Once we have constructed the API we have to make a Web App that will use this API and other similar API's to present detailed information to users about disease outbreaks depending on the parameters they specify.

Software Architecture:

Figure 1 illustrates the overall functioning of our web application.

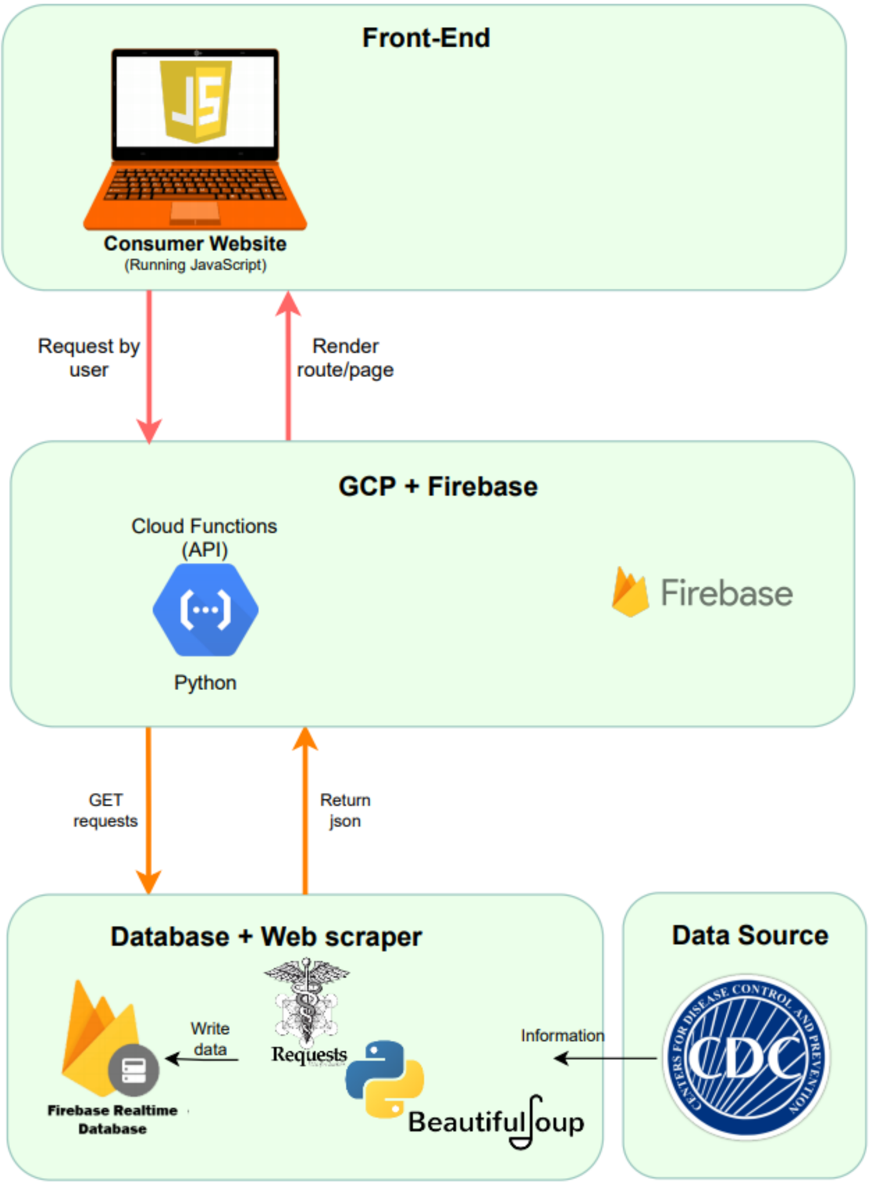


Figure 1: Overall Architecture

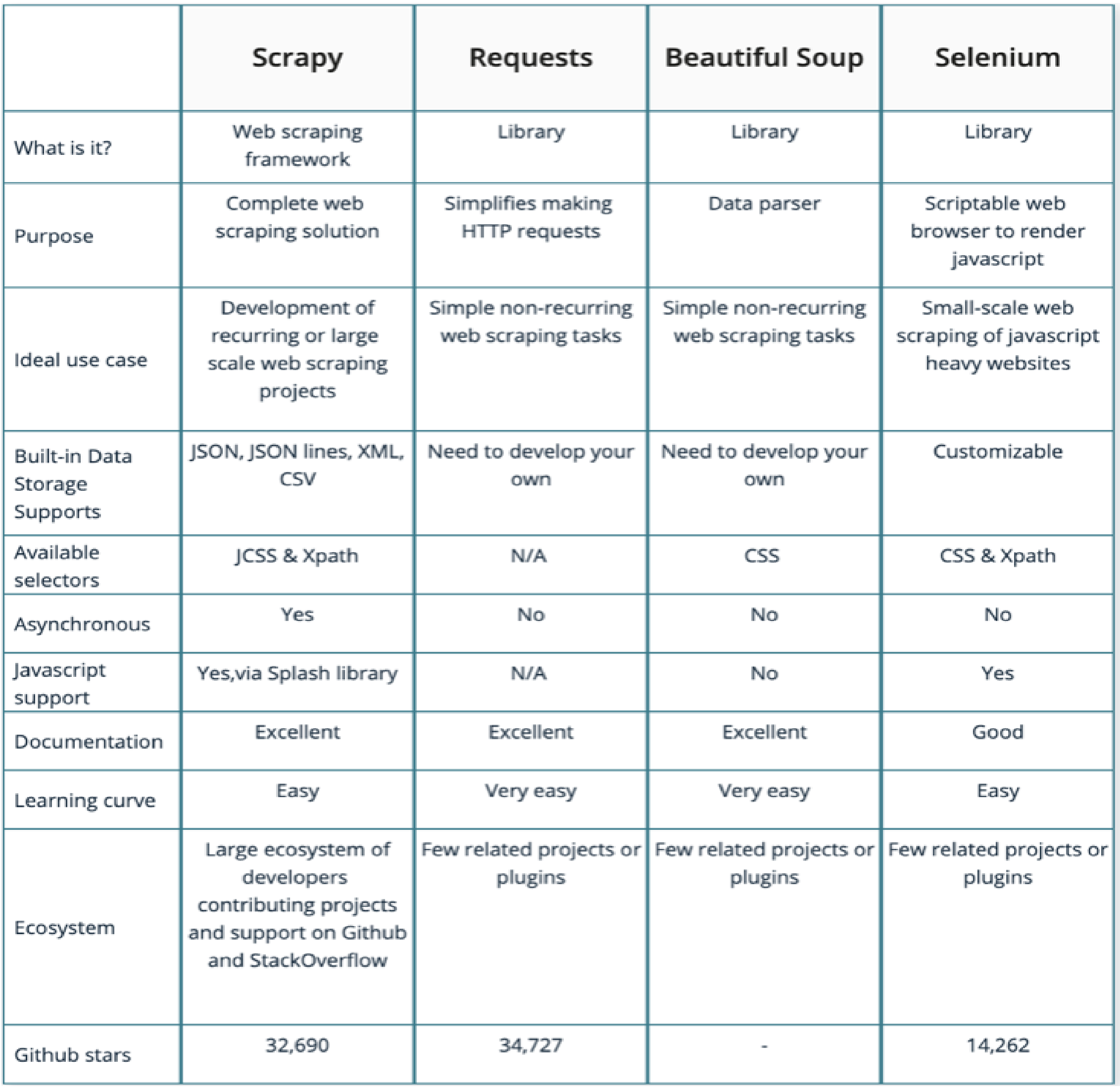
Therefore, the main components of the project include :

1) Web scraping
2) Development of API
3) Construction of Web App

Detailed information about each of the design choices mentioned below will be elaborated on in later parts of the report

Web Scraping:
The web scraper will extract information from the CDC website on a regular basis. We will be using a combination of BeautifulSoup and the Requests library to conduct our scraping. The web scraper will be hosted as a separate cloud function. For each run the scraper will create

disease reports and add them to our database. Table 1 shows the comparison between commonly-used web scrapers.



