



NUI Galway
OÉ Gaillimh

National University of Ireland Galway
Computer Science & Information Technology

Final Year Project 2021/2022

**Generating Analytics Based on
Steam Friend Networks**

Cathal O'Callaghan
18365213

 <https://github.com/NeoSteamFriendGraphing>

 <http://neofyp.com>

Glossary	3
Abstract	4
Objectives	4
Personal Objectives	4
Project Objectives	5
Research and Planning	5
Literature	5
Hosting	7
Practical Testing	7
Similar Projects	9
Implementation	10
Microservices	11
Crawler	11
Datastore	14
Frontend	16
Index/Home Page	17
Crawl Page	19
Graph Page	20
Interactive Graph Page	23
Shortest Distance Page	25
Common	26
Infrastructure	26
Logging	26
Messaging/Queue System	27
Data Persistence	28
Metrics	29
Uptime Monitoring	31
Hosting	33
Testing	34
Security	37
Documentation	38
Development	39
CI/CD (Continuous Integration And Continuous Deployment)	40
Languages	42
Evaluation	43
Encountered Issues and Solutions	43
Excessively large user documents	43
Accessing the Steam API as few times as possible per user crawled	44
Accidental leaking of sensitive files	44
Issues arising from high throughput concurrent workloads	45
Degradation of performance for large graphs	46
Caching Strategies	46
Memory Leaks	47
Metrics	47
Logging	48

Testing	48
Survey	48
Project Management	49
Hosting	50
Future Aspirations	50
Project Objectives	51
Personal Objectives	51
References	52

Glossary

Steam Popular computer game marketplace and social network

CI/CD: Continuous integration and continuous deployment.

ELK stack: Elasticsearch, logstash and Kibana. Popular stack that's used for log storage (elasticsearch), log aggregation (logstash) and log analysis (Kibana).

Job: A job in its most basic sense when referred to in this report is a message published and consumed in the RabbitMQ messaging system that indicates that a particular Steam user needs to be crawled. Some other metadata is also in a job such as the current crawl that it belongs to and which user this current ID is friends with.

Devops: Development operations. Development of software that relates to its operation and deployment tasks.

VPS: Virtual private server

Abstract

From the outset I wanted to develop a distributed system for my final year project. With a strong understanding of the properties that distributed systems are known for I then set out to find an application that not only needed a distributed system to function but one that excels when it can leverage the many strengths of one. To the end user the fact that the application is powered by a distributed system isn't at all important but for myself it would be an invaluable learning experience. Ultimately I settled on the idea to create a web application that analyses friend networks on the popular gaming platform Steam. My main goal was to be able to show users interesting facts about their wider network of friends (something that they know quite well to an extent) that they most likely would not have known before. The project's name is Neo and I will be referring to the project as Neo during this report.

Objectives

The objectives for Neo can be split into two categories:

1. Personal objectives with respect to managing and developing a distributed system effectively
2. Project objectives which cover features that the application will offer

Personal Objectives

During my work placement in third year the team that I worked on developed and maintained a handful of microservices written in Java. I found that the approach that was taken when working on a distributed scale was fascinating as it was all completely new to me. Many of the little things that I picked up during my placement were invaluable as I didn't read about them in the various books on microservices that I've read since. Although I could not manage quite as many separate microservices as my team could, I aimed to maintain the same core principles of distributed systems that I had learned in my own system. This involved taking on new practices across project management, devops, project architectures and general programming. Under the research heading I will later discuss how I came up with the following list of requirements that my proposed distributed system must follow.

My final project should:

- Never require me to manually ssh into remote instances to hunt down bugs. Log aggregation and analysis should be integrated seamlessly to enable me to diagnose issues in real time across all deployed services.
- Automatically provide rich metrics in real time to effectively display load and performance on the system across its many services at any given time.
- Be deployed across multiple remote hosts and not require any server side code to be executed on my own hardware. This in turn will help me to understand solid devops practices as I will be the sole maintainer of all services and infrastructure dependencies.
- Not use any managed pieces of infrastructure such as databases from cloud providers. This will teach me valuable lessons in the life cycle of deploying and maintaining services by setting up all the infrastructure and dependencies myself.

Project Objectives

Project objectives are essentially the only things that matter to everyone but myself. It doesn't matter how impressive the system powering it is, if the experience of the end users of my application isn't something worth telling their friends about then the project is ultimately a failure in everyone's eyes.

The core features offered by Neo will be:

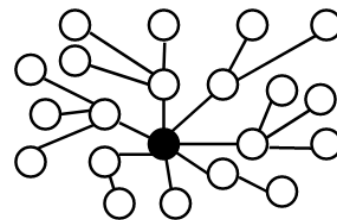
- Users can request that the service crawl their friend network and generate statistics based on these users in an interactive format. Analysis should be given based on the geography of users and the games that they play.
- Users can request that the service crawl two users' friend networks and not only generate statistics for both users as before but also find the shortest path between these two users if it exists and display this information to them in an interactive format.

Determining which statistics and analysis should be displayed in the final product was an important task and although it was given considerable thought and its results were treated as pieces of inspiration instead of guidelines. I had devised some basic traits about a network that I wished to highlight such as geography and the games owned by the users but there wasn't ever a clearly defined list of graphs and statistics that I had to implement. With regards to the final output of the project I wanted to place equal emphasis on both the statistics and the interactive graph output of the network. Either of the two features could be constructed entirely from the proposed data that was to be collected from the Steam API so there weren't any worries during development about not collecting enough data which would later hinder the potential analysis that could be undertaken. It should be noted that this advantage is entirely based on diligence in the way of planning to ensure that enough data was being collected by the system to facilitate both features.

Shown below is a quick guide to the levels of crawling offered. In the report when a user's friend network is mentioned it can mean either of these levels and not just their direct friends which would be a level two crawl.



Level 2: The user and their immediate friends are crawled. This is everyone on a user's friends list



Level 3: The user, their immediate friends and friends of these immediate friends are crawled.

Research and Planning

The preparations taken for this project can be split into two methods: reading and researching about distributed systems and other applications that I planned to use and practical testing and setting up of some of the applications in order to get some experience before committing to using them.

Literature

One of my favourite technical books that I've read was "Building Microservices: Designing Fine-Grained Systems" by Sam Newman. It gives a high level overview of some core microservice principles, their strengths and real world examples of why certain design patterns are must-haves when it comes to microservices. Although many of the examples were for topics that I never intended to use for my project I found that it was a very good experience to have read the book. I made sure to take notes throughout my reading that I believed might help in future planning and implementing of proposed microservices. I started reading this book not for examples of why I should use certain technologies but instead to get some basic rules for things that I should keep in mind when architecting my project as I knew that I'd need all the help I could get. Overall I picked up many great practices and design patterns that I didn't know before so it was a very beneficial experience. Some of the key points I noted from this book and a great article written on the principles of microservices[2] by Martin Fowler are as follows:

