Testing Documentation

Performance, correctness and security are imperative in designing a high quality and trusted API. Thus these areas have been thoroughly tested to ensure the usefulness & reliability of our API. Figure 1.1 illustrates all the testing and tools used to undergo these tests.

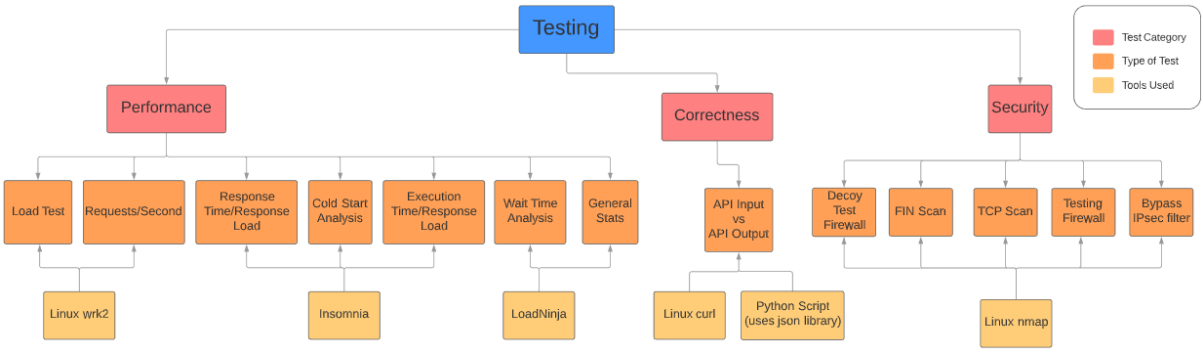


Figure 1.1: Comprehensive Summary of Testing

All of the testing conducted was done on using the actual API that is hosted on Google Cloud since we wanted to examine real numbers that can be expected from everyday usage. Every test script used to run a non-browser or application specific tool can be found in the ‘TestScripts’ directory of our GitHub repository with their corresponding results in a text file. Additionally when possible the testing environment used was set to match the Google Cloud environment, i.e. requests sent from Sydney or Australia.

Performance

Performance is critical in designing a useful API as high latency will only deter developers and users from using our API. Thus a series of tests have been conducted in order to evaluate the speed and efficiency of our API. Firstly the current performance of the API will be examined, then it's major performance hindrance will be analysed and then notable performance discoveries will be mentioned.

Overall API Performance

General Statistics

Firstly to understand how the API should typically perform, a raw set of statistics were gathered for a range of requests over a wide spectrum. The tool LoadNinja was used as it provided the most detailed statistics and analysis which is applicable to gain an overall insight into the performance of the API. The statistics from the LoadNinja tests are detailed below in Figure 1.2.

