We Byte- Deliverable 2

Note: Data source is promedmail.org

1.1 API Design Details

Describe final architecture of your API, justify the choice of implementation, challenges addressed and shortcomings, in your Design Details report in Github (or equivalent). You may update management information if necessary.

1.1.1 Final API Architecture

jamesbradleysyd@gmail.com → could use the same diagram u made in D1 but with it slightly modified and postgres instead of dynamo?
Didn't see this so I made one down below in 1.3.3 - Lachlan

1.1.2 Implementation Justification

Implementation Language: Python

	Python	**PHP**
Team familiarity	5/5	0/5
Web Frameworks	Many web frameworks	Access to mature frameworks
Speed	Interpreted language, slower	Faster than Python
Open Platform	Designed with features/libraries to facilitate data analysis and visualisation	Platform independent
Web Service Libraries	Easy integration with flask and/or django	Large array of libraries available
Industry Relevance	Industry Leader, many deployment options	Open source, easy development
DataBase	Doesn’t support database connectivity as much as PHP but RDBMS like PostgreSQL, MySQL, SQlite are still accessible	Able to access more than 20 different databases
Dependency management	Pipenv	Php composer
Readability	Indentation requirements makes it makes it very readable	Highly documented- classic approach
Debugging	Faster	Slow
Usability	Very versatile - able to use it for web development, robotics, science etc	Mainly used for web development

Python has the greatest appeal for developing an API given its flexible nature, ability to integrate seamlessly with many web frameworks, cross-platform design and simple integration into the rest of the chosen web stack. Additionally, all team members are highly familiar with Python, making it the best language to use to ensure our team is easily able to collaborate and develop across multiple platforms. Finally, the abundance of modules and libraries that come alongside Python means it has added versatility and will aid with adding any additional features. Despite the advantages that come along with using PHP and other backend programming languages like access to mature frameworks, speed and community support, the teams' familiarity with Python and simultaneously its simple integration with Flask means it would be the best suited language for our project.

During the implementation of the API, Python has proved to be the right choice and has served as the most optimal language for this task.

Platform Provider: Amazon Web Services

There are many cloud platforms to choose from, however, AWS was chosen over the others due to the reasons listed below:

- **Prevalence in industry**: it is by far the most used singular cloud service. With a market share of 33%, with its next biggest competitor Azure at 13%.
- **Large ecosystem**: of over 200 cloud products. Making it easy to deliver more complex microservices.
- **It's free tier:** offers an abundance of database storage and compute time. More than enough for this project
- **It is very heavily documented:** and has many tutorials, which will help make it easier to learn

Serverless Platform: AWS Lambda & AWS Gateway

We decided to implement our API with a serverless architecture. This meant we would be using AWS Lambda and API Gateway, although to abstract the configuration on AWS and ease the development and deployment processes we decided to use the **Serverless framework**.The reasoning for this is listed below:

- **Easily scalable**: We expect that the usage of our API will have long periods of inactivity then spikes of high activity (such as demo day). Using Lambda provides a lot of horizontal scaling, it defaults at 1000 concurrent requests.
- **Cost effective**: It only gets run when it is triggered, this saves on a lot of expense compared to using a server model in which the server must have long periods of uptime. The free tier of Lambda includes 1 million requests and 400 000 GB-seconds of compute time a month, as long as each function is limited to a compute time of 5 minutes. This plenty for the scope of this project
- **Simple**: The Serverless framework has an extremely easy set up. Especially in comparison to ECS, which has significantly more configuration. By using this framework,

the development and deployment environments can be simply handled and we can focus on writing code.

- **Flask Compatibility**: Flask is a web framework our team is extremely familiar with, it is therefore desirable that our platform is compatible with it. AWS Lambda is not compatible with Flask without complex configuration, but by using the Serverless framework we can easily use Flask.
- **Modular**: Each Lambda function is designed to serve a singular purpose. This allows us to design a much more decoupled design, making it much more simpler and more manageable. As opposed to an entire EC2 container which would run many different aspects of the API, and therefore a simple bug could compromise the entire API more easily.