- **Loose coupling:** Newman explains that "the whole point of a microservice is being able to make a change to one service and deploy it, without needing to change any other part of the system"[1]. Loosely coupled services strive to share as little detail as possible with its dependent services apart from the minimal information required for communication between them. In practice this means that to call an endpoint on service K the caller shouldn't need to understand exactly how that service fulfils that request. It should only need to know the minimal amount of information that the endpoint requires to fulfil that action. By ensuring loose coupling is applied everywhere internal changes to dependent services don't trigger an avalanche of broken services which depend on outdated code.
- **Componentization through services:** Microservice/distributed system architectures extends the practice of dividing up code into functions to the level of entire services. Systems that successfully make use of componentization with their services enable themselves to deploy individual services at a much faster rate. Small changes in single services only require their own service to be deployed compared to small changes in monolithic applications requiring the entire system to be redeployed. When multiple teams are working on the same application this is a necessary step to stop a large and computationally expensive redeployment of a monolithic app occurring multiple times per day. Horizontal scaling is a prominent feature of successful distributed systems as cheaper hardware can be used to spin up new instances instead of relying on a few high power instances that realistically can only scale vertically (adding more compute power to individual instances).
- **Infrastructure automation:** Continuous integration (CI) and continuous delivery (CD) are key factors in the lifecycle of a microservice. CI relates to the automatic testing of code usually before deployments or successful pull requests/commits. With the quick release cycles of microservice architecture projects placing confidence in their testing suites is more important than ever as you need to be able to trust that when a test suit passes that the code will not introduce regressions or hidden bugs. Fowler places emphasis on a successful CD system being boring. Minimal supervision and tinkering should be needed when dealing with deployments in a successful CD system as this enables developers to focus on more important tasks.
- **Plan to fail:** When working with code running as one process failure is sometimes expected to occur. When working with code distributed globally the chance of failure is nearly guaranteed due to the many new problems that must be accounted for such as unreliable communication networks and service uptimes. Any and all calls to other services should be expected to be able to handle failures in a reasonable manner. In addition to handling failures gracefully these failures should not be silent and logging/metrics applications should highlight these as soon as they happen so that they can be taken care of as soon as possible. Vague error messages are no use to anyone and ultimately increase the chance that a bug will not be fixed until it happens again and more context can be examined in an attempt to fix it. In the case of a failure occurring, all relevant details regarding a failure should be available to give insight as to what conditions were met that the testing obviously did not account for to cause it.
- **Data belongs only in databases:** When a service has local data saved and its environment is wiped on redeploy or moved to a new environment that data is lost. A service should hold no data for anything longer than the span of internal requests so that a service can be deployed and instantly be operational without having to worry about

the many issues that patterns like sticky sessions bring. Another important rule that must be followed when utilising datastores in any distributed system is that access to the data must be regulated via a strict API in a service. Direct unregulated access to databases via any service only leads to problems. This is mostly due to a lack of regulation being placed on transactions that occur as a database usually only ensures that ACID (atomicity, consistency, isolation and durability) and similar qualities are maintained. This means any service who wishes to interact with data in a database must first interact with a specific service as an intermediary that imposes its own rules on all interactions.

From the beginning I realised that just research on its own without some form of testing wouldn't be adequate. Having some key principles that I had to follow from earlier research were nice to have but I felt that implementing parts of my proposed system would help particularly in developing my own opinions on certain applications/design patterns that I was proposing to use for my final project. When it came to all of the planned infrastructure that I had planned to use I had only used some of them before during placement and for the rest I had never used before. The main pieces of infrastructure that I needed to get hands-on experience on was logging (creation, aggregation, persistence and analysing), metrics (collection and query writing) and the messaging system.

Hosting

Given the fact that I'm building a distributed system it would be entirely counterproductive to host all or most services on the same physical hardware as any connectivity issues would lead to a widespread outage. Therefore, the long task of finding the best balance between an adequate range of VPS cloud hosting plans that were both effective but didn't cost me a small fortune had to be done. In the past I had used Digital Ocean sparingly for testing but although its services are of great quality their tiers of packages offered left a lot to be desired for hosting many small services. I am very much against funding anything Amazon related for personal reasons so AWS was out of the question and the same can be said for other major cloud hosting providers such as Azure and GCP.

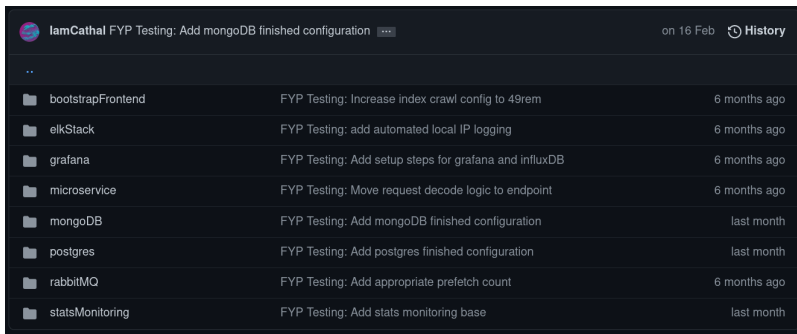
The operating costs that I would be taking on was also something that was at the forefront of any decisions on choosing hosting providers as although I wanted the best hardware to host the project, I didn't want to end up too out of pocket in doing so. During planning I mapped out some of the most prominent pros and cons of the cloud providers that I had researched. With this information it's clear that I chose OVH. There isn't anything about them that I could fault them for and at the time and after having completed the project I would be happy to use them again in the future.

Provider	Pros	Cons
GCP	Lots of tiers, data centers and good reliability	Google, Documentation/Pricing unclear
AWS	Lots of tiers, data centers and great availability	Amazon
Azure	Lots of data centers and good reliability	Microsoft, Poor choice of tiers
OVH	Lots of tiers, EU data centers, Great web UI and provisioning	2FA texts are in french
Digital Ocean	Simple web interface, excellent monitoring and UX	Tiers are too expensive/not varied enough

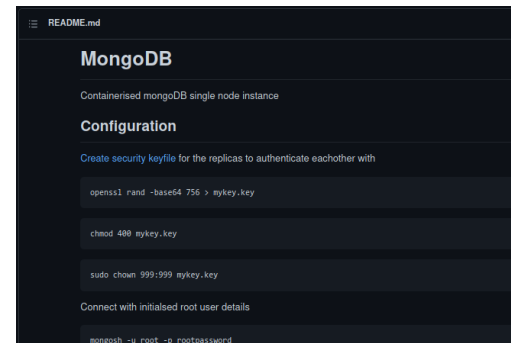
Practical Testing

Finding existing configurations of these online for testing was easy most of the time but it often required a decent amount of research to secure them and less often required me to write my own configurations from the ground up. I used docker and docker compose in the past and planned to continue using it to host my infrastructure as it's very straight forward and popular with lots of example configurations for popular applications available online. Taking the time to prepare these functional configurations of applications to use not only helped me get an idea as to whether it was appropriate to use for my project but it also gave me an immediate starting configuration for using the applications in my actual project. Another primary concern of mine was committing to less than ideal dependencies that initially seemed fine but later would be a significant detriment to the project. With personal projects in the past that were much smaller this concern was also in my mind but with the scale of this project I knew that I had to give every design choice an objective examination as to whether it would really be best for my goals in the long term.

For each piece of infrastructure I had a working configuration that I could use for deployments before I was fully committed to using them. This period was quite tough on some days getting to grips with new applications and researching the myriad of error messages that were thrown during testing but it was definitely the right thing to do in my opinion. After this stage I was free to focus entirely on developing code for my microservices knowing that when the time came I would instantly be able to interact with my planned infrastructure. In the end of the researching phase I had working prototype configurations for the frontend, logging, metrics, microservice system statistics, a database, the messaging system and the base code for my microservices that each would start from. In line with this planning I made sure to note important configuration settings and steps that I took just in case I had forgotten some minor obscure fixes that I had taken. This was invaluable later on as issues appeared that required me to redeploy some pieces of infrastructure long after I had forgotten the minor steps that I had taken beforehand to fix obscure bugs.



Test configurations of various applications



Extract of the configuration notes for setting up MongoDB

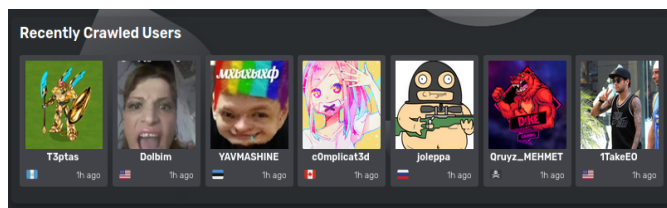
There were a number of other features from various non-Steam related websites that I planned to implement in some fashion which include the new user feed and a live updating globe on the homepage. Most sites have only static content on their homepages but Neo is an application that can often have a huge amount of activity happening that users never see. As was the case with many features of the frontend I often came up with methods to display the data after I had it instead of having an end goal in mind and gathering the data necessary to fulfil it.

