API DESIGN

Deliverable 2

Team QQ

SENG3011

CONTENTS

1 SYSTEM DESIGN

1.1 HIGH-LEVEL DESIGN

1.1.1 Overview (Updated from Deliverable 1)

The design of our system has remained the same from deliverable 1. The following is the design we settled on in deliverable 1. There is updated information in the individual subsystem sections and in the deployment section.

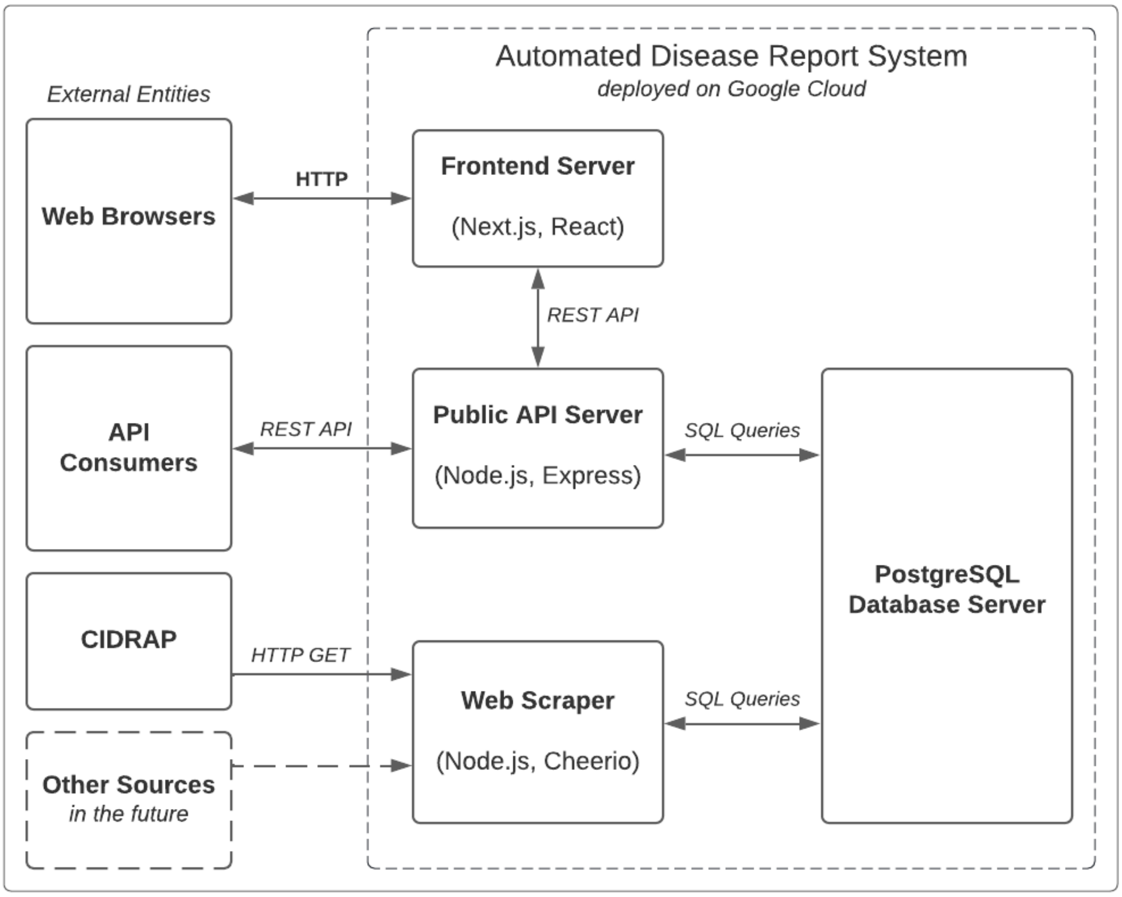
The overall purpose of our system is to extract information from CIDRAP and allow customers to access this information using a Public API we developed. Our entire system relies on the Web Scraper being able to gather information from CIDRAP, where from there it is processed and available through our Public API and frontend website.

Firstly, the Web Scraper automatically extracts data from articles published by CIDRAP using HTTP GET Requests. *Cheerio,* the library we are using for scraping does not produce visual rendering, load CSS or any external resources, or execute JavaScript but more-so parse markup therefore the scraping process is very quick. The scraped data can then be stored into our PostgreSQL Database automatically as the Scraper will make a connection to the database and automatically execute SQL queries (mainly INSERT) as data is gathered. The PostgreSQL Database is set up so that it is always online, therefore database calls from both the Web Scraper and Public API Server are reliable and always return a result.

The Public API Server makes a connection to the Database and uses SQL Queries (mainly SELECT) to extract the information. Information gathered by the API Server is accessed in two different ways; by customers using the API itself and by the Frontend Server. Customers are able to access information from the Public API Server using REST API methods, namely the GET method. Documentation and instructions on how to use the API has been developed using *Swagger*, where customers have been provided with a well-formatted set of examples and required inputs for each route endpoint. The Frontend Server will use the same available API methods to access information from the Public API Server. Customers will be able to access the Frontend Server using any modern web browser, and information on it will be displayed in a strategic manner to maximise user experience whilst browsing the website.

The Web Scraper, Public API Server, and Frontend Server are all deployed using Google Cloud. The Database Server is deployed on AWS. Please see the deployment section for more detail.

1.1.2 Visual System Design (Same as Deliverable 1)



1.1.3 Cohesion

The main method we have used to achieve high cohesion in our design is through the division of each component in the backend and frontend. High cohesion has been achieved by making each module only concerned about a smaller subset of the overall problem the system is trying to solve. In the backend, there is a clear distinction between the tasks of persistently storing information and processing the information based on user actions. In our system, these two tasks are separated into the PostgreSQL database and Public API Server. We have also considered the *single-responsibility principle* whilst designing our system to ensure a high level of cohesion is achieved. This principle states that each module in a system should have a single responsibility over a specific part of the system, and it should be entirely responsible for the implementation of part. Adhering to this will ensure that elements in a module are highly cohesive.

1.1.4 Coupling

The only communication between the backend and the frontend is done through a JSON REST API. The JSON API will be designed such that a minimal number of requests will be sent between the frontend and backend to perform tasks. This reduces the coupling between the frontend and the backend as the frontend does not need to know what actions the backend is performing when requests are sent. The backend does not need to know anything about how the frontend is generating the interface or what is causing the requests to be sent, it will just process the JSON requests accordingly.

The modular design of our system will reduce coupling and in turn have many benefits:

- It simplifies the delegating of work for team members as they can be assigned to a specific component of the system.
- The system is more scalable, and in this case one possibility (as visible in 3.1.2) is to add other sources to scrape from.
- Implementation will be made easier, and the codebase is more maintainable and as changes in one component such as the Web Scraper will not have a direct impact on any other components.
- The process of testing and debugging will be simplified as problems can be pinpointed into a specific component rather than the entire system.

1.2 WEB SCRAPER

Not much has changed with the scraping process since deliverable 1, however the following has been updated to fit the final design. Sections quoted from Deliverable 1 have been marked with “Same as Deliverable 1”.

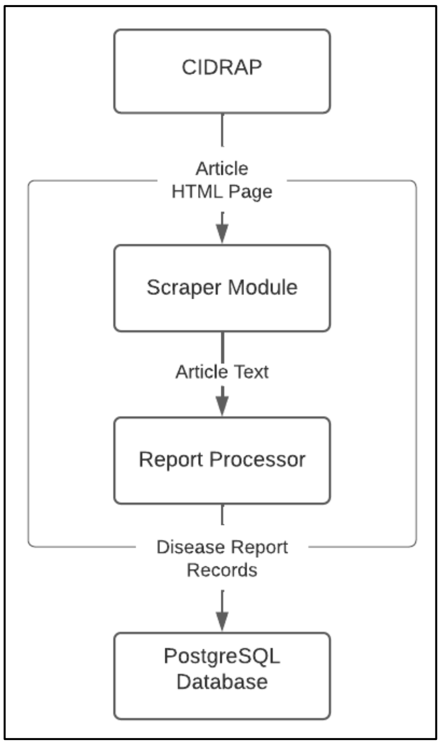
The purpose of the web scraper is to scrape and process articles, and to store the disease reports in the database server. The scraper runs every 1-4 hours, keeping the reports up to date without overwhelming CIDRAP or using too much processing power. The scraper uses HTTP requests to fetch the CIDRAP content, process it, then saves the results to the PostgreSQL database server.