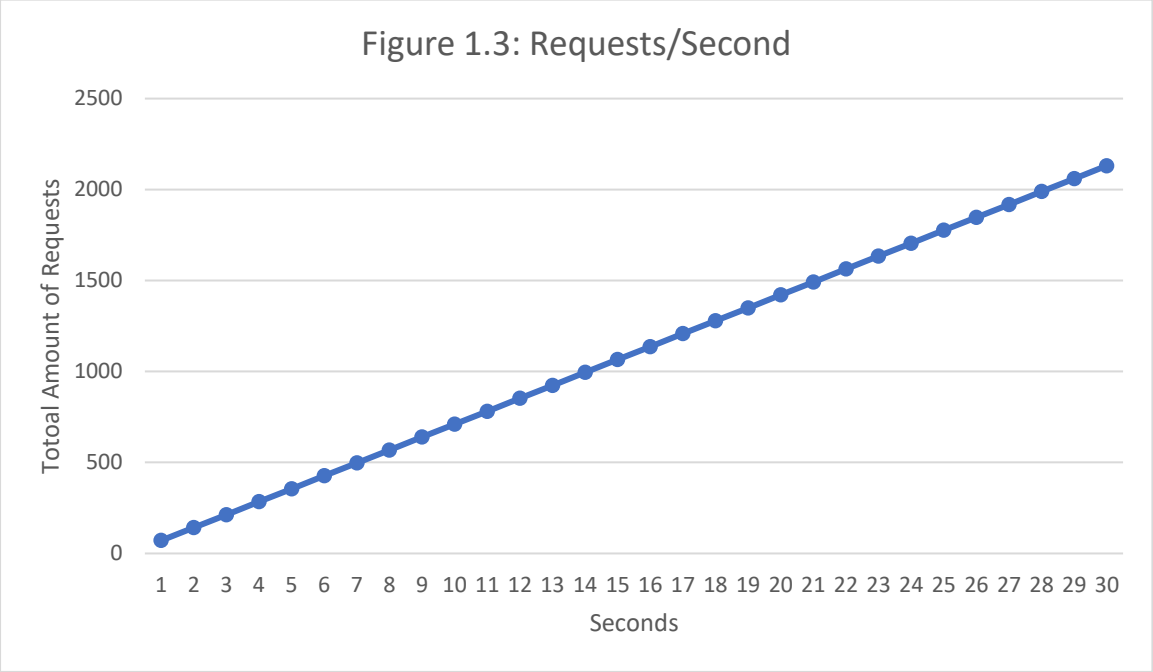
No. Reports Requested	90th Percentile Response Time	95th Percentile Response Time	Average Response Time	Minimum Response Time	Maximum Response Time	Standard Deviation
5	5830 ms	5980 ms	3470 ms	1610 ms	6010 ms	1840 ms
15	6120 ms	6540 ms	5020 ms	1670 ms	6700 ms	2900 ms
25	6200 ms	6780 ms	5350 ms	1800 ms	7080 ms	1860 ms
35	6100 ms	6300 ms	4700 ms	1620 ms	6500 ms	2240 ms
45	6150 ms	6620 ms	4560 ms	1690 ms	6820 ms	2990 ms
55	5890 ms	6370 ms	6240 ms	1830 ms	6400 ms	3080 ms
65	6950 ms	7240 ms	4550 ms	1600 ms	7420 ms	3310 ms
75	6840 ms	7500 ms	5670 ms	1660 ms	7680 ms	3830 ms
85	8470 ms	8960 ms	5720 ms	1690 ms	9090 ms	3110 ms
100	8380 ms	9490 ms	6390 ms	1790 ms	10490 ms	1380 ms

Figure 1.2: LoadNinja API Performance

Analysing figure 1.2, we can identify that for requests that will return approximately 5 reports, we can expect the API to perform the quickest while requests above this size (up to 100 reports) will take 5 to 6 seconds. Although any of these requests can take up 2 seconds if the search parameters provided exactly match the reports/articles queried from the Firestore database. This reassures our group that the API's best case scenario has been optimised. A solution to reduce these average response times is explored later, although we did not implement this as it limits the capabilities of our API.