	Scrapy	Requests	Beautiful Soup	Selenium
What is it?	Web scraping framework	Library	Library	Library
Purpose	Complete web scraping solution	Simplifies making HTTP requests	Data parser	Scriptable web browser to render javascript
Ideal use case	Development of recurring or large scale web scraping projects	Simple non-recurring web scraping tasks	Simple non-recurring web scraping tasks	Small-scale web scraping of javascript heavy websites
Built-in Data Storage Supports	JSON, JSON lines, XML, CSV	Need to develop your own	Need to develop your own	Customizable
Available selectors	JCSS & Xpath	N/A	CSS	CSS & Xpath
Asynchronous	Yes	No	No	No
Javascript support	Yes,via Splash library	N/A	No	Yes
Documentation	Excellent	Excellent	Excellent	Good
Learning curve	Easy	Very easy	Very easy	Easy
Ecosystem	Large ecosystem of developers contributing projects and support on Github and StackOverflow	Few related projects or plugins	Few related projects or plugins	Few related projects or plugins
Github stars	32,690	34,727	-	14,262

Table 1: Comparison of various web scrapers

Having compared and studied the various web scrapers offered in Python. We will use **BeautifulSoup and Requests** because of the following reasons:

1) The pages that we have to scrape do not have repetitive content. The content is not placed in the same tags on different pages. Hence eliminating the need to use Scrapy.

2) BeautifulSoup has selectors that are much easier to use as compared to other scrapers.

3) The CDC website has preloaded content and hence we do not need a website that handles JavaScript. This eliminates the need to use Selenium.

4) In the end, BeautifulSoup is very easy to use and the team feels comfortable in experimenting with it.

Development of API:

The development of the API involves 3 important design decisions -

1) Hosting the API
2) Storing the information
3) Documentation of the API

Hosting the API:

Our API will be hosted using Google Cloud Functions. Hence, given the function's URL, anyone on the web can query it using any web browser.

Storing the data:

The cloud function will make calls to a firebase database (which is a real time database provided by Google) and the database will return the disease reports in the required JSON format.

The Firebase database will be populated by the Python web scraper.

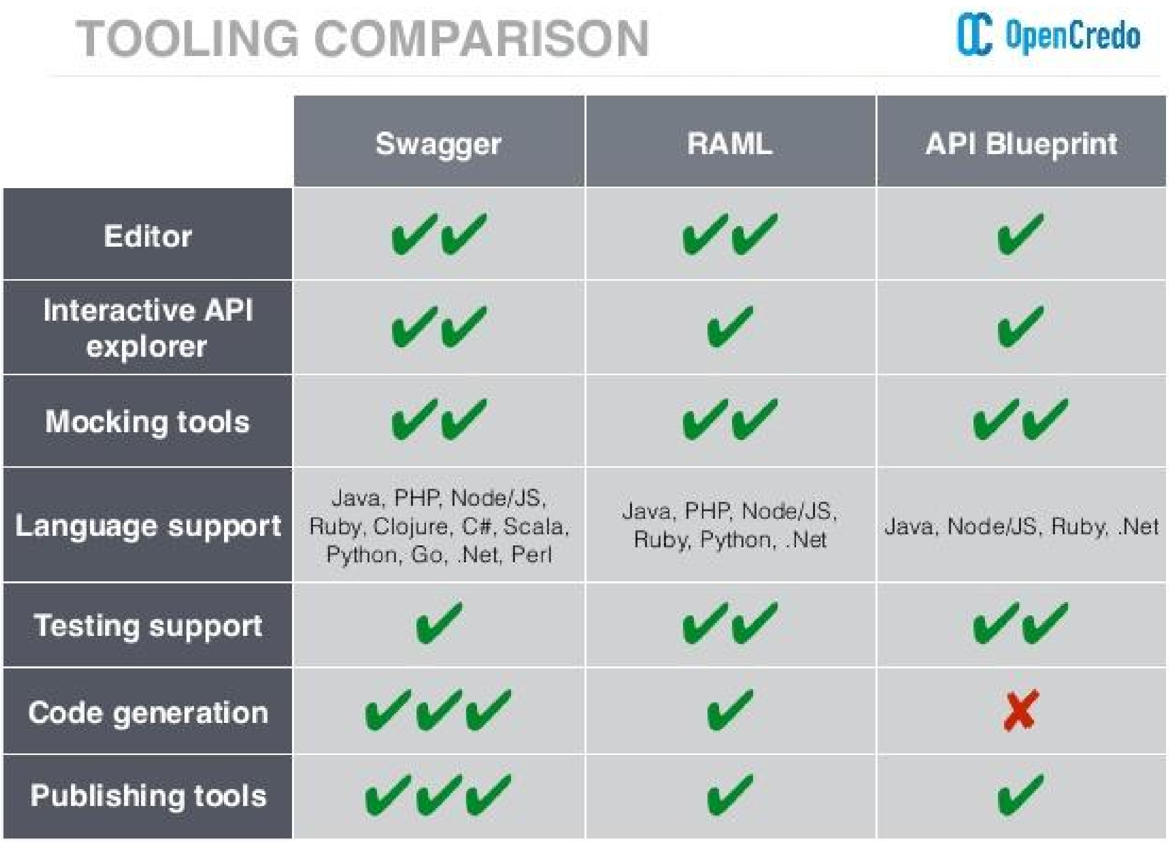
Documentation:

The documentation for the API would be made using Swagger. Any user who wants to use the API can use the swagger documentation to view the various calls that they can make.

Construction of Web App:

We will be using ReactJS for the frontend of our project. The backend will be made with Python. We will be using GCP to host the website. The website will use a combination of 2-3 APIs to present disease reports. The user will enter the required parameters and the website will render a page that will show various graphs and charts displaying details of the outbreak the user searched for.

API Documentations:



TOOLING COMPARISON

OpenCredo

	Swagger	RAML	API Blueprint
Editor	✓✓	✓✓	✓
Interactive API explorer	✓✓	✓	✓
Mocking tools	✓✓	✓✓	✓✓
Language support	Java, PHP, Node/JS, Ruby, Clojure, C#, Scala, Python, Go, .Net, Perl	Java, PHP, Node/JS, Ruby, Python, .Net	Java, Node/JS, Ruby, .Net
Testing support	✓	✓✓	✓✓
Code generation	✓✓✓	✓	✗
Publishing tools	✓✓✓	✓	✓

Table 2: Comparison of API documentation tools

Table 2 shows a comparison of various API documentation platforms. We will use Swagger to document our API because of the following reasons -

1) Easy to use
2) Recommended by the university
3) Has a beautiful UI and makes documentation very straight forward

Parameters

Passing Parameters and making calls to the API

There are 3 major ways of passing parameters to any API -

1. *As part of the **URL-path** (i.e. /api/resource/**parametervalue**)*
2. *As a **query argument** (i.e. /api/resource**? parameter=value**)*
3. *Or as **JSON** objects as part of url (i.e. /api/resource?)*

We will be using the query argument method to pass parameters because of two reasons -

1) Google Cloud Functions do not support passing parameters as part of the URL path.
2) Industry best practises indicate that query arguments are often used for specifying search terms.