This design has many benefits. Having the scraper run autonomously, saving results into the database, means that the system is not dependent on CIDRAP being online. If CIDRAP has scheduled maintenance or is down due to a technical issue, the public API and frontend will remain fully functional. The second benefit to this design is that CIDRAP is only send requests once every few hours. If the API scraped CIDRAP every request it received, CIDRAP would receive hundreds more requests and it might block our servers for denial of service. This bad design would also cause the scraper to do a lot of redundant work, wasting resources and money. Our system effectively caches articles and reports in the database for immediate access on request.

1.2.1 Pre-release Bulk Scrape

We think it is important for our system to also contain historic disease reports from CIDRAP. To facilitate this, our Node.js scraper script has the ability to be run offline. In this offline mode, the user can specify the number of CIDRAP list pages to scrape from, where each list page contains around 10 articles. We have manually run the scraper on the past 100 list pages, scraping roughly 1000 articles. This script uploads these results to the database. This pre-release bulk scrape ensures the system is useful from the beginning and it doesn’t need to be online for months before it is useful.

1.2.2 Scraping Process



The scraping process is roughly similar to the design in deliverable 1 but there are a few minor differences. The scraper is now made up of two subsystems, not three as in deliverable 1. The scraper module functions the same as described in deliverable 1; it traverses the CIDRAP article catalogue and fetches recent article content. It extracts the information using the Cheerio library and outputs the raw data.

This data is then fed into the second subsystem, the report processor. This system takes the text and uses the natural language processing library “Compromise” to find diseases and the corresponding report location. First, it uses the languages “match” function to find diseases using their list of possible aliases. Then, it uses the library’s “look behind” and “look ahead” functions to check behind and ahead of the found disease for a potential location. This is made significantly easier through Compromise’s location detection functionality. We only need to tell the library to look for locations, we do not need to specify a list of possible locations. Unfortunately, the library does not have inbuilt detection of diseases however it can detect them if a list is provided. These disease and location pairs are converted into a list of reports which are saved in the database.

1.2.3 Technologies

1.2.3.1 Node.js (Same as Deliverable 1)
The scraper has been built using Node.js. Node.js is a backend JavaScript runtime environment which allows rapid development of scalable and distributed systems. Node.js comes with an extremely useful package manager NPM which provides access to thousands of open-source libraries.

We chose Node.js because it has recently become somewhat of an industry standard for backend server solutions. Although not all members of the team are familiar with it, using it for our project opens up a great opportunity to develop our skills in a tool we might use in the real world. Because of its popularity, Node.js has a rich ecosystem of documentation and guides for almost anything you'd want to do with it. The language's rich library ecosystem means that it only takes a small amount of code and effort to build a complex web system. This is in contrast to a language like C++ which could take teams of developers to make a secure web service from scratch.

1.2.3.2 Cheerio (Same as Deliverable 1)
Cheerio is an open-source library found on NPM which parses HTML markup and allows it to be read and manipulated in a similar manner to jQuery. To extract content, Cheerio simply requires the HTML page as a string. The resulting structure can be queried using jQuery style queries, for example,

```
$('#body').text();
```

returns the content of an element with the id 'body'.

We chose Cheerio because it is a simple to use immediate HTML processing library which is not a web crawler. This is useful because our scraper is built as a cloud function; it is run every hour and it is offline between runs. This design means that our scraper is not a "web crawler;" it does not need to constantly run, updating data in real time. We can simply feed our HTML into cheerio every hour and process the output accordingly.

1.2.3.3 Alternative: Regex for HTML Processing (Same as Deliverable 1)
We chose not to use Regex for the HTML processing because Regex is not built to parse HTML. Regex is best for simple text patterns, while HTML is a very complex markup language. Tools like XML parsers are designed from the ground up to efficiently and intuitively read XML and HTML files and should be preferred. Regex provides too much room for error and makes the code a lot less readable.

1.2.3.4 Alternative: Python with Scrapy (Same as Deliverable 1)
The main alternative technology choice we had was to use Python with the scraping tool Scrapy. Python is an extremely popular, simple and highly readable language. It allows for rapid development, and has a rich ecosystem of documentation and libraries. We decided against Python for a number of reasons. Although we are all well versed in the language, we wanted to challenge ourselves and learn a new, industry-standard language in the form of Next.js.

Additionally, Scrapy is an open-source web crawling framework for Python. Because of its nature as a web crawler, it has to be running on a server all the time, checking for updates to the website. This didn't fit our design as we wanted a scraper that would only run periodically, processing updates and uploading them to the database. These reasons led us to decide against Python and Scrapy.

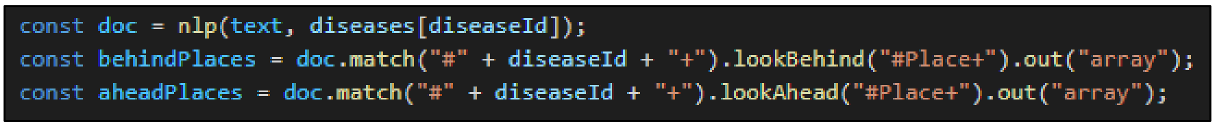
1.2.3.5 Alternative: Go (Same as Deliverable 1)
Another option was for us to use the language Go for the scraper. Go is an innovative, statically typed and compiled programming language built for web systems, made by Google. Go's main selling point is how it allows users to write highly performant code in a simple language built for multithreading.

The main reason we decided against using Go was because none of our team had much experience with it, and it is quite to Python and JavaScript which we are familiar with. Go has a good collection of libraries for scraping but this didn't change our decision to not use the language.

1.2.4 Compromise for Natural Language Processing (New)

At the time of the deliverable 1 report, we had not thought much about our approach to natural language processing (NLP). Natural Language Processing is the process of extracting meaningful information from long strings of text. In our case, the challenge was extracting the reported diseases and locations from the CIDRAP articles.

When researching libraries we could use for this task, we came across "Compromise". Compromise is an NLP library for Node.js which has many useful features to extract information from text. From its GitHub README, the core principle of the library is that it is small, quick, and often *good-enough*. It is not expected for the library to be accurate 100% of the time, but it strikes a good balance between accuracy and performance. This means that our reports aren't be 100% accurate, but they are generated fast and are accurate enough. An example of how compromise is used in our article processing is shown below.



```
const doc = nlp(text, diseases[diseaseId]);
const behindPlaces = doc.match("#" + diseaseId + "+").lookBehind("#Place+").out("array");
const aheadPlaces = doc.match("#" + diseaseId + "+").lookAhead("#Place+").out("array");
```

For each disease, the disease aliases are first fed into the NLP function of "Compromise". An example of disease aliases would be "COVID", "COVID-19" and "SARS-CoV-2". These are all different names for the same diseases. Next, we search the processed document for diseases using the "`.match`" method. We then look for place names ahead and behind of each disease, outputting the places as an array. Since this is done for each disease type, for each disease we get a list of possible outbreak locations. The system then picks the location closest in proximity to the disease word in the text as it is most likely to correspond to the case location. This result is then used to generate a report. If no location is found, no report is generated.

1.3 PostgreSQL Database

We have chosen PostgreSQL to hold the web scraped articles and reports. This is a relational database built for scalability, so it is perfectly suited to holding a large amount of data which will be the product of scraping news articles often.

1.3.1 Database Design (Updated from Deliverable 1)

Please see database schema in the appendix.

The database design has evolved since the initial deliverable 1 design. The database is now made up of six tables: one for articles, reports, diseases, disease aliases, symptoms and logs. Each article may have one or more reports associated with it via the *`Report.article_url`* foreign key. Each report has a single disease referenced by the foreign key *`Report.disease_id`*. Each disease has one or more aliases stored in the table `DiseaseAlias` with foreign key `disease_id`. Each disease also may have one or more symptoms associated with it through the `Symptom.disease_id` foreign key. The `Log` table sits separately and is not related to any other tables.

We updated this design because our old design did not facilitate the data we needed to store. First and foremost, the previous design did not account for the need to store logs. The reasoning behind why we now need to store logs in the database is discussed in later sections. We also realized that storing the aliases and symptoms as a string is not very efficient and reduces the querying power of the database. With disease aliases and symptoms split, it opens up the possibility of querying diseases with a specific symptom or finding the disease ID of a particular disease alias.