A favourite website of mine, Bandcamp, a music commerce website geared towards small artists has a live feed on its homepage for all purchases made on the website. Having this dynamic content on the page brings so much life and gives an intriguing insight into what other users are currently using the site for. With just static content there's no reason to look any longer at a page after browsing it but with a real time dynamic feed of new purchases/users I often found

myself glued to the page waiting for things like purchases made from Ireland or from Irish artists. Therefore implementing a similar feature for Neo only seemed appropriate. With Neo I could provide users browsing the home page with a real time feed of all new users being crawled by the system through websockets with little to no effort. Specifics of this implementation will be detailed in the implementation section.

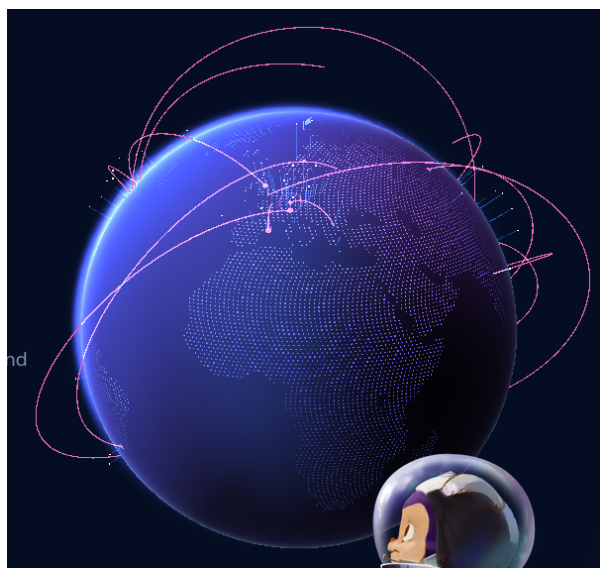


Bandcamp recent purchases feed



Neo recently crawled users feed

The second notable feature that I wanted to incorporate in Neo was Github's pull request globe on their homepage[4]. Although it is not exactly real time and filters out most real data in favour of a more varied and cherry-picked range of data points for the final version, it's a great example of using existing data in a new light that really stands out to me. The globe highlights distances between projects and contributors from pull requests on the site and the individual pillars indicate commits to projects. I planned to have a similar globe with locations being retrieved from the users in the previously mentioned user feed and although it looked great its implementation was plagued with many issues that ultimately forced me to have it replaced. Initial testing that I had done with the globe unfortunately was not representative of the end goal and therefore I didn't notice the issues that would eventually cause the feature to be removed. The biggest issue was the significant drop in framerate when the globe was being used. In short, the resolution of the mapping of countries was unfortunately quite precise which caused it to require more processing power for each rotation operation that was performed roughly 200 times per second. These issues were only slightly alleviated after hunting down a smaller resolution mapping but the overall performance impact was not worth including the globe. Other prevalent issues regarding the markers for users' locations also made any effort in trying to improve the feature to end up being useless.



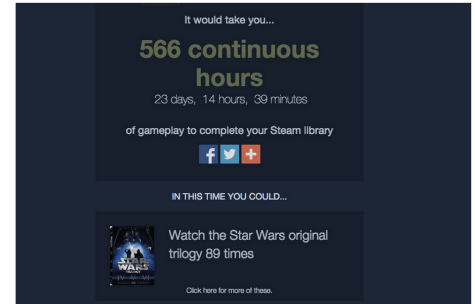
Github's globe



Neo's (deprecated) new user's location globe

Similar Projects

When it comes to learning from my experiences using similar applications there is nothing that specifically fulfils the use case of my project. The closest existing project in terms of graphing networks of steam friends is a simple weekend project[3] that has not been functional in years. Given the short timespan spent developing the project the features are quite simple overall but its main feature of displaying friend networks in an interactive graph format is what I aim to achieve, although to a much higher standard for the overall experience. Its graph representation is quite simple and lacks any kind of exploration but the basic idea of representing a friend network of friends in a graph format is still quite interesting even in this basic format. The second main feature of my project is that it displays a wide range of statistics based on the crawled friend network and although that too is unique there are some popular existing sites which generate statistics for a single user such as SteamDB and SteamID. While SteamDB is a self-described 'database of Steam' and it also aggregates various statistics based on a steam account. Its most popular project is the 'pile of shame', where the site analyses your games library for with regards to games that are owned by not played before and uncompleted games and returns some interesting facts about how much a user has spent on the platform or the more important things they could've done with their total playtime across all games. Shown below is one such metric that I particularly liked about the service. It shows a user's total playtime which they might be familiar with and shines a different light on it. Both projects combined wouldn't be my finished project but I certainly took inspiration from parts of each to incorporate and iterate into my own.



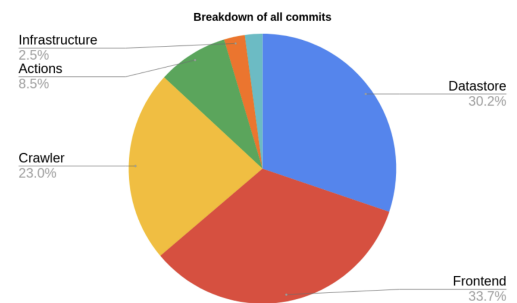
Interesting metric shown by SteamDB's pile of shame

I began my planning and research at the beginning of September and spent my first month going through a lot of trouble getting acquainted with new applications that I planned to use. Throughout this time also I started to plan the overall architecture of Neo and made revisions based on my testing where I felt my previous plans weren't adequate. For the most part my plans for the proposed applications that were going to be used were quite accurate and for most of the applications which reached the stage where I manually tested and made deployments for they were used in the actual project. A considerable amount of effort was dedicated to referring back to the core principles of a distributed system during this stage to ensure that I wasn't sabotaging myself by cutting corners with regards to following them at such an early but pivotal stage of the project's development.

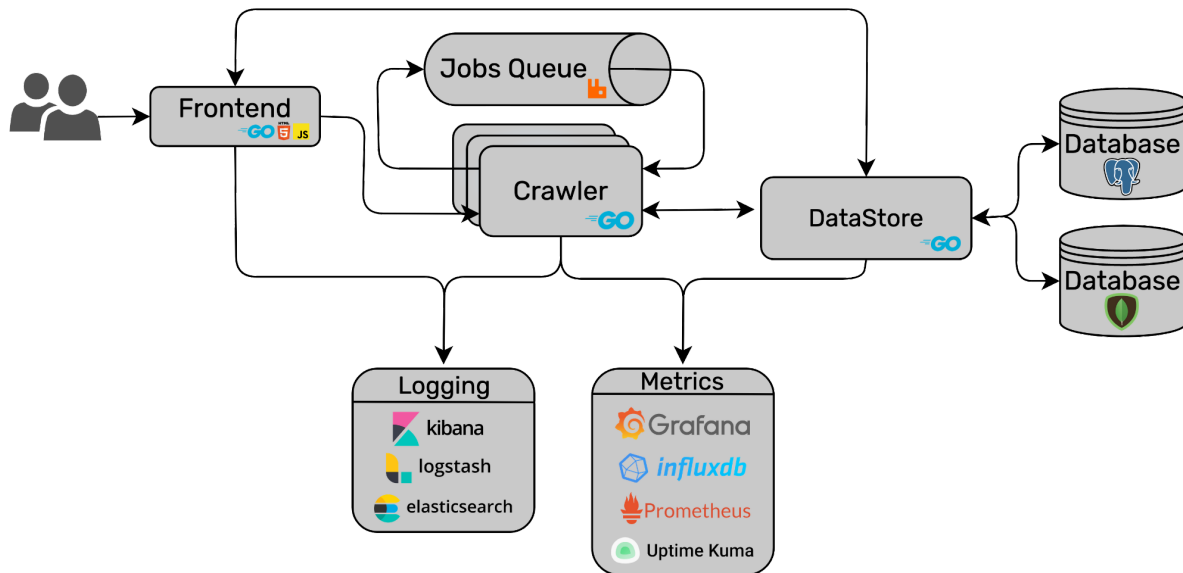
Implementation

Here are some general statistics to convey the scale of the system in a concise format:

- 3** Microservices
- 9** Servers
- 37** API endpoints
- 141** Tests
- 3,945** lines of client side Javascript
- 6,838** lines of backend Go code (tests not included)



Breakdown of commits to each topic



Neo architecture

Microservices

- Frontend
- Crawler
- DataStore

Infrastructure

- Jobs Queue (RabbitMQ)
- Databases (MongoDB and PostgreSQL)
- Logging (Kibana, Logstash and Elasticsearch)
- Metrics (Grafana, InfluxDB, Prometheus and Uptime Kuma)

Microservices

Crawler

The crawler service interacts with RabbitMQ to consume, process and publish crawl jobs. All of the interactions with the Steam API happen inside of crawler instances and it's currently the only service that has multiple deployed instances. In addition to crawling users consumed from RabbitMQ crawler instances are also tasked with aggregating all of a crawl's data into one document that is served when requested by users visiting the graph page for a given crawl. This is done using a worker pool implementation that enables the service to asynchronously collect all the users in a manner that, while is extremely fast, does not induce load onto the system that would cause negative effects like congestion for other services accessing the database.