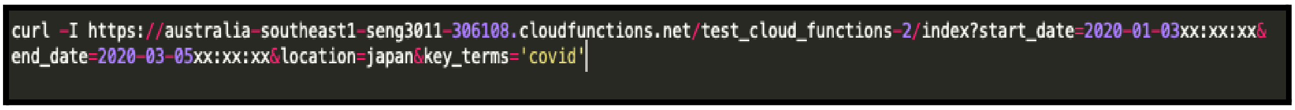
The main parameters to be passed in are -

1) start_date
2) end_date : The dates will be of the format “yyyy-MM-ddTHH:mm:ss”
3) location : example “Japan”
4) keywords : example “COVID,ZIKA”

The API will not work if any of these parameters are empty or None.

Sample Calls to API/Cloud Function

Curl sample GET request

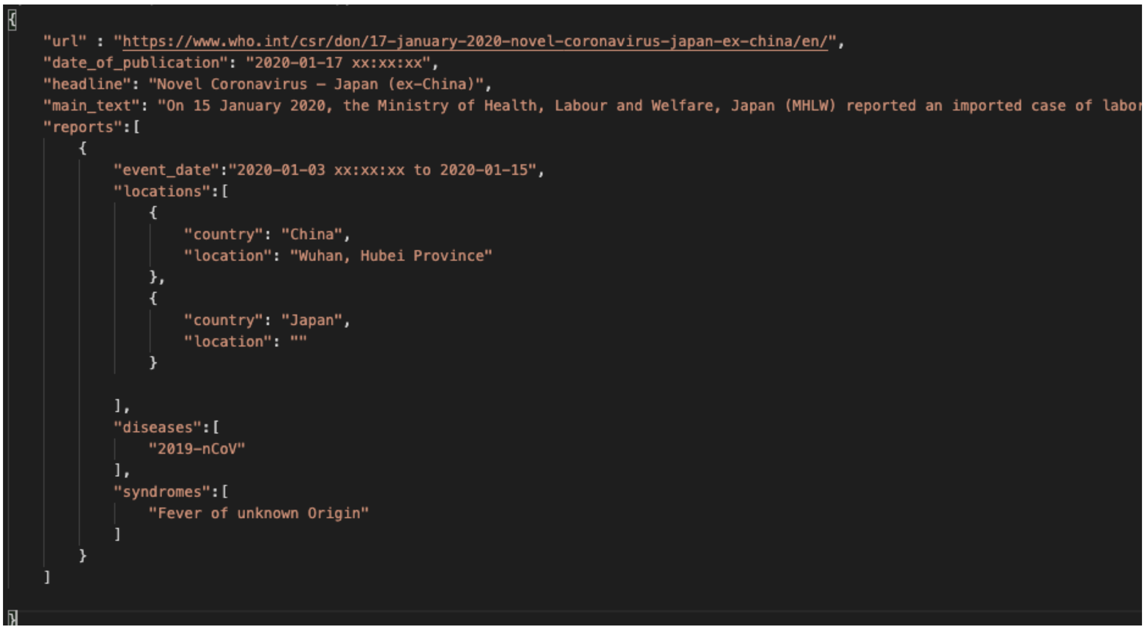


```
curl -I https://australia-southeast1-seng3011-306108.cloudfunctions.net/test_cloud_functions-2/index?start_date=2020-01-03xx:xx:xx&
end_date=2020-03-05xx:xx:xx&location=japan&key_terms='covid'
```

Figure 2: Sample GET request

Corresponding 200 OK Response

Figure 3 shows the sample disease report returned in JSON format.

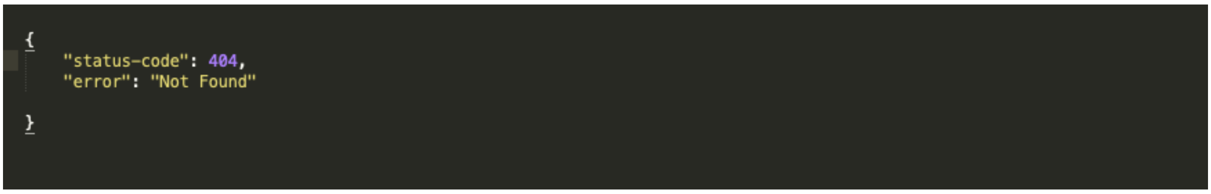


```
{
    "url" : "https://www.who.int/csr/don/17-january-2020-novel-coronavirus-japan-ex-china/en/",
    "date_of_publication": "2020-01-17 xx:xx:xx",
    "headline": "Novel Coronavirus – Japan (ex-China)",
    "main_text": "On 15 January 2020, the Ministry of Health, Labour and Welfare, Japan (MHLW) reported an imported case of labor
    "reports":[
        {
            "event_date":"2020-01-03 xx:xx:xx to 2020-01-15",
            "locations":[
                {
                    "country": "China",
                    "location": "Wuhan, Hubei Province"
                },
                {
                    "country": "Japan",
                    "location": ""
                }

            ],
            "diseases":[
                "2019-nCoV"
            ],
            "syndromes":[
                "Fever of unknown Origin"
            ]
        }
    ]
}
```

Figure 3: Sample 200 OK response

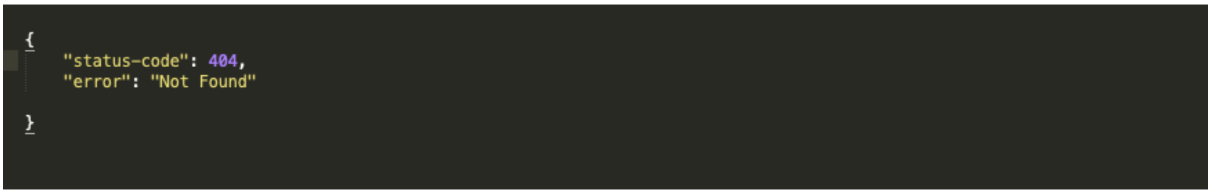
An example of a 404 error :



```
{
    "status-code": 404,
    "error": "Not Found"
}
```

Figure 4: Sample 404 response

An example of a 400 error: It means that the server was not able to process the request.



```
{
    "status-code": 404,
    "error": "Not Found"
}
```

Figure 5: Sample 400 response

Justification for Technological Stack

Architecture

Architecture is the most fundamental and critical aspect to consider for this project since it determines how the web application is hosted online and which programming languages and frameworks to use. In the following subsections, traditional server architecture is compared with serverless architecture and their benefits are shown in Table 3, and in Table 4, different serverless products are compared against each other.

Server vs serverless

Server	Serverless
- No cold starts - No vendor lock-in so one can choose any programming languages - No function timeouts which is critical for long-running tasks	- Low cost - Scale automatically to cope with traffic surges - Less effort for maintaining project infrastructure

- Advanced monitoring	- Simplified backend code - Reduced packaging and deployment complexity

Table 3: Benefits of server and serverless architecture.