The background reasoning of why we chose a relational database is the same as deliverable 1. Relational databases come with numerous benefits due to their design, including atomicity and type checking. Through the atomicity of the database, we will be able to ensure there are no concurrency issues between the scraper and the API. Type checking and having constraints on our data will ensure the data is consistent and can be reliably queried. Additionally, a relational database will allow us to perform join operations

1.3.2 Access in Node.js through Knex (Same as Deliverable 1)

To generate and send queries to the database, the scraper and API server will use the Knex library. Knex is a query builder for Node.js which also facilitates connection to most types of SQL database. In Knex, queries are built by calling functions like the "builder" design pattern, as shown below.

```
knex('users')
      .groupBy('count')
      .orderBy('name', 'desc')
      .having('count', '>', 100)
```

This builder pattern is great because it makes the queries more JavaScript friendly. Variables can be easily parsed in as parameters without having to worry about string concatenation or SQL injections. Knex also makes the queries SQL language agnostic. This means the Knex queries work for all SQL languages such as PostgreSQL, MySQL and SQL Server.

Knex also handles connecting to the database server through a secure channel. It creates and manages a "connection pool" when sending requests to the database. This means that instead of starting then dropping connections every time we want to query, Knex maintains a handful of persistent connections which are recycled as needed. This greatly improves the performance and scalability of database operations.

1.3.3 Deployment (Updated from Deliverable 1)

TODO: Change to AWS

In deliverable 1, we decided on hosting the database on Google Cloud Platform (GCP). During the development process, we were forced to reevaluate this choice when we realized the GCP database was using a significant amount of our free trial credit. We decided to switch to a different database provider but to keep using GCP for all other hosting needs. We settled on AWS because it is pretty much equivalent to Google Cloud when it comes to database hosting. AWS has a much more competitive free trial with the database which allows us to run it for the duration of the project. AWS automatically provisions the servers and manage their capacity to ensure we are not wasting any space and can scale up easily. Additionally, AWS supports high availability, so data will always be accessible. This increases the reliability of our API so users can be confident a result will always be returned.

1.3.4 Alternatives

We have chosen a relational database over a NoSQL database for multiple reasons, with the main one being that we want to preserve the relationship between articles and reports. It is important that when a user searches for an article, they are also able to see all the reports that were made within it. A relational SQL database will allow us to perform a join between articles, and reports table, allowing us to easily find reports within articles, or query reports and articles separately.

Some popular NoSQL databases are Firestore by Google, Atlas by MongoDB and DynamoDB by Amazon Web services. Each have their own benefits, Atlas brings with it integrity checks at the level of an SQL database, in that each operation is atomic and you can add type checking and create a schema for the database. DynamoDB is more fluid and a more traditional NoSQL database, with more flexibility and increased scalability. Firestore brings with it easy integration to a Firebase project and by extension the Google Cloud Platform.

A NoSQL database is more suited to storing related data together so that there is no need for joins. If we were to store the reports within each article document, we would lose the ability to query the reports separately from the articles. Alternatively, storing the reports and articles in separate collections but keeping a reference to each other would defeat the purpose of a NoSQL database. Hence, when working with data that has a composition relationship, i.e., a report cannot exist without an article, a relational database supports this most naturally.

SQLite is also an option however has a few downsides when compared with PostgreSQL. SQLite makes queries directly to a file stored on disc, however PostgreSQL makes requests to a server which then reads from a disc. This mean that SQLite lacks the ability to handle simultaneous access by multiple users at any one time, in contrast PostgreSQL was built with multiple user access in mind and has very well-defined permission levels for users to determine which operations are available. PostgreSQL additionally supports more complex queries than other SQL alternatives, in that we could create functions to search the globe by region, or to find countries near to the goal location.

Another SQL alternative is MySQL. AWS has an option for hosting a MySQL however this would come with a couple of downsides to PostgreSQL. PostgreSQL is more suited to applications that will scale up, given we are aiming for our API to be as scalable with a high capacity to handle lots of traffic, MySQL is the weaker choice. We are also a lot more familiar with PostgreSQL because it is taught in the database course at UNSW.

1.4 API SERVER

The API server design did not change much compared to deliverable 1. This section has been updated to reflect on our actual design choices for the final API. The main function of the API server is to fetch and process data from the database and serve it to the API user. The API serves as an interface by which the data that has been scraped can be accessed. It will later serve as a data source for both our and other teams' web applications.

1.4.1 Technologies

A crucial decision in the creation of an API is the choice of the language and web framework with which to construct the service. To make a suitable choice, we considered the positive and negative aspects of various solutions. It was decided that Node.js and Express would be most appropriate for this project.

1.4.1.1 Node.js (Same as Deliverable 1)

JavaScript with Node.js was chosen as the language for the API server. Node provides an event-driven runtime environment that handles asynchronous I/O. This allows multiple clients to concurrently utilise the API and allows multiple services to operate simultaneously. Node.js's V8 engine also provides **just in time (JIT) compilation**, which offers **higher performance** and **lower memory usage** than interpreted languages.

One drawback of Node is that is not well suited to CPU intensive tasks. It was decided that this would not be an issue as there are no significantly intense tasks that will be performed by the API. Another negative aspect is that the asynchronous model of Node can be difficult to understand, however, the scalability and performance of Node outweighs the initial difficulty of programming with asynchronous functions and call backs.

1.4.1.2 Express (Same as Deliverable 1)

Express is an open-source web framework for Node. It is the most **widely used**, industry-standard web framework used with Node, providing high-performance **routing**, HTTP handling (e.g., caching), and **content management**.

1.4.2 Alternatives (Same as Deliverable 1)

1.4.2.1 TypeScript with Node.js

TypeScript is a superset of JavaScript that provides static typing. This augmentation makes it easy to write correct code and avoid runtime errors as types are validated prior to compilation. TypeScript is transpiled into JavaScript, so it is also compatible with Node and its packages.

Due to the nature of working with a type system, development can become harder and require more work. While a type system would be useful for projects with larger scope and more complicated components, the comparatively simple architecture of our system does not require strong typing.

1.4.2.2 Python with Flask

Python and Flask were strongly considered due to their simplicity and our experience with the technologies. Python is an interpreted language with simple syntax. Flask is a lightweight web framework that provides features such as a development/debug server, a Web Server Gateway Interface (WSGI), and a templating engine.

Python allows for a rapid prototyping and development cycle due to its simplicity and comprehensibility. Since Python 3.5, concurrency in python can be handled similarly to JavaScript with async and await functions, as opposed to manual threading. It is slow however due to the overhead caused by its highly abstract nature.

1.4.2.3 Java

Another solution for the creation of an API backend was the use of Java and a web framework such as Spring Boot. Being an object-oriented language, Java allows for the construction of software using object-oriented design. Furthermore, it would be possible to run the API anywhere that has a Java Virtual Machine (JVM).

While Java is far more performant than JavaScript, the simple nature of our project does not call for such a heavyweight tool. Additionally, concurrency between clients would be harder to achieve and would have to be manually threaded, as opposed to the asynchronous function method in JavaScript and Python.

1.4.2.4 C#

Another similar option was to use the C# language with a web framework such as ASP.NET. C# is similar to Java in that it is also designed for use with object-oriented design and that it has static typing. It also has asynchronous function support. ASP.NET is a web framework for C#. One disadvantage of this combination is that to most simply create a .NET application, Windows must be used for development.

1.5 Deployment

Our service has been deployed mainly using Google Cloud Platform, but with the database hosted on Amazon RDS. Google Cloud is a cloud service that provides a full suite of **hosting, deployment, CI/CD, monitoring** and **analytic tools**. In addition, Google Cloud offers other services such as Software as a Service, Platform as a Service or Infrastructure as a Service. Once components are developed, deployment is done using the `gcloud` command. Google Cloud also handles SSL certificates which encrypt HTTP traffic.

The most notable change between the deliverable 1 report and our current design is how we ended up hosting the PostgreSQL database. Initially, it was planned to be on Google Cloud Platform, but as detailed below, this proved to be far too costly. Amazon RDS offers a more palatable pricing model for their cloud-hosted database, so it made the most sense to switch to that, considering the functionality is the same.

1.5.1 Google Cloud App Engine

The API server is deployed through Google Cloud's serverless "App Engine". App Engine is a platform as a service (PaaS) for deploying and hosting web applications in Google-managed data centres. Applications are automatically sandboxed and are run across multiple servers. This means that the server will be automatically elastically scaled when usage increases. It also increases redundancy as it would take multiple data centres failures before all site servers shut down. The platform as a service model is especially helpful for prototyping and rapid development because we don't have to consider purchasing servers or designing a scalable architecture for deployment. We can spend more energy developing the core functionality of the system.