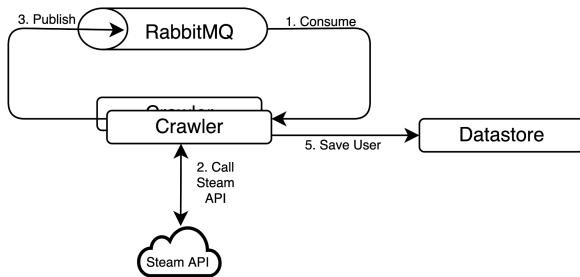
Shown below are illustrations of the main tasks performed by the crawler service. Its primary task is executing crawls which involves the service retrieving data from the Steam API and saving these individual users to the datastore service. Its second task is compiling all of these users retrieved during a crawl into one graph data document which is served on the frontend and holds all data that was retrieved for a given crawl.

Crawling process:

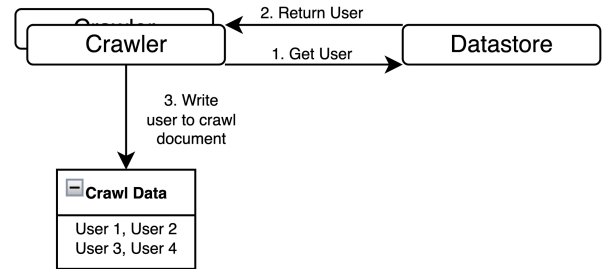
1. A worker thread on a crawler instance consumes a job from RabbitMQ
2. The main crawl loop is executed and all of the necessary data is retrieved from the Steam API. An in-depth illustration of this crawl loop will follow.
3. If appropriate (the current level does not exceed the crawl's maximum) publish all friends found to RabbitMQ
4. Save the user that has been crawled to the datastore service

Graph data generation process:

1. Once a crawl has completed all users are now saved in the database. For each user in a crawl their data is requested from the datastore service
2. Datastore returns the saved user document
3. Each user retrieved as part of a crawl is compiled into the crawl data document. After all users have been added this document is sent to the datastore service where it is saved and later served upon request.

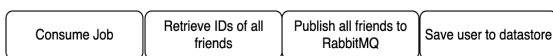


Crawling process

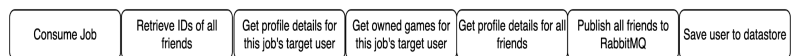


Graph data generation process

Shown below is an illustration of the tasks performed as part of crawling one individual user (the previously mentioned crawl loop). There is a big difference between the case where a user exists in the database and therefore does not need to request more data from the Steam API and when the user does not exist in the database but they both essentially fulfil the same purpose by publishing all friends from a user to later crawl them if appropriate. For each job the loop first checks to see if this user already exists in the database firstly because there's no point in calling the Steam API for a user that is already saved and because the roundtrip for this query is considerably less than a single call to the Steam API. For each crawl job profile details for the user (username, country, avatar, profile URL), friends of the user, profile details for all friends and details for all of a users' owned games must be collected.

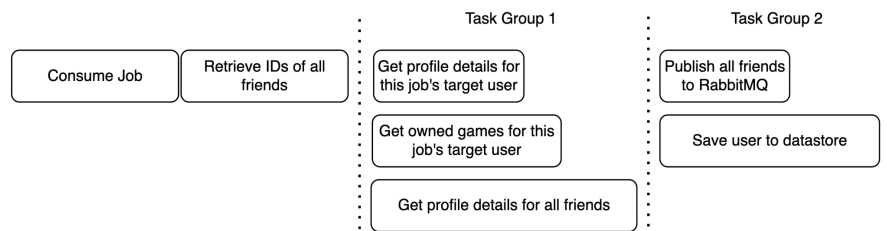


Crawl loop tasks if the user exists in datastore

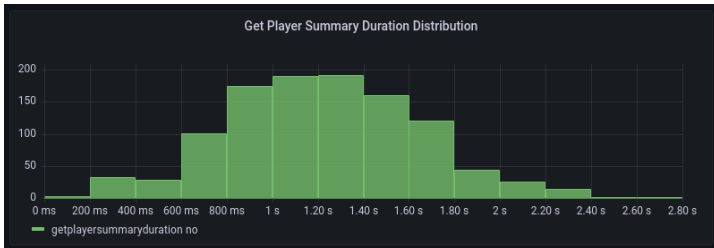


Crawl loop if the user does not exist in datastore

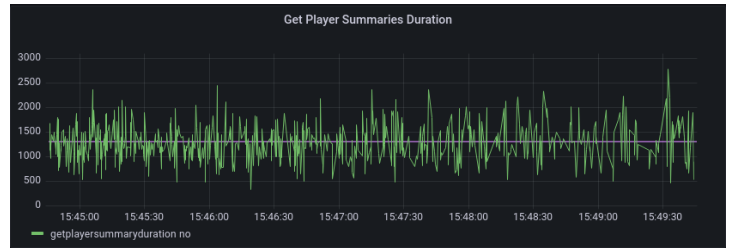
However, it's clear to see after illustrating the loop that some significant optimisations can be made to increase throughput of the system. Shown below is the optimised loop that separates the tasks that can operate asynchronously and those that require certain other tasks to be completed before being initiated such as publishing friends to RabbitMQ and saving the user document after all data is collected. All tasks in the first group must be completed before any task in the second group can be initiated. This means that each task group's duration is only as long as the longest running request, not as long as all of the requests combined. Improvements to this core loop have the most significant impact on overall crawl times as it's called for each individual user and suffers the most from idling in order to not trigger rate limiting measures from the Steam API.



Shown below is a graph of the distribution of calls to a Steam API endpoint where rate limiting measures are undertaken to prevent API keys from being used in a manner that would exceed the allowed limits set out by Steam. Distribution of these keys is in a round robin fashion where a central authority within the crawler service manages the distribution of the keys to ensure that no key is used only once in the allotted time frame that it is allowed. Currently this internal threshold (which is configurable via an environment variable in the .env file) is 1300 ms which means that each key can only be used at most once every 1300 ms. This distribution highlights that a seemingly near equal proportion do not have to wait to access a key compared to those that must wait for a key to become available.

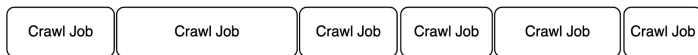


Distribution of the duration of calls to a Steam API endpoint with rate limiting measures in place on crawler

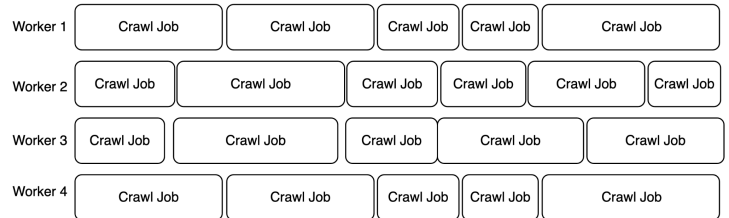


Durations of calls to a Steam API endpoint. The purple threshold signifies the threshold timer for use of the Steam API keys. Durations above the threshold were required to wait to access a key and those below did not have to wait

Synchronously crawling users was never going to be a viable solution so during testing I invested heavily into developing a race condition free and performant worker pool crawling implementation. Extreme care must be taken when writing asynchronous code as many issues such as deadlock, livelock and starvation can and will eventually arise. With synchronous crawling the rate of users being crawled was bottlenecked by the system itself. However, with a worker pool implementation the system is bottlenecked by measures that are put in place to limit interactions with the Steam API so that they do not exceed the specified rate limits imposed on all consumers of the API. The amount of workers is defined as an environment variable as during testing this value was changed many times to find the optimal value. A higher amount of workers isn't simply better than a smaller amount as the thundering herd problem[21] becomes quite prevalent when a new crawl is started. Essentially what would happen is even with 30 worker threads the first few jobs received would not be completed for anywhere from 40-90 seconds as after each request to the Steam API it then had to wait in a queue with all of the other threads for the next call to the Steam API and multiple requests are required to the Steam API. Over time the throughput would become more constant but from the perspective of the user it seemed as if the system was doing nothing for a questionably long amount of time. Lowering the amount of workers enabled constant throughput even when starting a new crawl and did not impact too heavily on the overall throughput of the system.