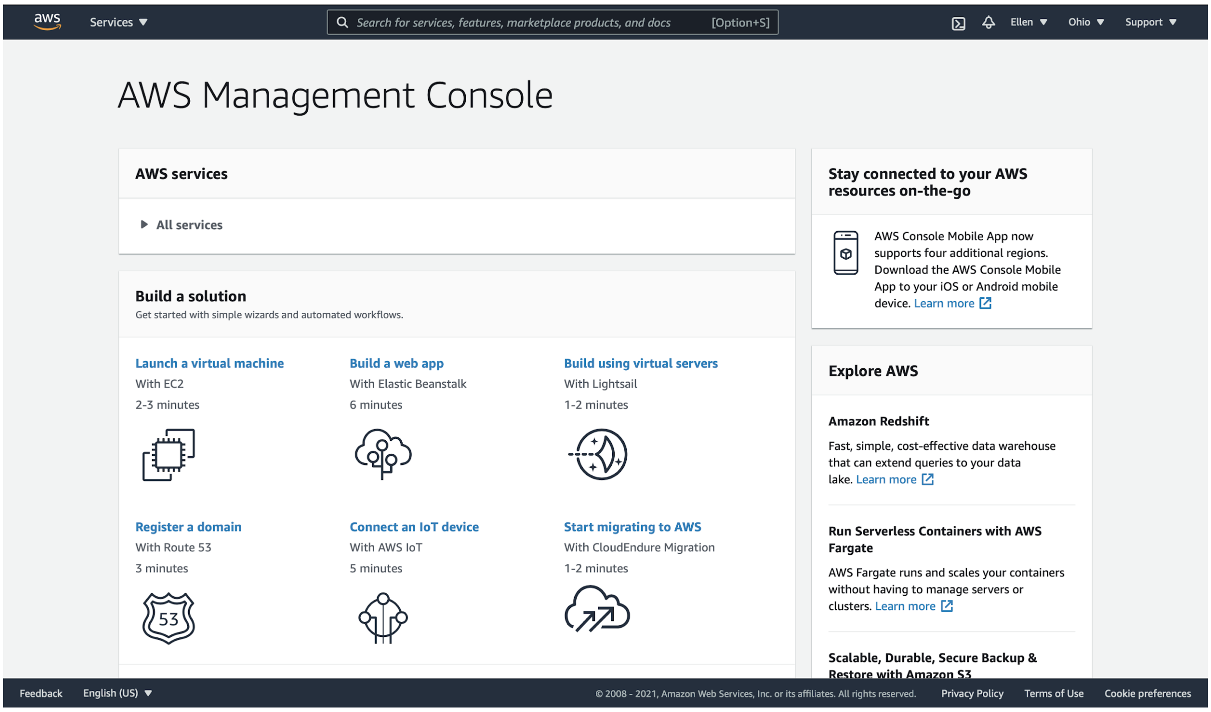
Even though the traditional server architecture gives complete freedom over programming language choices, popular programming languages such as Python, Java, Node.js and Golang are supported by serverless architecture. In addition, Python and Golang can considerably lower cold start times, and there are no time-critical tasks involved in this project. Thus, cold starts appear less problematic when considering serverless architecture. The simplicity resulting from serverless architecture is appealing given that we are new to web hosting. Therefore, we will use serverless architecture.

Serverless products and providers

Table 4: Comparison of some main features of different serverless products

	GCP Cloud Function	AWS Lambda	Microsoft Azure Function
Programming languages supported	Node.js, Python, Golang, Java, .NET, Ruby	Java, PowerShell, Golang, Node.js, C#, Python, Ruby	C#, JavaScript, F#, Java, PowerShell, Python, TypeScript.
Maximum execution time	9 mins	15 mins	10 mins
Scalability	Automatic scaling	Automatic scaling	Automatic scaling
Function limit	1000 functions per project	Unlimited	Unlimited
Concurrent executions	Upto 80 Concurrent executions	Upto 1000 Concurrent executions	Capped at 10X the number of core on the VM
HTTP(s) invocations	HTTP trigger	API Gateway	HTTP trigger

From Table 4, it is clear that these serverless products are generally similar to each other and thus, the UI of their web portals really affects our decision. Comparing Figure 6, Figure 7 and Figure 8, it is clear that Google Cloud Platform has the most simple and intuitive web portal. Thus, since our team has no prior experience with serverless products and is given limited time to complete the project, we will use Cloud Function provided on Google Cloud Platform.



aws
Services ▼
Search for services, features, marketplace products, and docs [Option+S]
Ellen ▼
Ohio ▼
Support ▼

AWS Management Console

AWS services

▶ All services

Build a solution

Get started with simple wizards and automated workflows.

Launch a virtual machine
With EC2
2-3 minutes

Build a web app
With Elastic Beanstalk
6 minutes

Build using virtual servers
With Lightsail
1-2 minutes

Register a domain
With Route 53
3 minutes

Connect an IoT device
With AWS IoT
5 minutes

Start migrating to AWS
With CloudEndure Migration
1-2 minutes

Stay connected to your AWS resources on-the-go

AWS Console Mobile App now supports four additional regions. Download the AWS Console Mobile App to your iOS or Android mobile device. Learn more

Explore AWS

Amazon Redshift

Fast, simple, cost-effective data warehouse that can extend queries to your data lake. Learn more

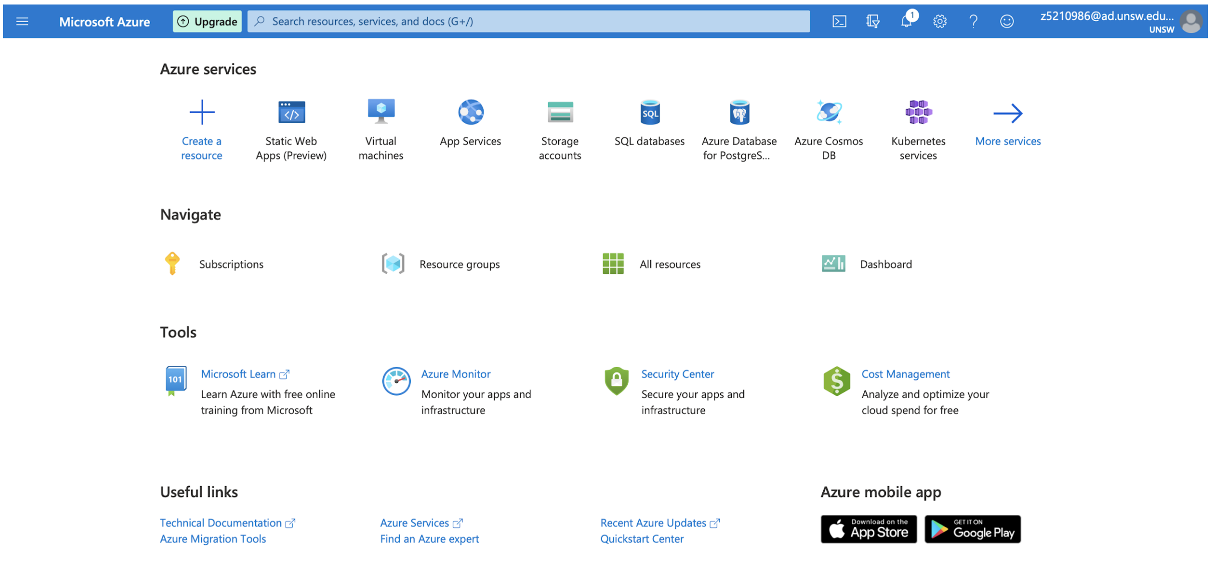
Run Serverless Containers with AWS Fargate