1.5.2 Google Cloud Functions

The scraper is deployed through Google Cloud serverless functions. Serverless functions allow the deployment of individual tasks which can be triggered by HTTP requests or Google Cloud Schedules. The scraper's functionality is implemented through a single Node.js JavaScript function which is deployed as a Cloud Function. It is triggered automatically by Google Cloud every few hours through

Google Cloud Schedules. This means the scraper can run automatically without a server running 24/7, significantly reducing the cost of the system. Google Cloud functions additionally greatly reduce the complexity of setting up a server to run specific code on a schedule.

1.5.3 Amazon RDS for PostgreSQL

We are currently hosting the database on Amazon Relation Database Services (RDS). As previously discussed, we required a relational database that could handle concurrency issues automatically, hence we chose PostgreSQL. There is limited free support for hosting databases on the cloud, particularly for PostgreSQL databases.

Initially, we planned to host the database on Google Cloud Platform, but this proved to be too costly far too quickly. With a small number of insertions and queries to the database over the course of 2 weeks, we had finished the $300 credit that Google provides upon signup. Considering we have a limited budget, cost was a big factor in our decision. Amazon RDS has a good free tier option, where you are not charged for the database when you keep your activity below a certain level, and they provide free credit to assist with the cost. It is for this reason we elected to host our database on Amazon RDS.

1.5.4 Alternative: Amazon Web Services

The main alternative for hosting the API and scraper would be Amazon Web Services (AWS). AWS is arguably more widely used than Google Cloud and provides a similar set of features. We decided against AWS because Google Cloud has a simpler interface that is easier to use for beginners. Since we have little to no experience with Cloud Service Providers, it made sense for us to use a slightly simpler but just as powerful provider.

Part of this complexity would come from setting up a scraper to run every few hours on an AWS service. It is challenging to trigger these functions on a timer and we would have had to integrate multiple AWS services together in order to achieve this. In contrast, Google Cloud Platform allows you to trigger cloud functions on a schedule.

1.5.5 Alternative: Google PostgresSQL

Google provides a cloud hosted PostgreSQL database with high reliability and up time. We initially used this database as it would make sense to have the entire application hosted all on one platform. However, keeping the database available 24/7 by hosting it all the time ended up costing far too much on the Google Cloud platform. It is for this reason we chose not to host the database on Google Cloud.

1.5.6 Alternative: Microsoft Azure

Azure is a cloud computing service operated by Microsoft for application management via Microsoft-managed data centres. Azure is a lot more enterprise-focused than Google Cloud, which makes it less relevant for our small-scale operation. It also is less open-source than Google Cloud and favours a Windows development environment. For these reasons, it is not the ideal platform for our use case.

1.5.7 Alternative: Virtual Private Server Hosting

Another option would be for us to use a virtual private server (VPS) hosting company like Digital Ocean or Vultr. A VPS is a virtual server hosted by an organisation, usually in the form of a simple Linux server. These servers can be accessed through SSH, and services can be deployed by manually running them on these servers. This approach would be more barebones than using a cloud provider like Google Cloud and has no scalability or automation as is. To make these VPS instances usable, we would have to automate deployment through scripts, containerization or Kubernetes, as analysed below.

1.5.8 Alternative: Cloud Compute Server

Another alternative, similar to VPS hosting would be to use a cloud compute instances such as Google Cloud Compute Engine or AWS EC2 instances. This option has the same pros and cons as VPS hosting, with the added benefit of elasticity. This means that the cloud provider will run your application on a virtual machine with the exact resources it needs. This makes compute instances slightly more scalable and efficient. The cloud provider also provides extra redundancy which reduces downtime. We decided against this option because it has similar problems to VPS servers. It would be too hard to set up and it’s a not very scalable solution.

1.5.9 Alternative: Kubernetes

Kubernetes is an open-source system for automating deployment, scaling and management of containerized applications. Kubernetes clusters coordinate a highly available cluster of computers that work as a single unit. They allow for the deployment of containerized applications to a cluster without tying them to individual machines. Kubernetes would give us more control of the servers our system runs on, allowing us to optimise the best combination of server count, locations and providers to maximise the efficiency of our system. We decided against Kubernetes because no one in our team has experience with it and it would be a lot of work to learn. Google Cloud handles all the hosting and deployment of our applications for us, removing the need for Kubernetes

2 API ROUTE DESIGN

2.1 SWAGGER DOCUMENTATION

Swagger is the documentation tool which our group used to document our API. Apart from it being recommended in the project specification, we decided to use Swagger as it has many benefits listed below:

- Consistently formatted and user-friendly documentation is automatically generated. This allowed for our API to be easily understood and removed the possibility of any human-made errors to exist in the documentation.
- Swagger makes available a 'Try it out' button where users are able to input parameters into text boxes and output from the real API server is returned. This is extremely beneficial as users are able to both view the documentation and experiment with the API with limited action required.
- The documentation is publicly accessible through any modern web browser.
- The Swagger editor encourages reusability of code through the availability of components which can be referenced to multiple times. This feature increases the maintainability of documentation as if a component (such as a 'report' object) is slightly modified due to changing requirements, the documentation can be modified in a single location and the changes will appear throughout the whole Swagger page.
- The nature of Swagger allows the documentation for each route to be easily identifiable as the API Method (such as GET) is colour-coded and appears next to the route name. In addition, each route has a drop-down menu where the documentation for that specific route can be viewed, improving the overall readability of the Swagger page.

The link to the Swagger documentation can be viewed below:

https://app.swaggerhub.com/apis-docs/SENG3011-qq/CIDRAP_API/1.0.0

Note: Use the 'vivid-apogee' server to get proper responses whilst using the 'Try it out' feature.

2.2 /ARTICLES

The ‘Articles’ route is the main route used for obtaining information related to disease outbreaks from a given source. For each provided news source such as ‘CIDRAP’, an article is considered to be a separate post or webpage (uniquely identifiable by URL) which contains fields such as a headline, author, and body of text.

2.2.1 Input Parameters

The parameters of this route can be seen below:

Parameter	**Example**	**Description**
`period_of_interest_start` (*required*)	`2021-04-13T12:40:00`	Articles posted before this date will not be returned. MUST be of the format ‘yyyy-MM-ddTHH:mm:ss’
`period_of_interest_end` (*required*)	`2023-01-09T12:32:01`	Articles posted after this date will not be returned. MUST be of the format ‘yyyy-MM-ddTHH:mm:ss’
`key_terms` (*required*)	`COVID19,hiv/aids,zika`	The returned articles will be filtered to only be about these diseases. A full list of key terms which are accepted can be seen on the published swagger documentation.
`location` (*required*)	`Iceland`	The returned articles will be filtered to only be about outbreaks within these location(s).
`sources` (*optional*)	`CIDRAP,CDC`	A list of sources to extract articles from. The default value is ‘CIDRAP’ if this parameter is left empty.

These parameters have been chosen to allow users to effectively filter what type of articles they want to be returned. An example of what a complete query would look like using the above input parameters for this route is:

```
/articles?period_of_interest_start=2021-04-
13T12%3A40%3A00&period_of_interest_end=2023-01-
09T12%3A32%3A01&key_terms=COVID19%2Chiv%2Faids%2Czika&location=Iceland&sources=CIDR
AP%2CCDC
```

2.2.2 Success Response

Assuming the query was successful, a '**200 OK**' status will be returned to the user alongside a list of article objects. Each article object contains the following key-value fields:

2.2.2.1 Response Table

Field	Description
`url`	The URL of the article
`date_of_publication`	The date the given article was published
`headline`	The headline of the article
`main_text`	The body of text within the article
`reports`	The list of reports which are found within the article. It is possible this list is empty if there wasn't a report detected in this article (as some articles aren't necessarily about disease outbreaks).
`category`	The category on the news website where the article was published.
`author`	The article's author
`source`	The news site the article was scraped from.

2.2.2.2 Successful Query Examples

The following are examples of real queries of our API.