Synchronous crawling



Worker pool implementation crawling

A similar worker pool implementation is used for collecting all users' data which is served as the entire crawls data on the frontend. This document contains all of the collected data on every user that was in a particular crawl and can often

exceed 16MB when uncompressed for larger crawls. Since this implementation is only accessing datastore to collect user data that has already been crawled through the Steam API it's only bottleneck is the rate at which it's allowed to send requests to datastore. Some consideration was given to determining the optimal rate that ensured that this operation wasn't too slow from the users perspective but at the same time wasn't overloading datastore unnecessarily for the sake of saving 1-3 seconds.

Most prominent in the crawler service with its huge amount of interactions with datastore is the logic surrounding these interactions such as retries. Retries are an essential part of a distributed or even any connected system as networks are guaranteed to fail at some point. For each interaction with another service a standard practice is followed. A maximum of three retries are allowed and in between these attempts the thread sleeps for an amount of time that is longer than the last. The sleep times are calculated using the function $(2 * i) + 16$ where i is the number of failed attempts. If three unsuccessful attempts have been made then the service then handles this case appropriately (stops execution of the service and logs the context surrounding the error returned). In each case where retries are employed there is logging to note that it has occurred and the associated request and response in the interaction to help with later diagnosis. With my current experience using Neo whenever there is one retry triggered this usually leads to all retries being triggered and a subsequent crash of the service as brownouts (in this case a temporary degradation of performance of another service) from other services are extremely rare. Most often other services being entirely down is the cause behind retry attempts but regardless of the infrequency of its usefulness it's particularly important to have robust retry mechanisms in place.

One of the shared features of all microservices is logging of key system performance metrics e.g CPU and memory usage percentages. All microservices asynchronously upload these metrics on a periodic basis to enable monitoring of system health from Grafana. Given that this functionality is implemented in pure Go code it's extremely lightweight and doesn't require any kind of sidecar deployment like Filebeat (although in this case it's an advantage to deploying Filebeat as a sidecar). Logging memory usage specifically is the most important metric for detecting memory leaks that increase the idle memory consumption very slowly over time through the forgetting to properly close resources such as bodies of HTTP requests in Go.

Datastore

The datastore service is the only service with access to either of the MongoDB or PostgreSQL databases. It handles all queries to these databases and serves any other data related needs such as new user and crawling status event stream updates which are used on the frontend. From the outset this service was designed to act as a thin client sitting in front of the databases to restrict and control access to them. Datastore is also tasked with calculating the shortest distance between two given users when a dual user crawl is requested using a very similar worker pool implementation that crawler instances use for asynchronously collecting all users during the graph data generation process.

The real time update feed of new users that the system crawls and updates on crawling statuses are both handled by the datastore service. In both cases the datastore service utilises change streams in MongoDB. Change streams allow a client to receive all (subscribed to) events from a database (MongoDB equivalent of a traditional table) as they happen in real time. Instead of connecting clients directly to the database the datastore service acts as an intermediary and parses the incoming data and publishes data on request to clients on the frontend through websocket connections. In the case of the new user events API endpoint all connecting clients receive all events whereas for crawling status updates clients specify a specific crawl ID that they wish to receive updates about. This simple difference removes all possible situations where over-fetching of updates from multiple active crawling status updates that are happening simultaneously instead of just one single crawling status that a particular user is interested in. I had never created websocket API endpoints before but I found the process of getting acquainted with them to be quite natural as it's simply a regular TCP connection that's upgraded into a websocket connection on initialisation. In both use cases the flow of data in the websocket connections is only one way even though websockets are designed to just as easily enable bidirectional real time communication between two clients.

One of the necessary design choices that was ultimately required to be imposed after some minor testing with consuming and serving finished crawl documents was that the document itself must be gzipped (compression standard) when publishing and consuming finished crawl documents due to their huge sizes. On my home network my bandwidth is often subpar to say the least and for anything beyond moderate sized crawls I would trigger timeouts put in place by the datastore service to limit the maximum amount of time that can be spent reading or writing a single request as I could not upload the crawl document in time. Gzip is a very popular file compression format and all the major browsers natively support decompression of packets which have the "Content-Encoding: gzip" head present. From a microservice standpoint the finished crawl document is gzipped before its sent to the datastore service and decompressed once it arrives. This drastically reduces the bandwidth required during transport by sacrificing a small amount of time and compute power to decompress on arrival. When serving the finished crawl documents this is gzipped again before being transmitted over the network. In practice the reduction in content size of these documents is often between 3-5x which is an impressive improvement for such a minor change. Simply increasing the read and write timeouts is quite simple but would be a foolish change to make. Allowing requests to take longer than the default 30 seconds to read or write opens the door to particularly poor performance from the system being recognised as normal. From a security standpoint this would further increase the severity for a Slow Loris attack[17] whereby an attacker initiates many connections to the server and coordinates their interactions to be as time consuming as possible in an effort to starve the server of resources.

Given that there are 24 API endpoints alone in the datastore service, the importance of standardising the basic API design pattern used was essential. Although many of the endpoints differ in what inputs they accept and the responses they return some general features were applied when developing all of them. Shown below are the common features that all endpoints must fulfil:

- If input is being accepted it first deserializes it into its native format before being sanitised and filtered where appropriate to detect any malicious/invalid input.
- The core logic that the API is exposing is executed.
- The response is initialised using a DTO that is defined in the common shared library. Appropriate headers and HTTP status codes are set and the response is written

Should an error occur during the fulfilment of a request then it's dealt within an appropriate manner which is based on if it's caused by user error or an error returned from the service itself. In the case of an error caused by a user error a simple utility method writes a generic invalid response to the user. No logging is required as users inputting invalid input is not a concern of mine. In the event that an error occurs that is caused by the system itself (either a bug or failures in assertions) then as much context surrounding the error as possible is logged in addition to returning a modified invalid response to the user. The invalid response received by the user in this case is inspired by Google's old HTTP 500 error response which informed the user that something went wrong and that a team of highly trained monkeys should be given what looks to be an extremely long request ID to help fix the issue. My implementation is similar and returns the request ID of the failed operation in the response which can be used in the logs to find all activities executed as part of a particular crawl to show the error itself and the context which preceded the error.

```
func (endpoints *Endpoints) InsertGame(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    bareGameInfo := common.BareGameInfo{}

    // Deserialise the request body into the native datastructure designated
    // for storing minimal game details
    err := json.NewDecoder(r.Body).Decode(&bareGameInfo)
    if err != nil {
        // Send back an invalid response with HTTP status code 400 as this
        // input is invalid. Do not specify the exact reason why
        util.SendBasicInvalidResponse(w, r, "Invalid input", vars, http.StatusBadRequest)
        return
    }

    // The input is valid, attempt to insert this document
    success, err := endpoints.Cntr.InsertGame(context.TODO(), bareGameInfo)
    if err != nil || !success {
        // If unsuccessful or an error was returned this means that
        // there is an issue within the system that must be fixed
        configuration.Logger.Sugar().Panicro("failed to insert game: %v", err)
    }

    response := common.BasicAPIResponse{
        Status: "success",
        Message: "very good",
    }

    // Write appropriate Content-Type and HTTP status code headers
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusOK)
    json.NewEncoder(w).Encode(response)
}
```