Database: AWS RDS- Postgres

	AWS RDS- Postgres	**DynamoDB**
DB Type	Open source relational database systems	Fast and flexible NoSQL database with seamless scalability.
Schema	Required- to adhere to relational database structure	Requires no Schema
Keys	Primary and Foreign Keys (Link to other tables)	Primary and Sorting Keys
Querying & DB manipulation	Simple query syntax & ease of extraction of data	- Query: Only used for primary and sorting keys, requires a primary key to be set to a value - Scan: Expensive and only used for attributes
Community	Large community, and abundance of documentation and help resources	Released in 2012- small community
AWS Free Tier	- 20GB General Purpose Storage - 20GB Database backups and DB snapshots - 750 hrs/month of micro database usage	- 25Gb of Free Storage - Enough to handle up to 200M requests/month
Linking Tables	Use of foreign keys to link multiple tables	Linking of tables is complex and/or not feasible
Ordering/Sorting Tables	ORDER BY & GROUP BY query syntax	Potentially with GSI but complex and expensive method
Ease of Use	Large extraction of data and filtering of table is made simple through large array of queries possible	Steeper learning curve but becomes simpler once you become familiar with the DB

AWS Setup	Complex initial setup of DB instance and interaction of AWS Lambda is similar to any python script interacting with a PostgreSQL DB	Simple setup and interaction with AWS Lambda is documented and simple
JSON	JSON format not applicable with RDMS structure	JSON acceptable

Despite the easy integration of DynamoDB with Lambda and the simple AWS setup, through research and testing the database it became apparent that it was expensive to make queries and extract information. Given the requirement for our API to be efficient and scalable as it would potentially be used by other teams DynamoDB proved not to be the right choice. Ultimately, the team's familiarity with SQL Syntax and the ease of extraction of information from a database meant PostgreSQL was the most suitable choice.

WebScraping & Libraries

- **Beautiful Soup & Urllib**

In order to aid with the web scraping functionality, Python's Urllib and Beautiful Soup libraries will be used. Urllib allows us to open URLs in order to process them. Beautiful Soup parses the HTML from opened URLs and allows us to extract relevant information.

- **Selenium**

Selenium is an open-source automation tool that has easy integration with Python and Beautiful Soup and allows for the automation of browsers which aided with our webscraper. Additionally it is much easier to use and has the potential to send commands to a multitude of different browsers.

- **NLP- spaCy**

spaCy is an open sourced natural language processor which can be used to pull locations out of text, aiding with the process of taking in location as a parameter. It can also be used later to distinguish between different reports within articles.

- **Psycopg2**

Psycopg2 was utilised in the linking of the PostgreSQL database and the AWS lambda scripts. It was useful in that it automatically converts results from queries into a python list allowing for easy interaction and extraction of data.Additionally, it is commonly used for multi-threaded applications and has its own connectivity allowing for efficiency.

Other libraries used in our API, lambda functions and webscraper were Boto3, Json, Requests and DateUtil all which served their own purpose in increasing the functionality of our API and allowing for easy integration across all platforms and/or frameworks and adherence to the specification requirements.

1.1.3 Challenges and Shortcomings

Database: SQL vs NoSQL

Originally we tried to use a DynamoDB database, due to its efficiency and ease of use. However, as we developed our API endpoints, it became more difficult to implement the functionality we wanted with the key-value model used by DynamoDB. Our team's relatively limited knowledge of no-sql compared to sql meant continuing to use Dynamo was going to be very challenging, and we decided to switch to postgresql, which we were more familiar with.

AWS Lambda

While porting our web scraper into AWS Lambda, we hit issues due to the size of the libraries required for our scraper and NLP engine spacy. Lambda imposes a file size limitation of 250mb including source code and all required libraries. We managed to solve this by separating the code that uses the web scraping libraries, and the code that uses spacy to do reports processing into two separate Lambda functions.

GeoNames API

We decided to use GeoNames to lookup country information due to it being free and relatively easy to set up. However, GeoNames has a limit of 20,000 requests per day, and during our initial scrape, we hit that limit quickly meaning much of our location data is missing country information even when it should be clear what country it is from. To mitigate this, we will slowly update the locations table in the database to perform new look-ups of the country where it is currently missing, as our request limit allows. Another option was to switch to Google Places API, however this would have been expensive due to pricing at around $32USD per thousand requests.