Example 1: Successfully query COVID-19 reports in China.

```
Request
/articles?period_of_interest_start=2022-03-
13T14%3A00%3A00&period_of_interest_end=2022-03-20T13%3A00%3A00&key_terms=COVID-
19&location=China
```

```
Response
(HTTP Status 200)

[{
  "url":"https://www.cidrap.umn.edu/news-perspective/2022/03/chinas-covid-19-cases-
double-other-nations-eye-resurgences",
  "date_of_publication":"2022-03-14T13:00:00",
  "headline":"China's COVID-19 cases double as other nations eye resurgences",
  "main_text":"A flurry of lockdowns and other strong measures in China are keeping
millions of people at home and workplaces shuttered, amid the country's quickly
rising Omicron surge that saw COVID-19 cases double over the last day…",
  "reports":[{
     "diseases":["COVID-19"],
     "syndromes":["Acute respiratory syndrome","fever","cough","fatigue","shortness
of breath","vomiting","loss of taste","loss of smell"],
     "event_date":"2022-03-14T13:00:00","location":"China"
  }],
  "category":"COVID-19",
  "author":"Lisa Schnirring | News Editor | CIDRAP News",
  "source":"CIDRAP"
}]
```

Example 2: Successful query of measles reports in Brazil from January 13th to April 13th 2022.

The following is also an example of how our reports are not 100% accurate due to language processing limitations.

Request
/articles?period_of_interest_start=2022-01-13T14%3A00%3A00&period_of_interest_end=2022-04-20T13%3A00%3A00&key_terms=measles&location=Brazil

```
Response
(HTTP Status 200)
[{
  "url":"https://www.cidrap.umn.edu/news-perspective/2022/01/global-covid-surge-
slows-quickens-multiple-countries","date_of_publication":"2022-01-25T13:00:00",
  "headline":"Global COVID surge slows but quickens in multiple countries",
  "main_text":"... They said, for example, that Brazil is battling a measles
outbreak and that ongoing diphtheria transmission is occurring in Haiti and the
Dominican Republic. The global COVID totals rose to 361,022,822 cases, and
5,623,004 people have died from their infections, according to the Johns Hopkins
online dashboard.",
  "reports":[{
    "diseases":["diphtheria"],"syndromes":["sore throat","fever","swollen lymph
nodes","weakness"],
    "event_date":"2022-01-25T13:00:00","location":"Brazil"
  },{
    "diseases":["measles"],"syndromes":[fever","cough","runny nose","watery eyes"],
    "event_date":"2022-01-25T13:00:00","location":"Brazil"
  }],
  "category":"COVID-19",
  "author":"Lisa Schnirring | News Editor | CIDRAP News",
  "source":"CIDRAP"
}]
```

2.2.3 Error Codes

2.2.3.1 Table of Codes

Code	**Name**	**Message**	**Sent when…**
400	*Bad Request*	Missing parameter <parameter>	One or more required query parameter is not specified.
400	*Bad Request*	Missing parameter <parameter>	One or more required query parameter is null.
400	*Bad Request*	<parameter> must be a string	One or more query parameter is not of type string when it is required to be. This will most commonly occur if the '`key_terms`' or '`sources`' field are inputted as a list object as opposed to a comma-separated list of values in a single string.
400	*Bad Request*	Invalid timestamp for <parameter>, must be in format 'yyyy-MM-ddTHH:mm:ss'	The start or end date query parameters are not in the format '`yyyy-MM-ddTHH:mm:ss`'.
500	*Internal Server Error*	An internal server error occurred. <error message>	A runtime error occurs on the API server during query processing.

2.2.3.2 Erroneous Queries Examples

The following are real queries and responses from our API server.

Example 1: Parameter `period_of_interest_start` uses an hour value of '25' which is invalid.

Query	`/articles?`**`period_of_interest_start=2022-01-01T25`**`%3A00%3A00&period_of_interest_end=2022-03-20T00%3A00%3A00&key_terms=Wealth&location=Qatar&source=CIDRAP`

Response	`(`*`HTTP Status 400`*`)`
	
	`{`
	`  "status": 400,`
	`  "message": "Invalid timestamp for 'period_of_interest_start',`
	`             must be in format 'yyyy-MM-ddTHH:mm:ss'"`
	`}`

Example 2: Missing query parameter `key_terms`

Query	`/articles?period_of_interest_start=2022-01-01T21%3A00%3A00&period_of_interest_end=2022-03-20T00%3A00%3A00&location=China&source=CIDRAP`

Response	(*HTTP Status 400*)
	
	`{`
	`  "status": 400,`
	`  "message": "Missing parameter key_terms"`
	`}`

Example 3: `locations` as an array

Query	`/articles?period_of_interest_start=2022-01-01T21%3A00%3A00&period_of_interest_end=2022-03-20T00%3A00%3A00&key_terms=COVID-19&`
	`location=Germany&location=China&location=Japan``&source=CIDRAP`

Response	(*HTTP Status 400*)
	
	`{`
	`  "status": 400,`
	`  "message": "key_terms must be a string"`
	`}`

Example 4: Try to input no parameters – the first missing parameter will be detected in the error response.

Query	`/articles`

Response	(*HTTP Status 400*)
	
	`{`
	`  "status": 400,`
	`  "message": "Missing parameter period_of_interest_start"`
	`}`

2.2.4 Design Choices

A lot of the attributes of this route were defined as per the specification. For example, the spec mandates that the period of interest, key terms and location parameters are required. This has the advantage that it is harder for users to accidentally request a large number of articles because they are forced to restrict the search space. This is advantageous for performance and cost because the server does not have to send hundreds of articles and their long main text, reducing bandwidth required. Unfortunately, this design has the disadvantage that users must know what they are looking for in advance. For example, if the user wanted to view COVID cases in any location in the world, they are unable to do this without listing all possible locations which is unreasonable. We were unable to make this change because it was mandated by the spec. For our frontend server, we are considering adding an alternative route or parameter which allows users to see disease reports in all locations.

One design choice we made for parameter checking is that the API allows users to enter an end date which occurs before the start date. In this case, the system will simply return an empty list because there are no reports in this empty time period. We decided this was the best option because it significantly simplifies the error correction logic and reduces the error handling the user has to do on their end. On our frontend server, we would have to add extra error handling logic if users were allowed to enter start and end date. This reduction in system complexity makes the system more maintainable and simpler to understand to both developers and users. This is the same behaviour for all date ranges in our API.

For both the /articles route and the /reports route, we decided to add an optional ‘sources’ input parameter which would aim to return articles or reports from only the listed sources (such as ‘CIDRAP’), or from all sources if this field is left empty. Even though this was not required in the specification, the reasoning behind this decision was to allow our API to be expandable in the future and more flexible in how the API can be used, which will appeal to a larger variety of users. The addition of this feature also allows for a larger quantity of information to be returned to each user, which in most cases will be in their best interest.

2.3 /REPORTS

2.3.1 Input Parameters

The parameters of this route can be seen below:

Parameter	Example	Description
`period_of_interest_start` (*required*)	`2021-04-13T12:40:00`	Reportsbefore this date will not be returned. MUST be of the format 'yyyy-MM-ddTHH:mm:ss'
`period_of_interest_end` (*required*)	`2023-01-09T12:32:01`	Articles posted after this date will not be returned. MUST be of the format 'yyyy-MM-ddTHH:mm:ss'
`key_terms` (*required*)	`COVID-19, hiv/aids,measles`	The returned reports will be filtered to only contain these diseases. A full list of key terms which are accepted can be seen on the published swagger documentation.
`location` (*required*)	`Iceland,USA,China`	The returned reports will be filtered to only be in these location(s).
`sources` (*optional*)	`CIDRAP,CDC`	A list of sources to extract articles from. The default value is 'CIDRAP' if this parameter is left empty.

These parameters have been chosen to allow users to effectively filter what reports they want to be returned. An example of what a complete query would look like using the above input parameters for this route is:

```
/reports?period_of_interest_start=2022-03-
10T00%3A00%3A00&period_of_interest_end=2022-03-21T13%3A00%3A00&key_terms=COVID-
19%2Ctuberculosis%2Cebola%2Cmalaria%2Chiv%2Faids&location=Sweden,North%20Carolina
```

2.3.2 Success Response

Assuming the query was successful, a '**200 OK**' status will be returned to the user alongside a list of report objects. Each report object contains the following key-value fields:

2.3.2.1 Response Table

Field	Description
`diseases`	The list of diseases found in the report.
`syndromes`	The syndromes associated with the found diseases.
`event_date`	The date of the article in which the report was found.
`location`	The location associated with the disease report.
`url`	The URL of the article in which the report was found.