AWS Fargate runs and scales your containers without having to manage servers or clusters. Learn more

Scalable, Durable, Secure Backup & Restore with Amazon S3

Feedback
English (US) ▼
© 2008 - 2021, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use
Cookie preferences

Figure 6: AWS web portal



Azure services

Navigate

Tools

Microsoft Learn
Learn Azure with free online training from Microsoft

Azure Monitor
Monitor your apps and infrastructure

Security Center
Secure your apps and infrastructure

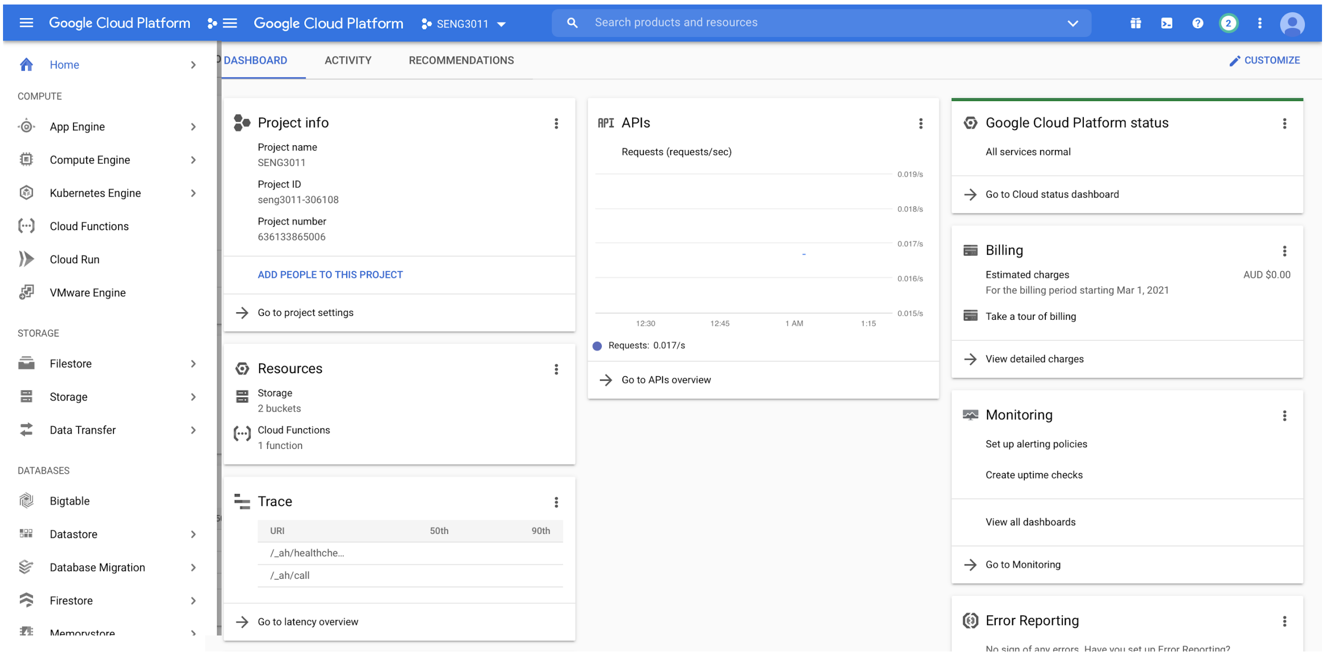
Cost Management
Analyze and optimize your cloud spend for free

Useful links

Technical Documentation
Azure Migration Tools
Azure Services
Find an Azure expert
Recent Azure Updates
Quickstart Center

Azure mobile app

Figure 7: Azure web portal



Project info

Project name
SENG3011

Project ID
seng3011-306108

Project number
636133865006

ADD PEOPLE TO THIS PROJECT

Go to project settings

Resources

Storage
2 buckets

Cloud Functions
1 function

Trace

URI	50th	90th
/_ah/healthche…		
/_ah/call		

Go to latency overview

APIs

Requests (requests/sec)

Requests: 0.017/s

Go to APIs overview

Google Cloud Platform status

All services normal

Go to Cloud status dashboard

Billing

Estimated charges AUD $0.00
For the billing period starting Mar 1, 2021

Take a tour of billing

View detailed charges

Monitoring

Set up alerting policies

Create uptime checks

View all dashboards

Go to Monitoring

Error Reporting

No sign of any errors. Have you set up Error Reporting?

Figure 8: Google Cloud Platform web portal

Frontend

Since the frontend will interact only with the API endpoints, we have complete freedom over language choice. We will choose JavaScript as it is inherently designed for frontend web development and supports all popular browsers. Although plain JavaScript is capable of building a website, JavaScript frameworks and libraries make the development process much faster and create web applications that load faster and have an aesthetic UI. Therefore, we are led to the comparison of popular JavaScript frameworks/libraries which is presented in Table 5.

Framework /Library	Angular	React	Vue
Pros	- Lots of built-in features - MVC structure allows separation of concerns - Good community support - Support	- Smooth learning curve - High performance due to virtual DOM implementation and rendering optimizations - Good community support	- Smooth learning curve - Tiny size (20KB) which gives better performance - High performance due to Virtual DOM implementation

Cons	- Steep learning curve due to complex Syntax - Using real DOM which causes slower rendering - Largest size (500KB)	- Mixing templating with logic (JSX) - Provides only the View part of the MVC model so that extra libraries are needed to implement state and model	- Lack of documentations and community support - Difficult to integrate into large projects
Popular browsers supported	Chrome, Firefox, Safari, IE(11+)	Chrome, Firefox, Safari, IE(9+)	Chrome, Firefox, Safari, IE(10+)

Table 5: Comparison of popular JavaScript frameworks/libraries

In general, all these JavaScript frameworks/libraries bring simplicity to frontend development. However, we will use React instead of the other two as our team is more familiar with React.

Backend

Language	Speed	Firebase/ Community support	Group Familiarity	Frameworks & Libraries support	Integrations, APIs, SDKs
Python	Moderate	Moderate	5/5	High	3/5
JavaScript	Fast	Thorough	3/5	Moderate	3/5
Node	Fast	Moderate	1/5	High	3/5
Java	Moderate	Not much	5/5	Low	4/5
C#	Very Fast	Minimal	2/5	Minimal	2/5

Table 6: Comparison of common backend programming languages

From Table 6, Python will be chosen as the best option among these languages because of the group familiarity and moderate-high average score. 100% group familiarity also means that the learning curve is least and we can spend more time on adding additional features to the web stack.

Below is a further pointer comparison between the languages. Describing why we will use Python.

Python vs JavaScript vs Node.js vs Java vs C++

Python

- Good flexibility→ several Python implementations integrated with other programming languages.
- Many frameworks available → Scrappy will be used for our web scraping.
- High level, object-orientated.
- Dynamically typed, known to be concise and readable → make developing apps faster than using languages like Java.
- Open-source language → multiple libraries available.