1.2 Testing

In Testing Documentation report, describe the testing processes used in the development of API, referring to the data and scripts included in Phase_1 folder. This should describe your testing environment and/or tools used, and limitations (e.g. things that are not tested). Describe your testing process i.e. how your team conducts testing using the test data (e.g. in which order) and an overview of test cases, testing data and testing results. Describe the output of testing and what actions you took to improve the test results

1.2.1 Testing Processes (Data and scripts used)

Testing Environment

The main testing pipeline included 4 shell scripts (one for each checker/check in the flow diagram). Each shell script had an accompanying python script which was used to verify the JSON responses. Each response was checked for a generated report and valid locations. Verified data was then cross examined with the stored data to check that the API requests are being successfully handled and that the data was being stored correctly.

We also utilised the following Python testing modules to ensure our API is robust:

- pytest for unit testing.
- pytest-cov library to ensure maximum code coverage
- requests module for automated testing of our endpoints
- Swagger and AWS "Test" feature for manual verification and debugging during development

Limitations

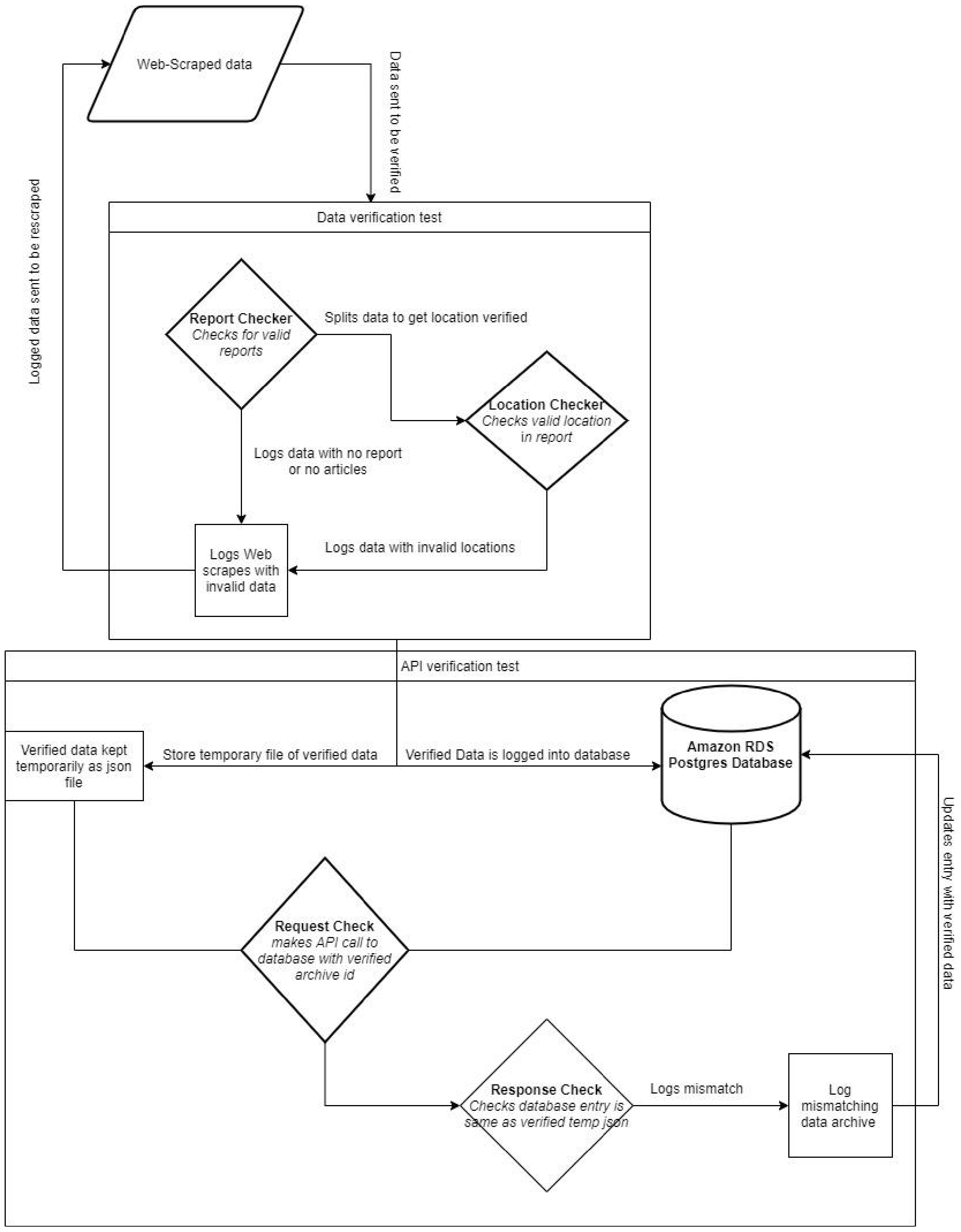
- The output from the NLP Spacy module is not verified. We are relying that Spacy can extract the report data correctly.
- Currency the testing pipeline has only been set up in unix as is not cross compatible with other OS systems. Therefore testing has to be conducted by specific individuals on demand.
- Load testing hasn't been considered. We are relying that AWS automatically handles that.
- Only archive ID queries are being tested at the moment, other queries such as location, date, keywords or latest articles are not being tested.