2.3.2.2 Successful Query Examples

The following are examples of real queries of our API.

Example 1: Querying possible reports of COVID-19, Anthrax or Ebola from the start of the year up until the 20th of March for Germany.

```
Request
/reports?period_of_interest_start=2022-01-
01T00%3A00%3A00&period_of_interest_end=2022-03-20T00%3A00%3A00&key_terms=COVID-
19%2Canthrax%2Cebola&location=Germany%2CIndia%2CRussia&source=CIDRAP
```

```
Response
(HTTP Status 200)

[
  {
    "diseases": [
      "COVID-19"
    ],
    "syndromes": [
      "Acute respiratory syndrome",
      "fever",
      "cough",
      "fatigue",
      "shortness of breath",
      "vomiting",
      "loss of taste",
      "loss of smell"
    ],
    "event_date": "2022-01-06T13:00:00",
    "location": "Germany",
    "url": "https://www.cidrap.umn.edu/news-perspective/2022/01/global-covid-19-
case-total-passes-300-million-mark"
  },
  {
    "diseases": [
      "COVID-19"
    ],
    "syndromes": [
      "Acute respiratory syndrome",
      "fever",
      "cough",
      "fatigue",
      "shortness of breath",
      "vomiting",
      "loss of taste",
      "loss of smell"
    ],
    "event_date": "2022-01-04T13:00:00",
    "location": "Germany",
    "article_url": "https://www.cidrap.umn.edu/news-perspective/2022/01/european-
countries-report-post-holiday-covid-surges"
  }
]
```

Example 2: Successful query of polio reports in Nigeria from January 1st to March 20th, 2022.

Request

/reports?period_of_interest_start=2022-01-01T00%3A00%3A00&period_of_interest_end=2022-03-20T00%3A00%3A00&key_terms=polio&location=Nigeria&source=CIDRAP

Response

```
[
  {
    "diseases": [
      "polio"
    ],
    "syndromes": [
      "paralysis"
    ],
    "event_date": "2022-03-17T13:00:00",
    "location": "Nigeria",
    "article_url": "https://www.cidrap.umn.edu/news-perspective/2022/03/new-polio-
cases-africa-malawi-launches-vaccine-campaign"
  }
]
```

2.3.3 Error Codes

2.3.3.1 Table of Codes

Code	Name	Message	Sent when…
400	*Bad Request*	Missing parameter <parameter>	One or more required query parameter is not specified.
400	*Bad Request*	Missing parameter <parameter>	One or more required query parameter is null.
400	*Bad Request*	<parameter> must be a string	One or more query parameter is not of type string when it is required to be. This will most commonly occur if the '`key_terms`' or '`sources`' field are inputted as a list object as opposed to a comma-separated list of values in a single string.
400	*Bad Request*	Invalid timestamp for <parameter>, must be in format 'yyyy-MM-ddTHH:mm:ss'	The start or end date query parameters are not in the format '`yyyy-MM-ddTHH:mm:ss`'.
500	*Internal Server Error*	An internal server error occurred. <error message>	A runtime error occurs on the API server during query processing.

2.3.3.2 *Erroneous Queries Examples*

The following are real queries and responses from our API server. These responses are of the same nature as the /articles route, as the same parameter values are used for both routes.

Example 1: Missing query parameter, misspelt `period_of_interest_start`

Query	/reports?**period_of_interest_starts**=2022-03-13T14%3A00%3A00&period_of_interest_end=2022-03-20T13%3A00%3A00&key_terms=COVID-19&location=China
Response	(*HTTP Status 400*) { "status": 400, "message": "Missing parameter period_of_interest_start" }

Example 2: Missing query parameter `location`

Query	/reports?period_of_interest_start=2022-01-01T21%3A00%3A00&period_of_interest_end=2022-03-20T00%3A00%3A00&key_terms=COVID-19&source=CIDRAP
Response	(*HTTP Status 400*) { "status": 400, "message": "Missing parameter location" }

Example 3: `key_terms` as an array

Query	/reports?period_of_interest_start=2022-01-01T21%3A00%3A00&period_of_interest_end=2022-03-20T00%3A00%3A00&**key_terms=COVID-19&key_terms=ebola&key_terms=yellow%20fever**&location=Germany&source=CIDRAP
Response	(*HTTP Status 400*) { "status": 400, "message": "key_terms must be a string" }

2.3.4 Design Choices

In similar fashion to the /articles route, the parameters chosen for the /reports route were defined as per the specification. All input parameters are technically the same for both routes but the semantics for some input parameters differ slightly. For example, in the /reports route the 'location' parameter filters for reports in a certain location which is a more refined search as opposed to the /articles route, where the 'location' parameter filters for news within a certain location. We decided to keep these parameters having the same name as the overall logic of filtering an article or report by location is still existent.

A benefit of having all query parameters being the same is that users are able to reuse parameter values for both the /articles and /reports routes, depending on if they want to search for articles or reports. This is advantageous from a simplicity point of view for the user as the understanding required to use both routes is lowered. From a developer point of view, consistency of parameter values encourages the reusability of code and documentation (Swagger allows 'components' to be made where a single component such as an input parameter or return value can be referenced to multiple times), improving the overall quality of the codebase. In addition, common tests can be made for both routes at once which will reduce the overall development time and ensure consistency in the behaviour of parameter values for both routes will remain consistent.

One negative of having all query parameters being the same is that users may get confused about which route they are using, or what the difference in purpose between the two routes are. The reason why we decided to make a separate /reports route is that it allows users to view reports without the need for receiving the article information (the main part being the 'main_text' return value in the /articles route). This difference sounds minimal but is useful in allowing users to analyse reports over time without needing to know the details of the article in which the report exists in. Removing the need to return all article information for the /reports route will place less load on the API servers as they will use less bandwidth – the difference of returned information in comparison to the /articles route is clearly evident in the example outputs given in sections 2.2.2.2 and 2.3.2.2.

2.3.5 Changes since Deliverable 1

Slight changes have been made for the /reports route based on the submitted Deliverable 1 Report. The optional 'diseases' and 'syndromes' input parameters have been removed since their purpose was essentially the same as the required 'key_terms' input parameter. The idea behind these two parameters was to filter reports returned to contain only the diseases and syndromes which were inputted by the user. However, we felt in this case the 'key_terms' parameter would have no use as filtering reports (or articles) by general key terms isn't the aim of our project, so there wouldn't be anything meaningful to input into that parameter. Therefore, the 'key_terms' parameter is now being used to input diseases for which returned reports must be about.

Furthermore, the 'syndromes' parameter was removed as a whole since we found that reports didn't mention syndromes relating to the certain disease the report was about. We adjusted to this situation and still adhered to the requirement of including syndromes by researching the syndromes of each provided disease and returning that list of syndromes within the report object corresponding to the disease of the given report. We also added more generic syndromes such as 'fever', 'cough' and 'loss of smell' to the list of possible syndromes so reports were more informative about each disease.

2.4 /PREDICTIONS

The COVID-19 pandemic highlights the importance of epidemic prediction in the highly connected modern world. For an extra feature, we decided it would be useful to provide epidemic prediction functionality to our API through the `/predicitons` route. An epidemic is predicted when there are a certain number of reports of a particular disease in a specified period of time. The route then returns these reports as a list, described below.

2.4.1 Input Parameters

The parameters of this route can be seen below:

Parameter	Example	Description
`min_report_count` (*required*)	8	The minimum number of reports required for a disease to be included in the prediction
`day_count` (*required*)	10	The number of days in the past from the present day to include in the prediction

These parameters have been chosen to allow users to effectively define what classes as an outbreak prediction. The period of interest is defined by the day count, the number of days in the past from the present day to include in the prediction. The minimum report count defines how many reports of a particular disease are required in the specified time period to be classified as a prediction.