JavaScript

- JavaScript is a complex to understand, debug and learn language. On the other hand, Python is a user-friendly, easy to learn language.
- Does not support asynchronous programming.
- Client-side security is at risk with JavaScript.

Java

- Memory consuming and slower than compiled languages i.e. C++ → run 5PHP sites in equivalence for one java site → expensive hosting.

Node.js

- Inefficient in handling CPU-intensive apps- "generating audio, video, editing graphics are some concurrent requests which cannot be currently managed"

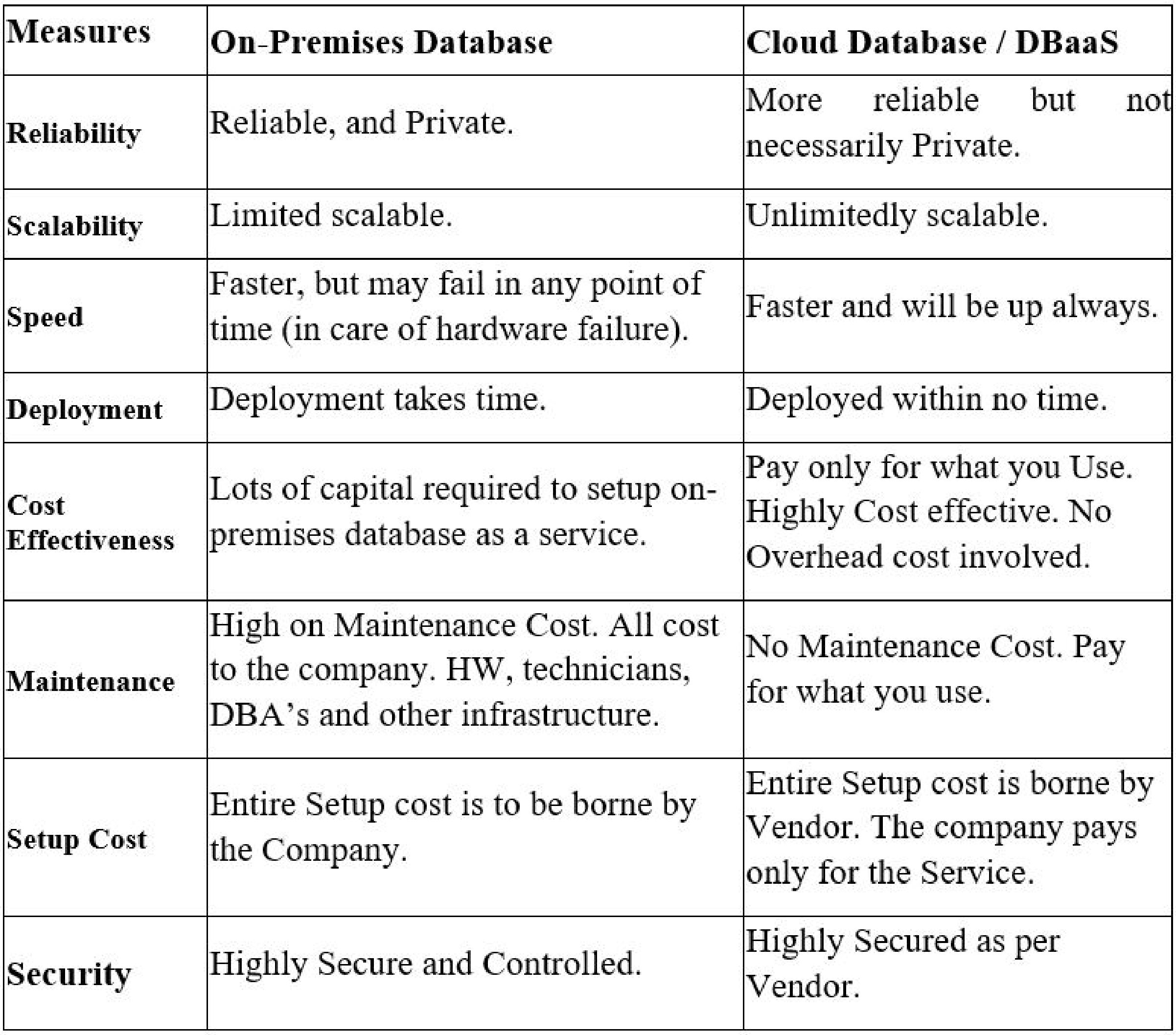
C#

- Low security
- Complex to debug
- Bootstrap issues
- C# is not used on the client-side because there isn't good community support and infrastructure available.
- Easier to create templates using other languages → implementation is more difficult

Python will be used as the main language for our project. Given it's flexible nature, it is best able to integrate with all other frameworks chosen within our web stack. Also, all team members are highly familiar with using Python. Additionally, it has multiple libraries and frameworks available which will be essential given the dynamic range of features our website will have i.e. web crawling, graphing, charts, other APIs.

Database

For storing our data we will be using a real time firebase database provided by google. Table 7 lists several reasons for choosing firebase for our project.



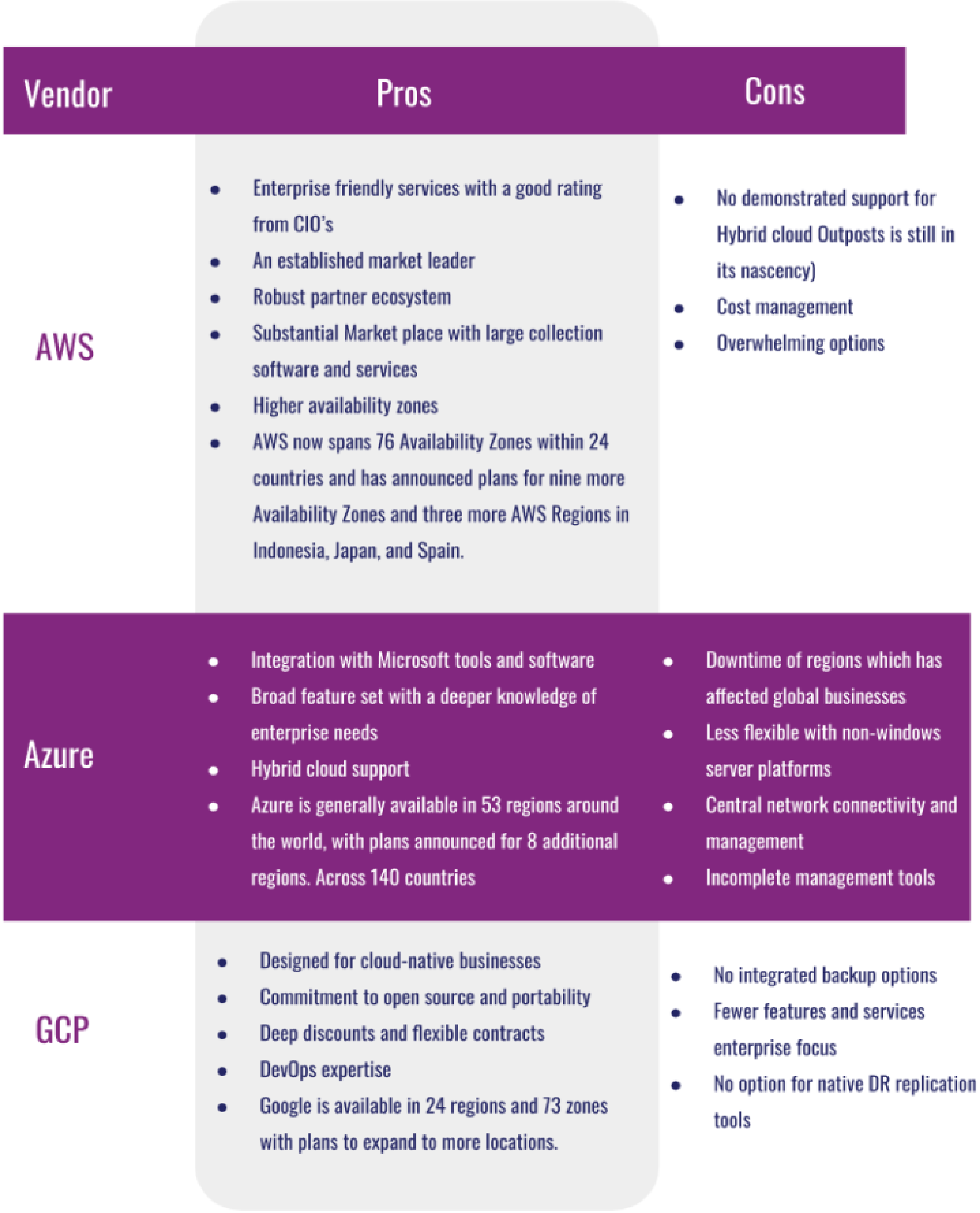
Measures	On-Premises Database	Cloud Database / DBaaS
Reliability	Reliable, and Private.	More reliable but not necessarily Private.
Scalability	Limited scalable.	Unlimitedly scalable.
Speed	Faster, but may fail in any point of time (in care of hardware failure).	Faster and will be up always.
Deployment	Deployment takes time.	Deployed within no time.
Cost Effectiveness	Lots of capital required to setup on-premises database as a service.	Pay only for what you Use. Highly Cost effective. No Overhead cost involved.
Maintenance	High on Maintenance Cost. All cost to the company. HW, technicians, DBA's and other infrastructure.	No Maintenance Cost. Pay for what you use.
Setup Cost	Entire Setup cost is to be borne by the Company.	Entire Setup cost is borne by Vendor. The company pays only for the Service.
Security	Highly Secure and Controlled.	Highly Secured as per Vendor.