1.2.2 Testing Process (How testing was conducted)

The testing process was broken into 2 parts:

- **Data Verification:** Data that has just been scrapped undergoes a verification process whereby the existence of the generated reports are checked (as sometimes they are not created due to multi threading issues in the scrapper) and the locations in those reports are valid as we have a limited amount of request with the GEO encode API. If not, that data is logged and sent to be rescraped.
- **API request verification:**.The verified data is then checked to ensure it has been added to the databases correctly. It also checks that the API is still running and that queries can still be made.

Below is a flow diagram of how it proceeded:


Web-Scraped data
Data sent to be verified
Logged data sent to be rescraped
Data verification test
Report Checker
Checks for valid reports
Splits data to get location verified
Location Checker
Checks valid location in report
Logs data with no report or no articles
Logs data with invalid locations
Logs Web scrapes with invalid data
API verification test
Verified data kept temporarily as json file
Store temporary file of verified data
Verified Data is logged into database
Amazon RDS Postgres Database
Updates entry with verified data
Request Check
makes API call to database with verified archive id
Response Check
Checks database entry is same as verified temp json
Logs mismatch
Log mismatching data archive

1.2.3 Testing Output

Data Verification
The testing demonstrated that there were a lot of archives that weren't being scraped properly. The main text article were never being analysed, thus the reports never being generated. This was linked to an issue with multi threading the web scraper, and is currently being debugged to try and amend this issue. Currently 29/78 scraped diseases have some archives (can vary from 5 to approx 50) that don't generate reports. There are also 15/78 diseases in which no archives are found.

The Location checker also showed how we underestimated how many requests we make to the GEO encode API, already maxing out the free 20 000 monthly requests.