An example of what a complete query would look like using the above input parameters for this route is:

```
/predictions?min_report_count=8&day_count=10
```

2.4.2 Success Response

Example: COVID-19 detected as a possible epidemic with 9 reports in the past 10 days.

```
Request
/predictions?min_report_count=8&day_count=10
```

```
Response
[
  {
    "disease": "COVID-19",
    "reports": [
      {
        "report_id": 5,
        "event_date": "2022-03-15T13:00:00",
        "disease_id": "COVID-19",
        "article_url": "https://www.cidrap.umn.edu/news-perspective/2022/03/twice-
many-black-covid-patients-deemed-lowest-priority-icu-triage-system",
        "location": "Boston"
      },
      ... 8 more reports ...
    ],
    "report_count": 9
  }
]
```

2.4.3 Error Codes

2.4.3.1 Table of Codes

Code	Name	Message	Sent when…
400	*Bad Request*	Missing parameter <parameter>	One or more required query parameter is not specified.
400	*Bad Request*	Missing parameter <parameter>	One or more required query parameter is null.
400	*Bad Request*	<parameter> must be an integer	`min_report_count` or `day_count` is not an integer
400	*Bad Request*	Parameter `'min_report_count'` must be greater than 0.	`'min_report_count'` is less than or equal to 0.
400	*Bad Request*	Parameter `'day_count'` must be greater than 0.	`'day_count'` is less or equal to 0.
500	*Internal Server Error*	An internal server error occurred. <error message>	A runtime error occurs on the API server during query processing.

2.4.3.2 Erroneous Queries Examples

The following are real queries and responses from our API server. These responses are of the same nature as the /articles route, as the same parameter values are used for both routes.

Example 1: Misspelt minimum reports parameter.

Query	`/predictions?min_report_counts=4&day_count=1`

Response	(*HTTP Status 400*)
	
	`{`
	`  "status": 400,`
	`  "message": " Missing parameter min_report_count"`
	`}`

Example 2: Invalid integer for minimum report count.

Query	`/predictions?min_report_count=4a3&day_count=1`

Response	(*HTTP Status 400*)
	
	`{`
	`  "status": 400,`
	`  "message": "min_report_count must be an integer"`
	`}`

Example 3: Minimum report count of 0.

Query	`/predictions?min_report_count=0&day_count=3`

Response	*(HTTP Status 400)*
	
	`{`
	`  "status": 400,`
	`  "message": "Parameter 'min_report_count' must be greater than 0."`
	`}`

Example 4: Day count of 0.

Query	`/predictions?min_report_count=4&day_count=0`

Response	*(HTTP Status 400)*
	
	`{`
	`  "status": 400,`
	`  "message": "Parameter ‘day_count’ must be greater than 0."`
	`}`

2.4.4 Design Choices

This route implements a rather simple definition of an epidemic prediction by counting disease reports in a specified period of time. The first and main reason we settled on the simpler design was because we wanted to make our system transparent. If we had implemented a complex algorithm which statistically analysed the frequency of cases, the user would have a significantly lower understanding of the data they actually receive. A highly complex system would appear as a black box to all but the most experienced users who take the time to review the algorithm. We felt that users will benefit more from a simple but useful prediction system which they can understand and tune to their use case. A complex, probability-based system sacrifices its usability and flexibility for prediction accuracy.

The second reason we decided on a simple solution is because it was the most feasible prediction system for us to implement before the deadline. If we had tried to implement a complex frequency-based algorithm, we would be more likely to make a mistake and produce incorrect predictions. This is a lot harder to test for with this route because epidemics do not happen very often. We would have to create sample data for a possible epidemic which may not even be accurate to the real world.

2.5 /LOGS

This route returns a list of all the previous requests to the API which have been made. This route is useful because it allows developers and users to debug requests and analyse traffic to the API.

The requirements given by the spec outlined that we had 2 options as to how to keep track of logs. One of these options was to keep a log file in the backend, however we opted against using this option since our API is serverless, which means a log file could not be stored on disk. Instead, we opted to store all logs persistently in the PostgreSQL database, accessible through the /logs route (this corresponded to the other option given by the spec). The logs route allows users to view and filter usage logs of our API.

A major advantage of this method is that users can filter results by period, route, status, team and IP. This is pretty much impossible with a simple log file without using external programs.

2.5.1 Input Parameters

The parameters of this route can be seen below:

Parameter	Example	Description
`period_of_interest_start` (optional)	`2022-03-18T23:51:56`	The earliest time for which logs generated at this time will be returned. Must be in the format 'yyyy-MM-ddTHH:mm:ss'
`period_of_interest_end` (optional)	`2022-03-20T00:25:53`	The latest time for which logs generated at this time will be returned. Must be in the format 'yyyy-MM-ddTHH:mm:ss'
`routes (optional)`	`/articles,/reports`	A comma separated list of routes to filter by. All routes begin with a forward slash.
`status (optional)`	`400`	The status to filter by. If unspecified, all statuses are included.
`team (optional)`	`Team QQ`	The team name to filter by. If unspecified, all teams are included.
`ip (optional)`	`2001:8003:360d:2100:5970:7009:7201:886e, 169.254.1.1`	The IP address to filter by. If unspecified, all IP addresses are included.

These parameters have been chosen to allow users to effectively filter which logs they want to be returned. This will be very useful in allowing teams to debug specific requests to our API and when analysing traffic.

An example of what a complete query would look like using the above input parameters for this route is:

```
/logs?period_of_interest_start=2022-03-18T23%3A51%3A56&period_of_interest_end=2022-03-
20T00%3A25%3A53&routes=%2Farticles%2C%2Freports&status=400&team=Team%20QQ&ip=2001%
3A8003%3A360d%3A2100%3A5970%3A7009%3A7201%3A886e%2C%20169.254.1.1
```

2.5.2 Success Response

The following are examples of real queries of our API.

Example 1: Querying all logs which have used the /predictions route on March 21st, 2022, between 12:47am and 1:00am:

```
Request
/logs?period_of_interest_start=2022-03-21T00%3A47%3A00&period_of_interest_end=2022-
03-21T01%3A00%3A00&routes=%2Fpredictions&status=200
```

```
Response
[
  {
    "id": 729,
    "status": 200,
    "route": "/predictions",
    "req_params": "{\"min_report_count\":\"1\",\"day_count\":\"5\"}",
    "timestamp": "2022-03-21T00:49:51",
    "message": "success",
    "ip": "::ffff:127.0.0.1",
    "team": "Team QQ"
  },
  {
    "id": 722,
    "status": 200,
    "route": "/predictions",
    "req_params": "{\"min_report_count\":\"1\",\"day_count\":\"5\"}",
    "timestamp": "2022-03-21T00:47:41",
    "message": "success",
    "ip": "::ffff:127.0.0.1",
    "team": "Team QQ"
  }
]
```

Example 2: Querying all logs from a time in the future (should return none):

```
Request
/logs?period_of_interest_start=2023-01-01T00%3A00%3A00
```

```
Response
[]
```

2.5.3 Error Codes

Code	Name	Message	Sent when…
400	*Bad Request*	<parameter> must be a string	One or more query parameter is not of type string when it is required to be.
400	*Bad Request*	Invalid timestamp for <parameter>, must be in format 'yyyy-MM-ddTHH:mm:ss'	The start or end date query parameters are not in the format '`yyyy-MM-ddTHH:mm:ss`'.
400	*Bad Request*	Invalid route list	Any of the routes within the routes list parameter do not start with a forward slash, or any other invalid character is used

400	*Bad Request*	status must be an integer	The status parameter is not an integer
500	*Internal Server Error*	An internal server error occurred. <error message>	A runtime error occurs on the API server during query processing.

2.5.3.1 Erroneous Queries Examples

Example 1: Using an invalid timestamp for period_of_interest_start.

Query	`/logs?period_of_interest_start=2022-03-21T23%3A51-56`
Response	`{ "status": 400, "message": "Invalid timestamp for 'period_of_interest_start', must be in format 'yyyy-MM-ddTHH:mm:ss'" }`

Example 2: Supplying a routes list where one of the routes does not start with a forward slash.

Query	`/logs?period_of_interest_start=2022-03-21T23%3A51-56`
Response	`{ "status": 400, "message": "invalid route list" }`

Example 3: Supplying a status parameter which is not an integer.

Query	`/logs?status=h`
Response	`{ "status":400, "message":"status must be an integer" }`

2.5.4 Design Choices

A major advantage of our choice to use a route to display logs is that it allows us (and other teams) to filter logs by period, route, status, team and IP. Considering the very high number of logs, this feature will come in very handy when we are troubleshooting in the future, it will mean we won't need to search through a log file of thousands of logs to find what we are looking for.