Example API flow for an internal game details insertion endpoint

```
// SendBasicInvalidResponse sends an invalid response
// back to the user with specified status code and
// error message. This is used for invalid user input
func SendBasicInvalidResponse(...) {
    w.WriteHeader(statusCode)
    response := struct {
        Error string `json:"error"`
    }{
        msg,
    }
    json.NewEncoder(w).Encode(response)
}
```

Handling of user error

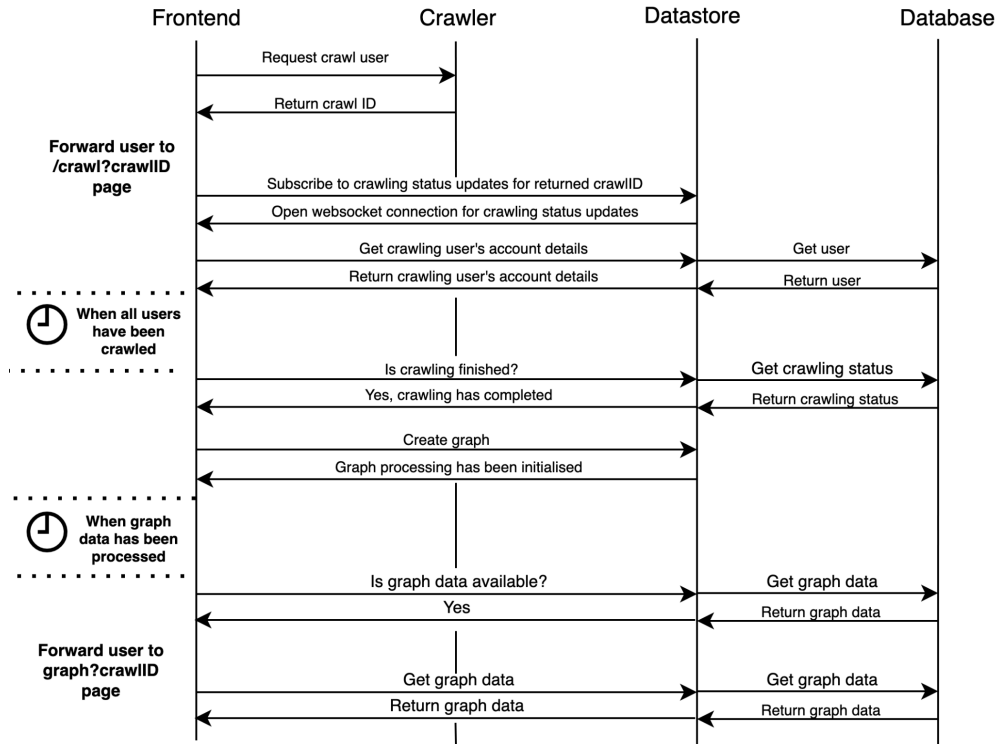
```
defer func() {
    if err := recover(); err != nil {
        vars := mux.Vars(r)
        w.WriteHeader(http.StatusInternalServerError)
        response := struct {
            Error string `json:"error"`
        }{
            fmt.Sprintf("Give the code monkeys this ID: '%s'", vars["requestID"]),
        }
        json.NewEncoder(w).Encode(response)
    }
}
```

Handling of a service error

The widespread use of request IDs is another trick I picked up during my work placement. All requests (or subrequests spawned from an initial request) include a request ID field in HTTP headers and in logs which allows for all of the activities of one request to easily be found in logs after. When multiple crawls are happening simultaneously reading the most recent logs are useless as it's incredibly difficult to differentiate which logs are a result of a specific crawl. When a crawl is initialised it is assigned a unique request ID (also referred to as a crawl ID) which is used to track its progress in the logs if anything goes wrong and is the unique identifier used to visit the graph page by users. Services utilise KSUIDs for generating and validating UUIDs across microservices. These UUIDs can be generated very quickly and are guaranteed to be unique which enables many concurrent processes to generate UUIDs simultaneously and never arrive at a situation where two are the same. Although I do not personally take advantage of KSUIDs "natural" ability to be sorted by creation timestamp, it's the biggest difference between its own implementation and the official RFC 4122 UUID specification[19].

Frontend

The frontend is a simple service that mainly is tasked with serving static files such as pages, images and client side Javascript. The vast majority of code written for the frontend service is client side Javascript that interacts with datastore to fetch data and render graphs and statistics to the webpage. Discussion on the frontend will be divided into sections for each page to best express the many design decisions chosen and to give a thorough explanation of the requests made from the client from crawl initialisation to retrieving the crawls data on the graph page. The overall theme of the site is inspired from my own personal site cathaloc.dev[12], where I initially came up with the design myself.



Requests made from when a crawl is started to when it's served from the user's perspective (ignores internal crawler and some datastore requests)

Beginning at the home page when a user gives a valid Steam ID and level a crawler instance is requested to initiate a crawl. If a crawl has already been undertaken at the specified level for this user, then a duplicate crawl is not started and the user is forwarded to that graph page. If successful, the uniquely assigned crawl ID will be returned. This crawl ID is used to identify this crawl and the user is forwarded to the crawl page with the crawl ID specified in the URL.

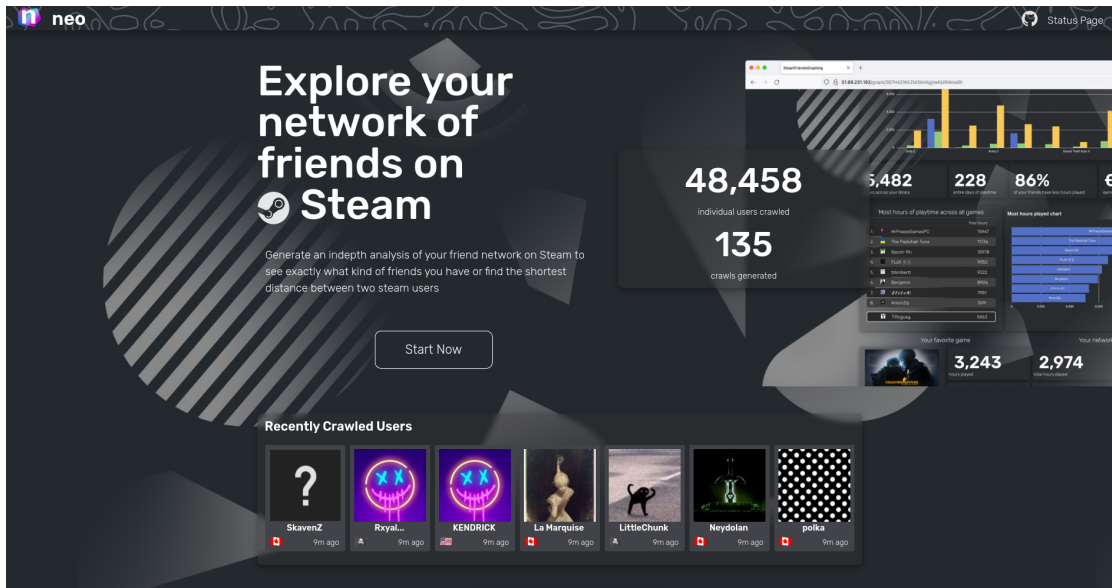
On the crawl page a websocket connection is initiated with the datastore service that subscribes to updates for the user's specified crawl. As new users have been crawled this progress is passed to the frontend for the frontend to render to show the crawl's progress. In addition to this some account details of the crawl's target user are retrieved from datastore and also displayed on the page. Once the frontend has received the final updates of the crawl it then requests that datastore initiates the graphing phase. It should be noted that datastore itself first also verifies that the crawl is actually complete before starting this process. The graph processing phase is where the datastore instance collects all of the users from the crawl and combines them into the final document that's served for users visiting the graph pages for crawls. This usually takes some time so the client will periodically check in with datastore to see if this data has been written to PostgreSQL yet and when it has the user is forwarded to the graph page.