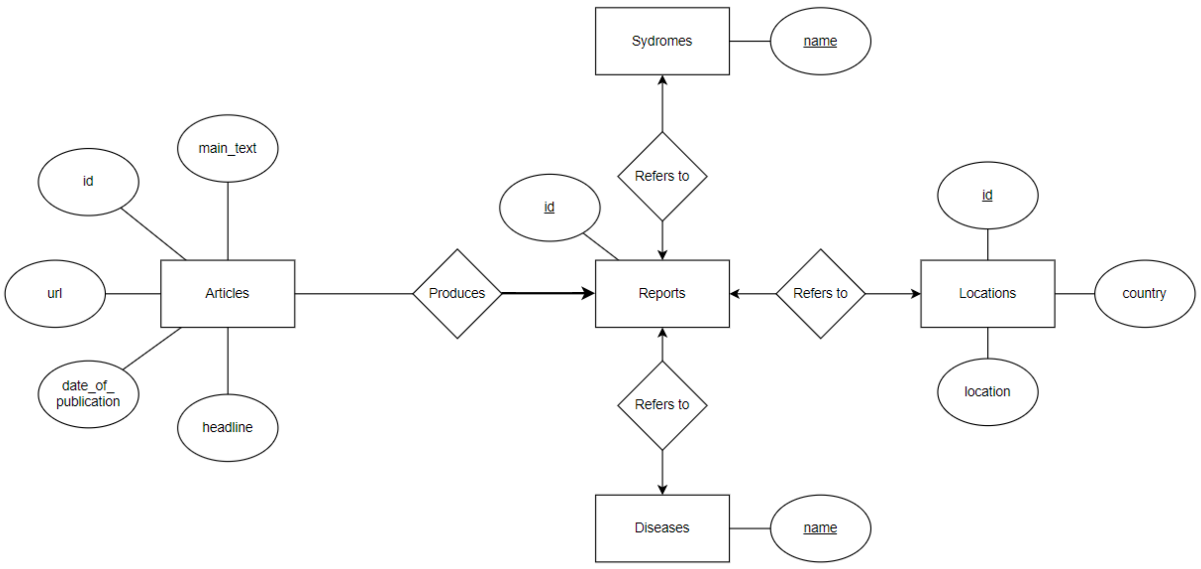
API Request Verification
So far no data mismatching archives have been logged, giving us confidence that the verified data is being inserted correctly, as well as, the API gateway being fully functional for archive queries .

1.3 Documentation

1.3.1 SwaggerHub API Link
https://app.swaggerhub.com/apis/Seng-We-Byte/We-Byte/1.0.0#/

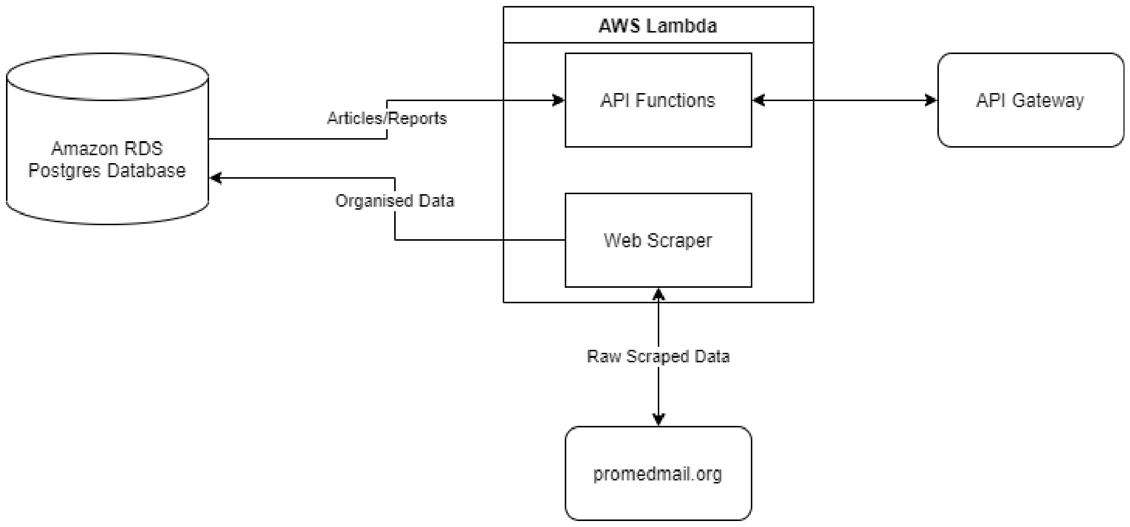
1.3.2 Database ER Diagram

Draw.io (Edit link)



1.3.3 Architecture Diagram

Draw.io (Edit link)



1.4 Project Plan

In order to best articulate team member responsibilities and plan out any deliverables and/or sprints, Trello, Teams(and in built Microsoft OneNote) and TeamGantt were used as all platforms allow for easy real time collaboration. Trello in particular was used for the management of sprints and the breakdown of tasks amongst team members, whilst TeamGantt allowed for the careful planning and consideration of all project requirements .

A link to our Trello board is as follows: Trello Board

.

A link to our Gantt Chart is as follows: https://bit.ly/3bYwfdW