In terms of the input parameters that we have chosen to use, we believe that each of them have use cases which make them highly important and useful. None of the parameters are compulsory for the logs route, which simply means if no parameters are supplied then all the logs are returned. This could be useful for a team who wishes to look at all past logs. However, this is quite unlikely because often teams will want to filter the logs so that they can only see the ones which are relevant to what they are looking for. The period of interest start and end parameters would be useful for teams when they are trying to check the http requests to our API that happened at a specific point in time, for example if a team's system was functioning correctly yesterday but not today, they might use those parameters to select the logs from only yesterday so that they can retrace their steps. The routes parameter could be useful for teams trying to see their logs using a specific route, e.g. a team wants to see when they most recently used the /articles route. The status option allows teams to filter by the response status of their queries, for example a team might want to see all the times that their queries have been unsuccessful due to internal server errors. Finally, the team and ip parameters allow teams to filter out other teams or people, so that they can see the logs which are actually relevant to them rather than other teams or people. For example, our team might be trying to debug our system for deliverable 3, and we want to see our own logs without being distracted by the logs of other groups who have been using our API.

3 Shortcomings and Challenges Addressed

3.1 Hard-Coded Symptom List

One of the compromises we had to make was using a hard-coded symptom list. The first reason we chose to do this because it simplified the scraping process. This is because the scraper only needs to detect diseases and their associated location and doesn't need to try and find nearby symptoms. We noticed when analysing the content of CIDRAP articles that it is very rare that there is a clear set of disease, symptoms, and location. The second, main reason we chose to hard code the symptom list is because detecting and including symptoms would add a lot of inaccurate noise to the reports. Most symptom reports would actually be incorrect and would likely often occur when there just happens to be a symptom word like "headache" near a location name.

The main downside of this decision is that it makes the system worse at detecting new outbreaks. Since we believe that the symptom reports would be inaccurate anyway, there would be little practical benefit to these outbreak predictions.

3.2 Switching to AWS for Database Hosting

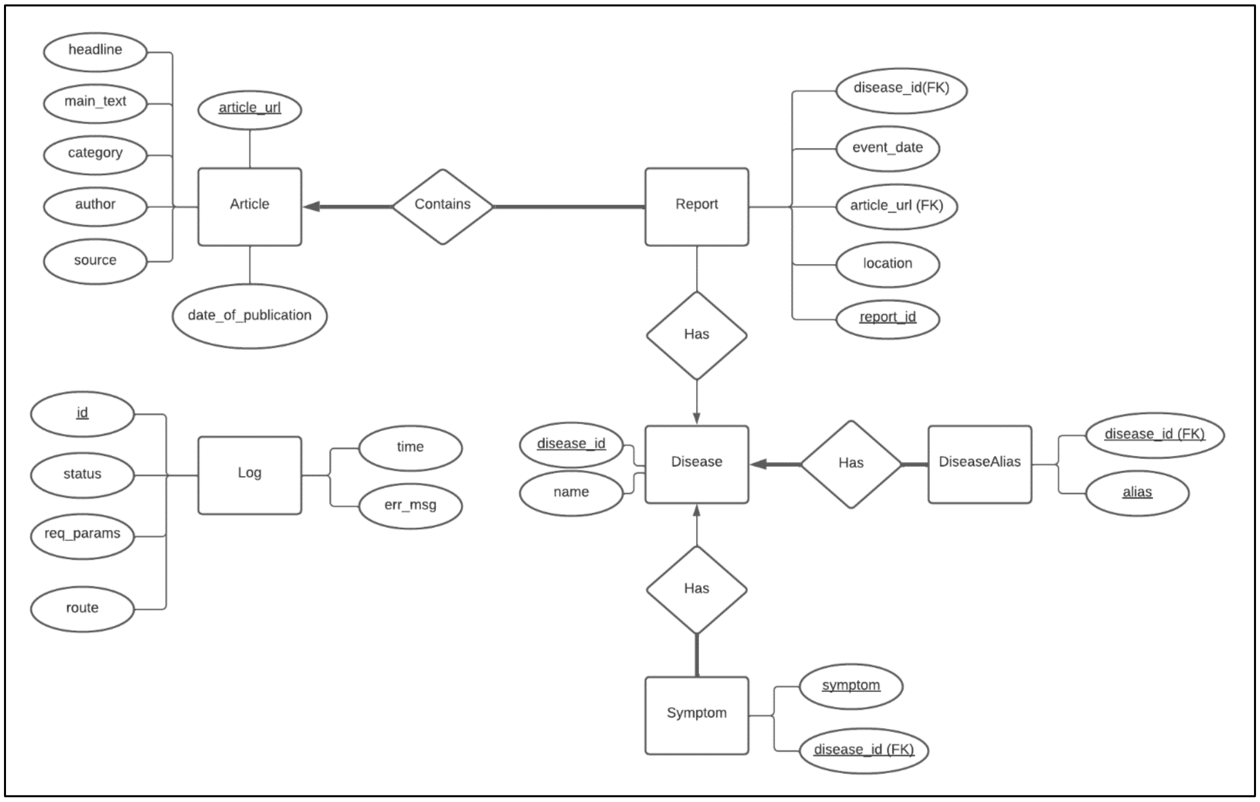
As described earlier in the report, one other compromise we made from deliverable 1 is that we switched from Google Cloud SQL to Amazon Relational Database Service (RDS). We did not want to make this change; however, we were forced to when our Google Cloud free trial ran out due to the database consuming resources. We decided to switch to AWS RDS because it offered a somewhat equivalent database hosting service with a much better free plan. We kept all other functionality hosted on Google Cloud because we had already become familiar with its workflow. The only downside of switch is that we are no longer using the same cloud platform for all of our systems which means we have to manage more accounts and billing.

3.3 Swagger Hub Free Trial

We decided to use Swagger-Hub to host the Swagger documentation for the project. Swagger-Hub is useful because it has many collaboration features and it automatically generate the docs as you write them. It also has hosting functionality and a useful "try-it-out" feature which allows users to easily make calls to the API. The main downside of Swagger-Hub is that it is not free for teams, and we are using a free trial which runs out on Thursday Week 6. We decided to keep using Swagger-Hub for deliverable 2 because the try-it-out feature is required for marking. Once the deliverable is over, we will save a copy of the source YAML, generate a static version of the docs and host them on Google Cloud. This has the disadvantage that you cannot use the 'try-it-out' button.

4 Appendix

4.1 Updated Database Entity Relationship Diagram



4.2 Database Schema

The database schema below is defined in "`reset-schema.js`" which can be found with the scraper source code. Underlined attributes are primary keys. Italic attributes are foreign keys.

4.2.1 Table "Article"

Attribute	Type	Required	Description
`article_url`	string	yes	URL of article, primary key of each record.
`headline`	string	yes	Headline of article.
`main_text`	string	yes	Main body of article.
`category`	string	no	Category of the article.
`source`	String	yes	Describes the source of the article, i.e. CIDRAP.
`author`	string	yes	Author of the article. If there are multiple authors then they will be in a comma separated list.
`date_of_publication`	date	yes	Date the article was published.

4.2.2 Table "Disease"

Attribute	Type	Required	Description
disease_id	string	yes	The unique identifier for the disease, used to query diseases.
name	string	yes	A human readable name for the disease.

4.2.3 Table "Symptom"

Attribute	Type	Required	Description
symptom	string	yes	The name of the symptom.
disease_id	string	yes	The ID of the disease it is the symptom of.

4.2.4 Table "DiseaseAlias"

Attribute	Type	Required	Description
disease_id	string	yes	The ID of the disease with this alias.
alias	string	yes	The disease alias.

4.2.5 Table "Report"

Attribute	Type	Required	Description
report_id	integer	yes	Unique auto incrementing ID assigned to each report.
disease_id	string	yes	The disease which the report is about.
event_date	date	yes	The date the article with the report was published.
article_url	string	yes	This is a foreign key, which describes the article this report belongs to.
location	string	yes	The location which was found with the disease report.

4.2.6 Table "Log"

Attribute	Type	Required	Description
id	integer	yes	The ID of the disease with this alias.
status	integer	yes	The status code the API server responded with.
route	string	yes	The route which was queried by the user.
req_params	string	yes	The query parameters sent by the user.
timestamp	string	yes	The time the log was created.
message	string	no	The success/error message sent by the query. On success, message is "success", otherwise it is the error message.
ip	string	no	The IP address of the requesting user.
team	string	no	The name of the team which accessed the route. This can be specified by adding "team" to a request's query parameters.

5 References

Deliverable 1 Report - SENG3011 22T1, Team QQ

(*Referenced in the report wherever “Deliverable 1” is mentioned*)