Load Testing

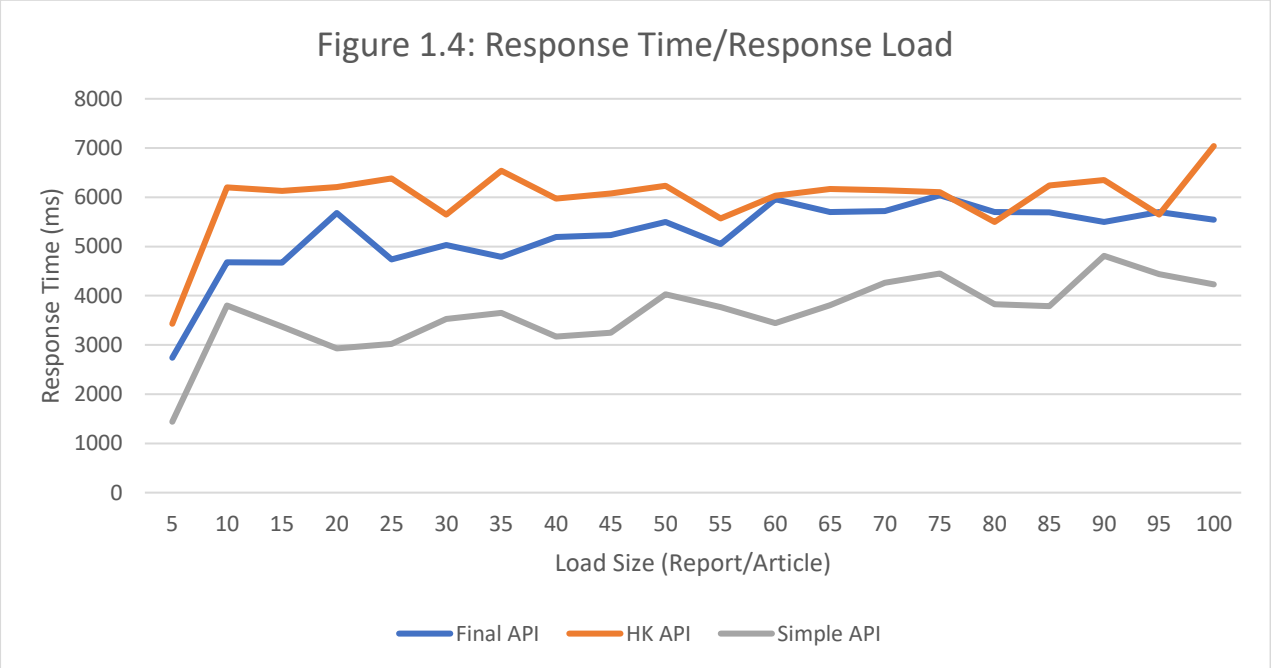
Now that we have a comprehensive understanding of how efficiently the API operates, we will use the linux tool wrk2, to thoroughly test the limits and capabilities of our API. wrk2 is a benchmarking tool that simulates several connections which sends constants throughput loads to a website. This is perfect for load testing as each of these connections represent a user. The test was ran using 2 threads (since 2 ports are open), with 2000 requests constantly being sent over 100 connections over 30 seconds. This load test revealed that our API is capable of 71.03 requests per second as shown in Figure 1.3.