Table 7: Comparison of databases

After analysing the comparison between on premise database options and cloud storage shown in Table 7, we will use cloud storage for the following reasons -

1) Requires a huge amount of capital which we as students do not have.
2) Cloud services are free for initial use and easily expandable. Hence perfect for a student project.
3) It also serves the purpose of experimenting with a Cloud database.

All points above indicate that a cloud database will be more suitable for us -



Vendor	Pros	Cons
AWS	• Enterprise friendly services with a good rating from CIO's • An established market leader • Robust partner ecosystem • Substantial Market place with large collection software and services • Higher availability zones • AWS now spans 76 Availability Zones within 24 countries and has announced plans for nine more Availability Zones and three more AWS Regions in Indonesia, Japan, and Spain.	• No demonstrated support for Hybrid cloud Outposts is still in its nascency) • Cost management • Overwhelming options
Azure	• Integration with Microsoft tools and software • Broad feature set with a deeper knowledge of enterprise needs • Hybrid cloud support • Azure is generally available in 53 regions around the world, with plans announced for 8 additional regions. Across 140 countries	• Downtime of regions which has affected global businesses • Less flexible with non-windows server platforms • Central network connectivity and management • Incomplete management tools
GCP	• Designed for cloud-native businesses • Commitment to open source and portability • Deep discounts and flexible contracts • DevOps expertise • Google is available in 24 regions and 73 zones with plans to expand to more locations.	• No integrated backup options • Fewer features and services enterprise focus • No option for native DR replication tools

Figure 9: Comparison of cloud database providers

Although AWS is the market leader in providing all kinds of cloud services. We will use GCP real time database because of the following reasons -

1) AWS offers a multitude of in depth facilities which are not required for our specific use case of a small project. The options it offers are overwhelming for beginners. AWS is a better fit for larger corporations.

2) The GCP real time database stores data as JSON and is synchronised in real time to every client as compared to other google databases. This works perfectly for us as our API has to return the data in a JSON format as well.

3) Since we will be using google cloud functions to host our API. It is only suitable to consider the google real time database option since they belong to the same platform.

4) Microsoft Azure as mentioned above can be potentially problematic for Non Windows platforms. Given that a significant portion of the team uses Mac we feel we should not consider it.

Deployment & Development Environment

Reasons why we choose FireBase and GCP Cloud Functions

Firebase is a set of solutions: Data Storage, Authentication, Hosting, User activity analysis, Crash data collection, etc. Using firebase we can actually avoid using several different SDK.

One of its solutions, Cloud Firestore is a NoSQL database that can handle huge volumes of data and has excellent expandability, which is precisely what we need to store data collected from continuous web scraping. It’s cloud-hosted, easy to maintain and integrates with cloud functions and other tools on GCP.

Cloud Functions for firebase is a serverless framework, which can automatically run backend code in response to events triggered by Firebase features and HTTPS requests. It’s easy to integrate with other tools on the Firebase platform and it has zero maintenance cost as the Firebase automatically scales up computing resources to match the usage patterns of the users.

In general, Firebase is charged based on the usage and starts for free, so it’s definitely the economic solution for students to use.

Virtual Environment

We will be using a virtual machine like Vmware or Virtual PC to simulate environments like Linux and overcome OS and architecture problems. However, we will finally deploy the functions to GCP Cloud Function and store data on Cloud Firestore, hence we don’t really need to consider much about the actual architecture as it will work with any platform.

Integrated Development Environment

We use **Visual Studio Code** as our primary IDE

- It has an easy-to-use interface, powerful functions like syntax highlighting, bracket-matching, auto-indentation, box-selection, snippets etc.
- Integrated Terminal for easy git pull and push and Build-in Marketplace that lets us install libraries we need in a very convenient way.
- Support on MacOs Linux and Windows, easy to code in any circumstance**.**

Operating System

We will be using MacOS to build and test the back-end, it has a nice terminal and ssh to conveniently transfer files between local and server, and it's the most commonly used by our teammates. Front-end and Web Scraper will be both coded using JavaScript and Beautiful Soup 3.9.3 and hosted on GCP cloud function, which will be compatible with Windows, Linux and MacOS.

How to achieve Cross-browser compatibility

- Keeping our code simple is the best and simple way to achieve this, as the simplicity in coding is a fundamental principle in any circumstance. Simple code means less bugs and easier to debug as well as transfer the website between browsers.
- Add DOCTYPE at the beginning to avoid 'quirk mode'. DOCTYPE is an instruction at the beginning of your code that tells browsers which language you used to write your code. And "quirk mode" is a backup plan for browsers to render old websites that don't have a DTD or outdated DTD version below HTML4, that is not the best way we intended to render our pages. Thus DOCTYPE is an important element we need to remember.
- Create CSS reset style sheets at the beginning of our CSS style sheets, to ensure different browsers can easily identify their style sheet and implement the appropriate default values throughout automatically. Which will make the display properly on any browser.
- After all, we will still keep testing across browsers while coding the front-end to make sure the code is compatible and prevent deviation.