Once on the graph page the processed crawl document is retrieved from the datastore service and its data is rendered for the user.

Index/Home Page

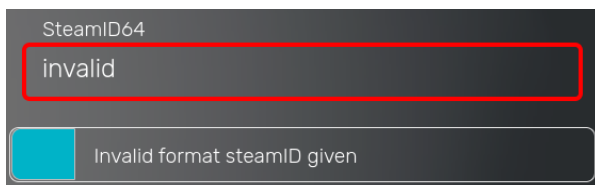
I'm a strong believer that within the confines of (preferably) the initial view of the home page or less, a concise explanation as to the features/use case of a site must be expressed. The objective is to show the user exactly what services the site offers in a manner that entices them to delve deeper. Any negative elements/bugs on the homepage reflect particularly poorly on the website similar to a spelling mistake on a CV.

With these important principles in mind, I set about constructing my home page. From the initial view of the homepage a clear explanation of the features offered are given in a clear and concise manner, a preview of the graph page is displayed and dynamic content in the form of the recently crawled users feed and overall statistics on the total amount of users crawled and crawls completed is given. Datastore is periodically queried to retrieve the newest statistics for these two metrics and updates when the values have changed without any input or page refresh required from the user. The displayed new user feed is rate limited as I found that as throughput of the system increased with two deployed crawler instances the viewing experience started to worsen.

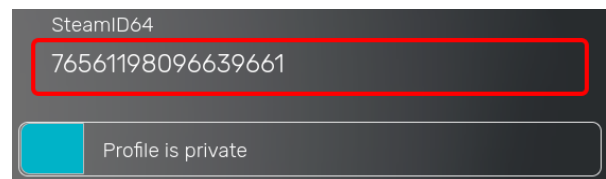


Home page view

The very next element seen after scrolling down from this initial view is the interface for starting a crawl. I invested a decent amount of time and effort into designing this interface in a manner that would express the inputs in a simplified design. For example, the steam ID inputs display error messages when the ID is not valid or if the profile is private. Right below these input fields there's a link to the FAQ detailing how a SteamID64 can be obtained since for most Steam users this might be a new concept.

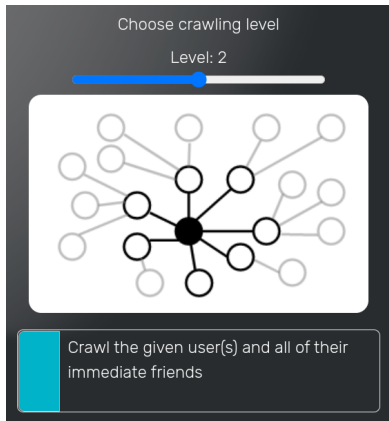


Invalid steam ID error message

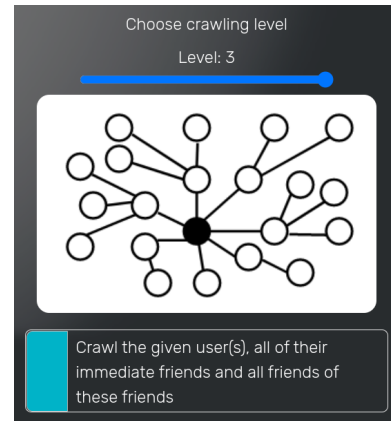


Private profile error message

The choice of which level to initiate the crawl with is then visualised for the user to clearly see the impact that their choice will have. I can appreciate that the text explanation on its own might not be sufficient and therefore I came up with the idea to illustrate the crawl level and I'm quite happy with the end result.

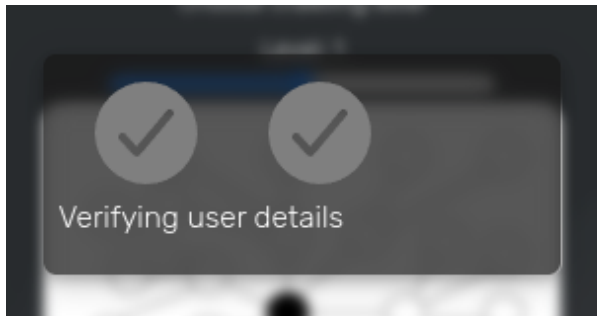


Level 2 crawl illustration

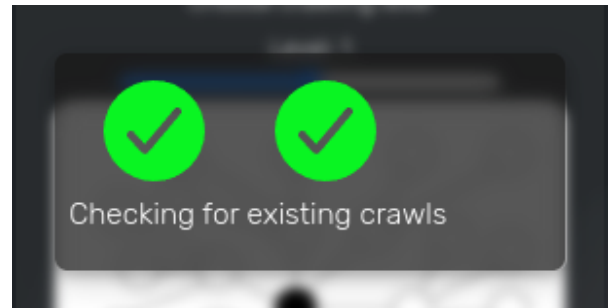


Level 3 crawl illustration

Following these checks when the crawl button has been clicked a new status box appears over the crawling interface with details on the status of the pre-crawl checks. These checks include validating that the user(s) given has their profile privacy setting set to public and then that no crawl has been executed before at the requested level. If a profile is private then the crawl is not initiated as no information can be gathered. If an existing crawl exists and only one user is requested then the user is forwarded to the existing page. If two users are requested and one has been crawled before then the crawl continues on as normal and is only required to crawl one user. These two consecutive checks do not normally take longer than 200 ms from beginning to end but including this status box as an indicator as to what the system is doing behind the scenes in my opinion is much better than having the user believe that the system is hanging for an unknown amount of time.



Status box indicator when neither pre-crawl checks have been completed



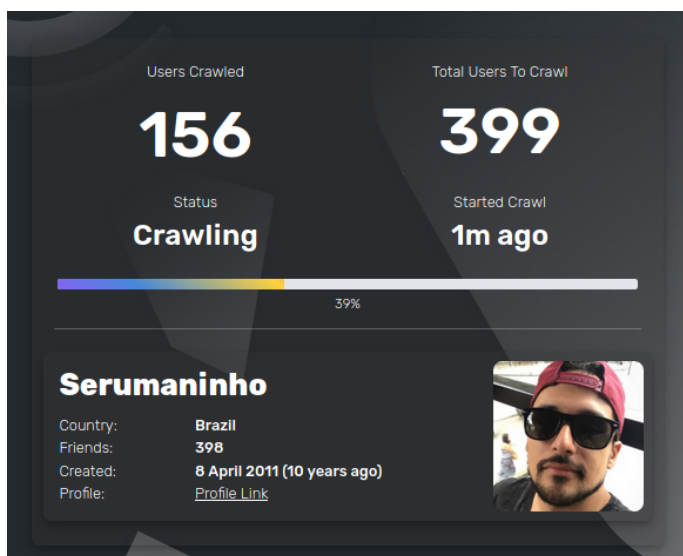
Status box indicator when both pre-crawl checks have been completed and the user will subsequently be redirected to the crawl page

After this I've included a simple FAQ for the sake of having it to explain details like how a Steam ID can be obtained. After creating the home page up until this point I felt that a lot of the activity of the site and work produced (e.g crawls) was hidden away needlessly from nearly all of the users. When a user creates a crawl that's only visible to that particular user and nobody else. I then opted to include live updating feeds for the most recently generated crawls for all users to browse on the homepage. Users who perhaps don't feel comfortable jumping right in and starting their own crawl or don't own a Steam account can then view some other existing crawls without having to supply any Steam account. I feel that for the little amount of effort that this feature required to be implemented that it turned out to be one of my most valued features on the frontend in terms of impact in opening up the site to casual users. Like with the dynamic updating users crawled and crawls executed counter this periodically checks for updates in the background and refreshes the list when appropriate. However, this feature makes clever use of providing the timestamp of the most recently retrieved crawl and when querying datastore to ensure that only crawls after this given timestamp are returned in order to stop over-fetching of data.