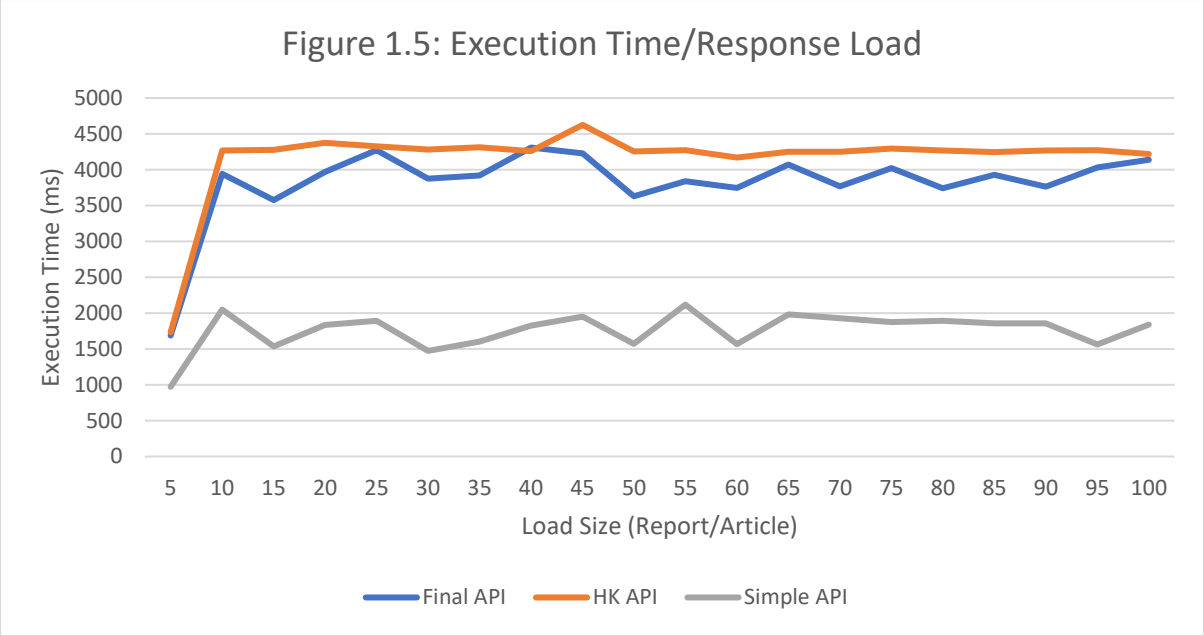


Specifically our API can handle approximately 2135 requests over 30 seconds, with each port having an average latency time of 19180ms. This means at any given time our API can facilitate up to 70 users at any given time with a 20s wait time for a response. Since our API already utilises the maximum amount of CPU resources it can (2GB with 2.4GHz CPU) in its Cloud Function, we would will consider 70 users to be an extreme maximum that our API can sustainably hold, with a user at the edge of their attention span/wait threshold. The limitations of using wrk2 and load testing are it is difficult to precisely replicate real world traffic as data can encounter problems such as network latency or traffic surges which challenge to recreate during testing, especially using a tool such as wrk2.

Analysis of Performance Hindrance (Above & Beyond)

Currently the final API prioritises accuracy over performance, by comparing the provided location with every location stored in the database. The purpose of this is to verify the location parsed in actually exists and to support geolocation features: if a country, state, iso2, iso3 is provided, find all mentions of any location that would be a subset of this area (Australia should match any report which mentions any of the states or cities). Therefore to accurately analyse how much speed this process is costing us, the execution time and response time in correlation to the response load (number of reports/articles returned). This will also be compared to the performance of the same API hosted from Hong Kong (as geographically it is closer to Australia than Tokyo) and to the performance of a simplistic API which completely omits the process of checking if the location exists by just verifying if the location is in the mentioned in a report. Our recordings are noted in Figure 1.4 And Figure 1.5. The response times were recorded using the Insomnia REST client as it is specifically designed for API testing, so its feature of accurately recording the complete response time was useful. While the execution time was taken from the logs of our API.





The overall trend that figure 1.4 expresses, is that as the response load increases, the response time barely increases, stabilising at a certain response time. This informs us that the overall implementation of our API is not heavily impacted by the extra documents that must be read and filtered out of the database. However when compared to the simple API, there is a clear performance difference as the simple API does not read every city in the world in the database. Figure 1.5 substantiates our deductions as the execution time/response load follows a similar curve. This suggests that there is a time cost associated to filtering a query which returns a large amount of documents (which we had to do as we were unable to get the Firestore custom indexes to work). Thus the act of accessing Firestore is an efficient process, but having to then filter the queried results may be contributing to the overall higher response times, compared to an ideal 1 second response.

Contrastingly, comparing the current API (which is hosted in Tokyo) to the same API code hosted in Hong Kong, we see a difference of about 1000 ms. This may hint that the servers used in Tokyo have more optimised CPU or that Hong Kong's servers could be limited by the Chinese government's control. Nevertheless, it proves that to optimise performance our API should be hosted in Tokyo.

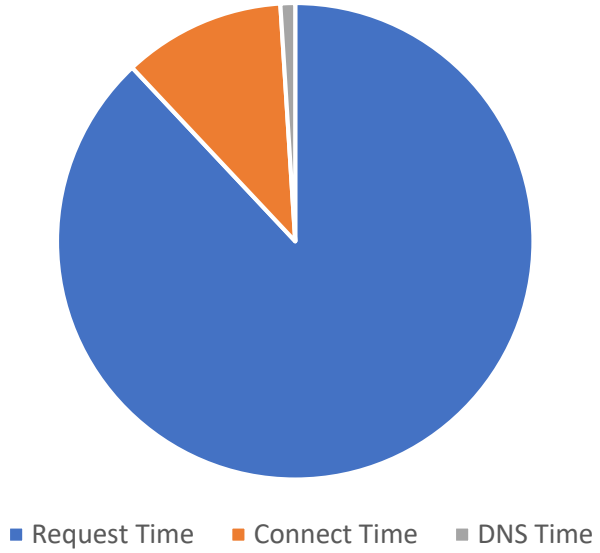
A limitation of these comparison tests is that the execution time used to track the total processing time spent handling the request may not be accurate as it is unclear how exact a Cloud Function is executed. Another limitation of all the performance tests is that web browsers behave differently as do the tools to test APIs (since using Insomnia may not provide the exact response time compared to Swagger or Postman), so the accuracy of these tests could slightly differ depending on what was used, which could lead to a different set of results. Although to ensure our tests we were not hindered by these limitations each of theses tests were performed multiple times with the overall averages from these individual tests documented. Similarly the response times tracked using Insomnia were cross checked with Swagger, ReqBin (an online HTTPS Requests application) and LoadNinja to ensure the final results used were not atypical of the real response time.