Deliverable 2 Updates on API Design

Our API has has 3 major components in its architecture -

1) The web scraper hosted on a gcp function
2) The API hosted on a seperate gcp function
3) The firebase realtime database

Largely, Our API has the same architecture as we had planned in the D1 report

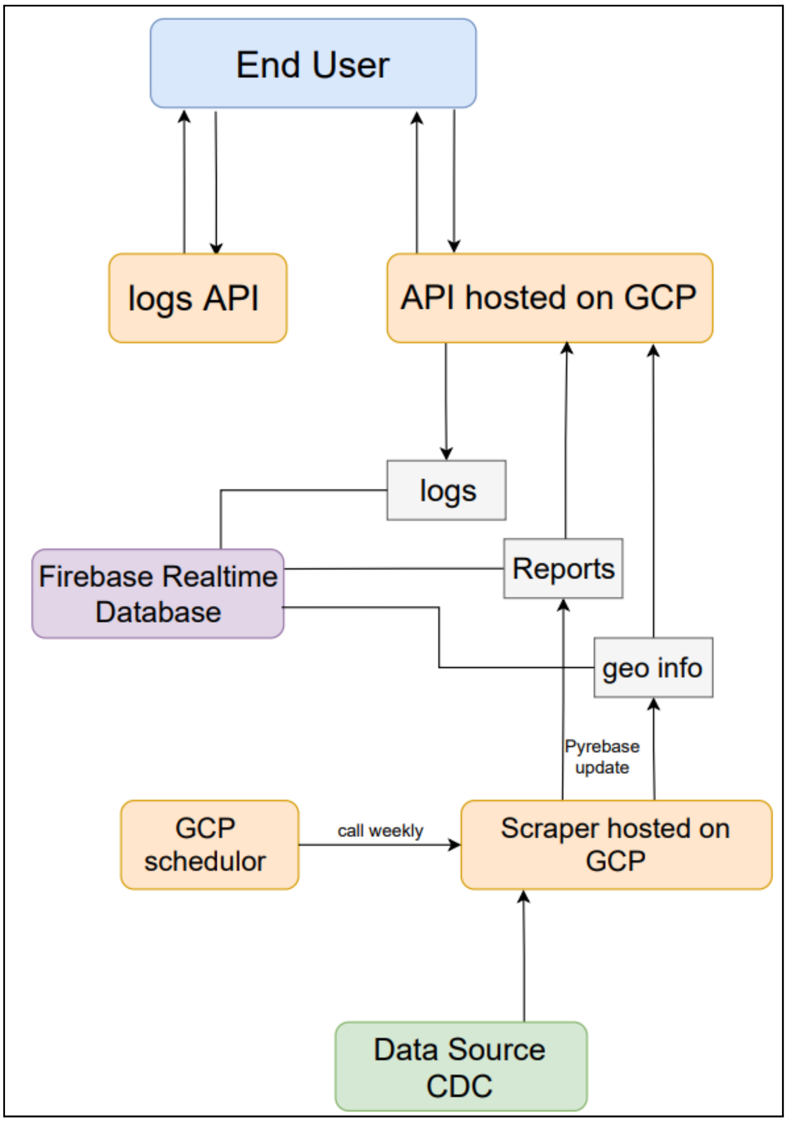


Figure 10: Database & Scraper interacting with the API

Web Scraper

We have two sets of scrapers. One set of scrapers only focus on past outbreak reports that will not change so that these scrapers will only be run once locally to push the data to our database. The other web scraper will focus on recent outbreak reports listed on extracts information from the CDC website every week. We only extract every week because the CDC website is not dynamic and since the past one month that we have been working on CDC, not many reports were posted on the site. We have used a combination of BeautifulSoup, Requests, Geocoder and Re library to conduct the scraping. The web scraper is hosted as a separate cloud function. For each run the scraper will create disease reports by collecting information from the CDC website and adding them to our database. We have also used spacy for natural language processing, it extracts important pieces of information like locations from the scraped data. The scraper also makes use of the python geocoder library to extract unique locations from the text.

Google Cloud Platform

The API and the web scraper are hosted as GCP functions. The plethora of advantages of GCP functions and cloud based hosting are detailed in Deliverable 1.

Firebase RealTime Database

There was a conflictive discussion between choosing RealTime Database and Firestore in our Deliverable 1 report. After doing some research online, we have decided to use RealTime Database for data storage. RealTime Database stores data in JSON format whereas firestore stores data in the form of documents and collections. We chose the realtime database out of the two because we have to return our data in a JSON format as specified in the project specification. In addition, it also has very low latency hence allowing for quick updates to the database.

Choice of implementation

We stuck by our earlier choice of implementation for using the backend language (Python), Google Cloud functions, Database (Firebase Realtime Database), Web Scraper (BeautifulSoup, Requests).

We used additional libraries that we had not discussed in our deliverable 1 report:

Spacy (Extract locations mentioned in a report): Spacy uses Natural Language Processing to extract different entities from text and one of these entities is location.

Geocoder (Convert location name to corresponding geonames_id): The basic structure to store location information requires us to provide both country name and place name which is extremely hard to extract since Spacy returns a list of places mentioned in the input text without labeling them as either country or city. Thus, we have decided to convert place names to their geoname ids and this makes our scrapers much simpler. Additionally, geocoder also provides methods to find hierarchy of a place which enables geographic locations to be matched at different levels.

Re (Regular expressions for date extraction): Re servers purposes in our API development. Firstly, it is used in our scrapers to extract dates from the outbreak report. Secondly, Re is also to validate user inputs.

Challenges Faced

Communication Problems

a) As the team members live in different timezones, we faced communication problems. Deciding a meeting time which was comfortable for all 5 of us was tricky at times.
b) We also faced a major miscommunication problem when we only scraped reports of U.S based outbreaks although we had to scrape reports for all diseases in the disease list.
c) Due to division of labour, we had to wait for the person responsible for that section to respond in order to clear our doubts or solve a bug.

Time Management

a) Our lack of time management led to some last minute issues, but this did not stop us from working hard and delivering our best.

Web Scraping

a) HTML pages on the CDC website have inconsistent formats. Due to this we had to make customised scrapers for each individual disease, which was a very tedious task.

New Experience with Cloud Services and Libraries

a) Since all of us had never worked with cloud services, we had to learn how GCP Cloud functions and Firebase Realtime Database work, this took up valuable time we could have spent on adding functionalities.

b) We also had to work with new libraries like BeautifulSoup, Spacy and geocoder which took up time we could have spent on the API.

c) Using the geocoder to get the location information was also a challenge. Initially, the problem was deciding the query logic between the child method and the hierarchy method, and we realized the hierarchy method was always faster since the child method is nested and involves searching down all the trees. The second problem was that even if we used the hierarchy method for each API request, it would still be very slow as it would involve repeated calls to the geocoder API. Our solution was to save the geo information into our database and check it prior to the API call

Shortcomings

1) The team was unable to get reports for all diseases in the disease list because of the lack of outbreak information on the CDC website.
2) The team wanted to implement a CI/CD pipeline which would have allowed us to constantly integrate new code and focus on the requirements of the application rather than deployment. But due to the paucity of time we were unable to add this feature.
3) The team had also planned to program the scraper such that it scrapes important information about deaths, hospitalisation etc. But again due to lack of time management we were unable to add this feature.
4) We only added the basic functionality of a GET request in the API and were unable to add other kinds of requests ex POST, DELETE.
5) We were not able to dedicate much time to the testing of the API as we kept a greater focus on the scraper.