Extra Performance Findings (Above & Beyond)

Wait Time Analysis (Above & Beyond)

When generating the results earlier with LoadNinja, it provided a snapshot into the how time is distribution to each process for any given request. An understanding of this would allow us analysed which areas of our performance can be further optimised.

Figure 1.6: Average Wait Time Distribution



This figure displays that waiting for a response took 88% of the time while 11% of the time was from waiting to connect to the API. In this chart, the connect time represents the average latency a user could experience based on the quality of their connection. However since we are in control over 88% of the time taken for a request, optimisations must be made to ensure the response time is at minimum. Besides optimising our code and minimising dependency usage, the primary influence on reducing our API's latency was by hosting it on Google Cloud's fastest server. This option provides our API with a CPU limit of 2.4 GHz, which halved our latency compared to 600 MHz. Additionally with a memory limit of 2GiB, if a user requested every single report in the database, our API is guaranteed to be capable of handling such a request, and future requests as the database increases annually. Although since the Cloud Function itself is hosted in Toyko (as faster alternative server location is available), a degree of latency will always be expected. So the only way to improve the performance of our API without altering any code, is reliant on Google offering a wider and faster service worldwide (so a server in Australia).

Cold Start Analysis (Above & Beyond)

Cold starts are the major hindrance of serverless applications and thus was thoroughly tested in order to gauge the severity of a cold start to our API. Specifically we tested the Cloud Function after it had just been deployed or not been invocated after a long period of time (> 5 hours) and used a typical request the API might receive. Figure 1.7 displays our findings which records the exact point the request was made, until a response had been delivered, which is then compared to receiving a warm response. These times were recorded using the Insomnia Rest Client and the warm start time was used from the earlier LoadNinja test (requesting for 45 reports) in Figure 1.2. Insomnia is specifically designed for API testing, so its feature of accurately recording the complete response time was useful.

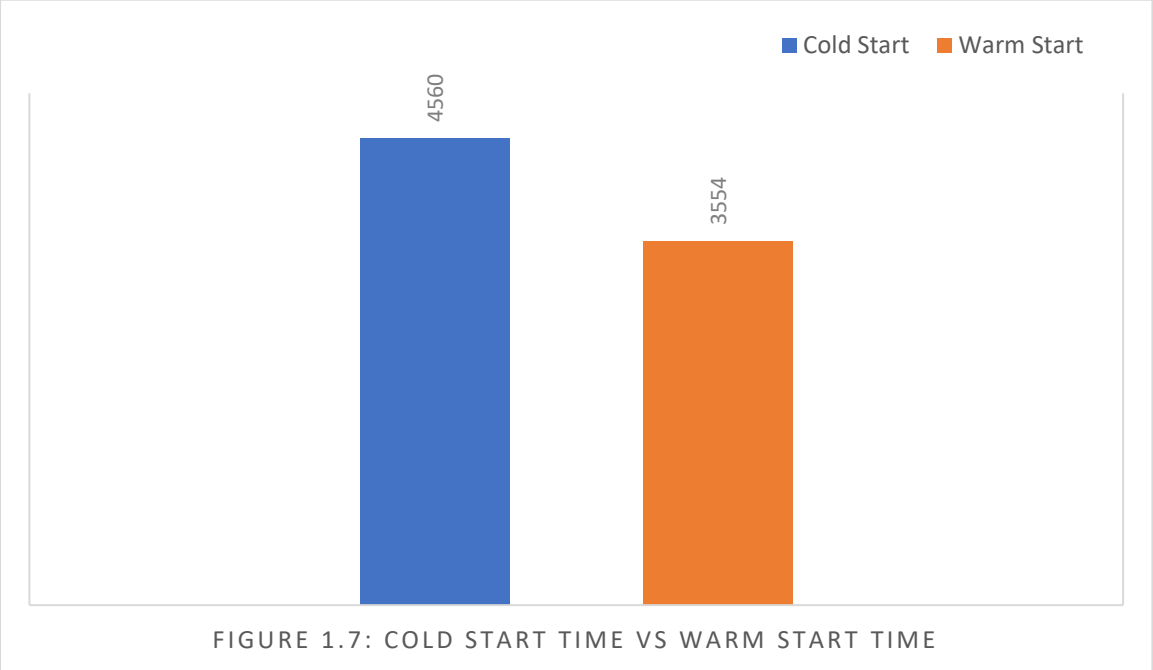


FIGURE 1.7: COLD START TIME VS WARM START TIME

The exact difference between the cold start and warm response duration is 1006 ms. This gap is quite ideal since it hardly increases the wait time for a user by just 1 second. Therefore the results prove that our API has been optimised and the techniques we used to reduce the affects of a cold start are effective. So we do not have to be overly concerned about our API initiating a cold start, as the already low latency times will only incur a small increase to the total wait time.

Overall our API's performs at a fairly reasonable speed especially considering we deliberately designed it to prioritise greater functionality over speed. Hence with every possible optimisation to our code and computing engine implemented, lower latency speeds may only be achievable through a completely different implementation of the API.

Correctness

Having a reliable and trusted API source is imperative to sustaining a functioning service. The correctness of the API can only be tested by verifying a set of specific inputs will result in a set of intended outputs. Each of these tests were ran in a bash script using the linux tool curl, which allows the transfer of data across a network. Curl is suitable for our tests as we only need to make a HTTPS GET request. Then the results of these curl commands were added to a results file and compared to their expected value. This comparison between the expected and results file was done using a Python script with the JSON library which loaded both json objects and then compared them. Specifically this approach was done to ensure the UTF-8 characters could be interpreted the reports are encoded in UTF-8 and allows for each attribute value to be verified. Hence it provides the most accurate and extensive results for our correctness tests. The tables below demonstrate the exact tests we conducted with the inputs that were tested and their outputs. Green represents a successful test (we received the expected result) and red represents the test failed (we received an unexpected result).

Correctness Tests & Results

First we check the usual/expected cases will properly return a 200 response:

Test Case	Does Input Match The Desired Output?
Start_date < end_date	
Location == city	
Location == state	
Location == country	
Key == term1	
Key == term1, term2, …	
Location == iso2 country code	
Location == iso3 country code	
Only Key is empty/not provided	
Only location is empty/not provided	
Both key and location are empty/not provided	

Secondly we check that for a malformed request an appropriate error 404 or 400 response will be returned:

Test Case	Does Input Match The Desired Output?
Request.method != GET	
Number of search parameters != 4	
Search parameters are empty	
Start_date/end_date is not in ISO 8601 format	
Start_date/end_date contain characters other than [:T-0-9]	
End_date < start_date	
Current_time < end_date	
Key > 100 characters	

Location > 100 characters	
A location is not fully spelt out	
A location is not spelt in English	
No reports were found which matched the any of the search parameters	

Lastly we check our API behaves as expected on edge cases

Test Case	Does Input Match The Desired Output?
Start_date == end_date	
Start_date == 1996-01-01	
End_date == 1996-01-01	
Start_date == current_time	
End_date == current_time	
Key = ,,, (or all commas with spaces)	
Key == 100 characters	
Location == 100 characters	

All the tests were successful, which was expected as all these situations needed to be accounted for to create a properly functioning the API. The wide variety of tests conducted for edge cases and for error cases were deliberately done to ensure our API would properly handle these situations and that the code written for these scenarios would be invoked. This is essential was it ensured we had no lazy code (so performance was optimised) and almost every possible scenario the API may experience will be properly handled.

Our Approach to Correctness Testing

Besides attempting to account for every scenario our API could experience, our group constantly tested each deployed/version of our API to ensure no code would interfere with new or previous work. This method proved useful as it allowed us to refactor code, which means robust code and results in expected code. However despite how much planning can be done, our correctness testing may not fully cover an unexpected set of inputs, since covering every single scenario is difficult. Thus we ensured we constantly have error checking throughout our code so if an unforeseen scenario occurs, our API will not crash and respond with an error.

Overall our API has been exhaustively tested for correctness with the final deployed version never undergoing a crash.

Security

Inherently the API is secure from multiple types of attacks as it does not generate any HTML, or use user input (it uses string queries), so the API is projected from code injection, command injection, Server Side Injection and Cross Site Scripting. Hence only the API's serverless architecture presents vulnerabilities.

Since the API uses Firebase's unique Firestore database, this database is not vulnerable to SQL injections. However since it is a serverless web application egress filtering must be managed, as the API is susceptible to information being leaked because of exploitable firewalls, unintended open ports or the usage of dependencies. Packages and libraries can be exploited since they may use internet resources or transfer data and if the dependency itself is vulnerable then the API is also at risk. Testing of our libraries and packages could not really be conducted as it is extremely difficult to thoroughly understanding how each library may use the internet and under what conditions is our API at risk. Thus we just ensured we used the latest libraries as they are the most stable and secure versions.

Security Tests & Results

To conduct these tests, the linux tool nmap will be used as it can thoroughly scan a website/host to detect any vulnerabilities and which networks are available. This makes the tool more than suitable for the types of tests we will run. Note all these tests were run when the Cloud Function had not been recently invocated and when it had been previously invocated, which both led to the same results. Additionally the web scraper does not need to be tested as it safely operates offline.

Firstly we tested which ports were accessible from our API by initiating a TCP scan of every port, since minimising the amount of ports open would ensure no data could be unnecessarily leaked.

PORT	STATE	SERVICE
80\tcp	Open	http
440\tcp	Open	https

Figure 1.8: All open ports

Figure 1.8 demonstrates the results, which show only 2 ports are open which both connect to the 1 API function. These results are as expected as the API must allow data to be transferred and usable, which is why the ports are open. Additionally the results also indicated that 65533 ports were filtered, which highlights how every other possible port is blocked by a firewall.

Next we scanned the API in order to find any possible firewall vulnerabilities or reveal any unintentional access points which could leak data. These tests involved investigating the firewall settings, attempting to do a FIN scan, trying to use decoys to fool firewalls and see if

the Windows IPsec filter could be bypassed by using port 88. Accordingly, the tests all failed resulting in either error:

```
You requested a scan type which requires root privileges.

QUITTING
```

```
You have specified some options that require raw socket access.

These options will not be honoured without the necessary privileges.

WARNING: no targets were specified, so 0 hosts scanned.
```

Receiving these error messages is desired since Google does not want to expose any of the infrastructure that maintain their servers, demonstrating the extensiveness of Google's protection. This means by using the Google platform, we receive Google's Cloud Armor security, so our API is just as much as risk as Google is, which is fairly secure. However since Google abstracts most of their security measures from us, the extensiveness of detecting firewall vulnerabilities is limited to the simple testing we have conducted.

Our security considerations

The API's security could be further tightened with a NAT gateway which would filter outbound traffic and require a user to access a VPC Access Connector in order to use the API. However enforcing such security measures would only increase the complexity for users who wish to use our API and increase the total wait time of sending a request to receiving data. So our group did not implement such security rules in favour of greater performance speeds and since the base level of security Google provides eliminates almost any threat.

Overall our API will always have the latest high quality protection as Google and Firebase continually refine their and monitor their servers.