I would like to note that the dynamic data such as the recently completed crawls, total users crawled and total crawls executed that are retrieved on the homepage is always served from cache from datastore. If this data was not cached then each new refresh/visit to the homepage would result in four separate requests that would each require a query to the database. On a small scale this is not a big issue but if Neo was ever to blow up in popularity and many users simply accessed just the homepage and nothing else I fear that this would have some particularly negative effects on the performance of the MongoDB instance. On a more security related note this problem is similar to a 13 attack. Essentially these classes of attacks aim to turn as few requests as possible into as many as requests as possible on the server side against a target[13]. In the context of the home page one request to the home page from an attacker would result in four to the database which would give it a multiplication factor of 4. To combat this all queries for this data are served from caches in datastore that are periodically updated on 30 and 60 second intervals as the consequences for this data not being entirely accurate at the exact time of viewing are negligible.

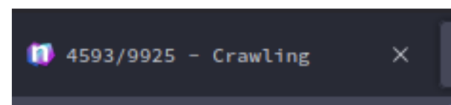
Crawl Page

The crawl page displays the current status of a user's crawl which is being executed. The crawl process can often take longer than 30 seconds and therefore it's important to keep the user updated as to how it's progressing instead of leaving them guessing as to if it will ever be finished. The main element on the page is the live updating crawling status that subscribes to updates on a specified crawl ID from a websocket endpoint on the datastore service. Some profile details that have been retrieved from the user. The client side code running this page is not too interesting as it's just a matter of establishing a websocket connection to the datastore service and rendering the new statuses as they are received. Once all of the users have been crawled client side Javascript requests for the crawl data to be compiled into one document which is then served when viewing the graph page. Once this document has been processed and is available to be served the user is redirected to the graph page.



Dynamic interface on the crawling page to update the user on the current status of their crawl

One of the smaller features that I'm proud of is the dynamic tab title when a crawl is active. The window's title is updated dynamically as more users are crawled. This allows users to browse other tabs and only have to glance at the title of the Neo tab in order to see how the crawl is progressing instead of requiring them to deliberately click back into the tab. While this feature is particularly simple in terms



Dynamic title during a crawl

of its implementation I find it to be very useful when I was executing long running crawls and wanted to check its progress while watching content on another tab.

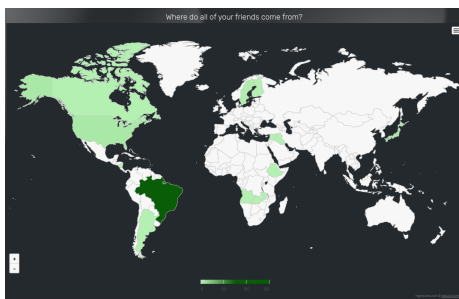
Graph Page

The graph page is easily the most complex of all of the pages due to the number of statistics and analysis that's carried out. This page pulls the processed graph data from datastore as a data source. All of the graphs are from Apache's Echarts which is an amazing and rich graphing library. The main goal of this page, which was a core goal of the project, is to show users facts about their friend network on Steam that they were not aware of before. Many people and myself included might roughly know that a lot of their friends are from a certain country but being able to see the actual distribution was always noted by users should be news to most people.

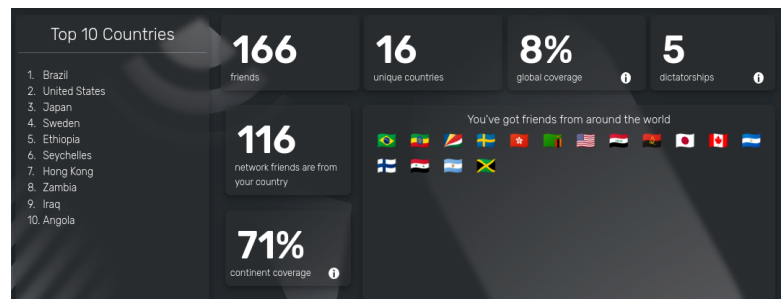
The graph page begins with a simple 2D graph of the friend network with toggles for users from the top countries. This representation is good to have right off the bat but isn't meant to be the full experience that's offered, the standalone interactive graph page is a much more fleshed out and interactive experience. Following this there's a map and some statistics that give an insight into the geography of the friend network. The map element is quite nice as it's fully interactable with options to view in fullscreen, download as an image/pdf, allows for zooming in and out and displays the exact friend count per country on hover. The statistics displayed allow for a more analytical representation of this data as it shows the amount of friends also living in the user's country, amount of unique countries that friends live in and other metrics.



2D graph of the crawled friend network

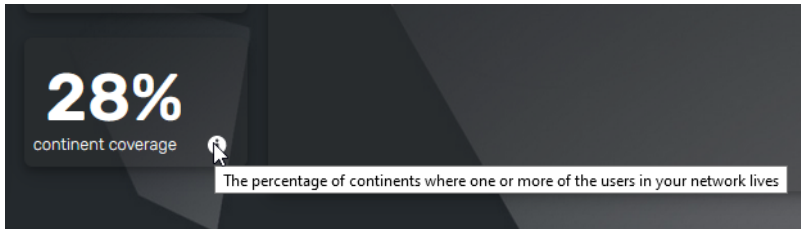


Word map view of all users countries

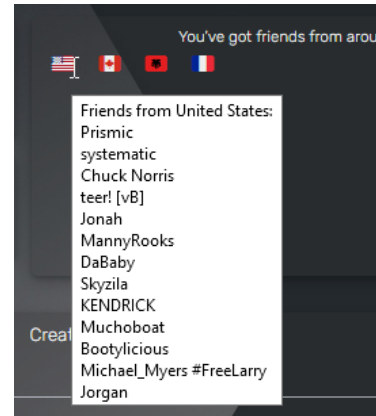


Geographical statistics of the friend network

Minor quality of life improvements are also present such as the tooltip icons for statistics that might be even slightly obscure. These were introduced to offer a short and concise explanation as to what a metric represents where I felt they were necessary. Everywhere across the graph page where this information icon can be seen this same functionality is implemented. Each flag icon also displays the specific friends from that country on hover which was also a piece of feedback that I received during testing. This was requested as without this feature there's no way to see which friends are from a specific country on this page.

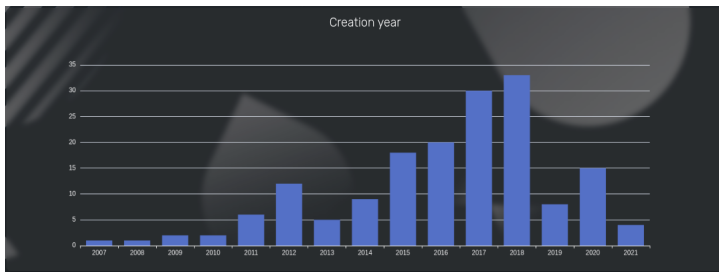


Tooltip icon that gives more information to the user

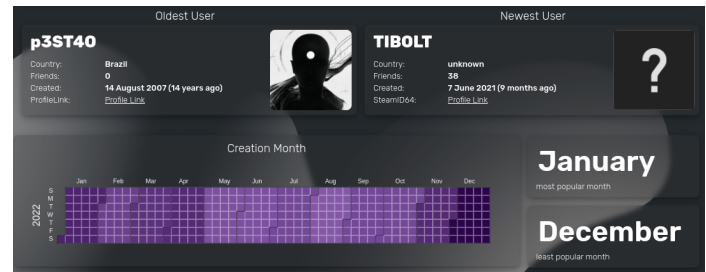


Another tooltip icon example

The next section focuses on the account creation dates of users in the friend network i.e when each user joined the Steam platform. The top graph gives a quick visual indication as to the distribution of creation dates and the following cards highlight the oldest and newest users in the network since this information is not specifically available in the proceeding chart. Next is a slightly more in-depth analysis that shows the distribution of creation dates divided into months with the most and least popular months. I found during testing and general use that November/December/January were most often the most popular months and the following months after were most often the least popular. I was inspired to use this graph format as I'm quite used to viewing this style of graph used on Github for the tracking contributions. Again, Echarts offers rich graphs that specify the exact figures for each month when hovered over which grants users a deeper insight into the data compared to if they just visually glance over this element. I'm particularly fond of this approach as at a glance some rough information is visible but if the user wishes to see more they are quickly able to gain a deeper insight without cluttering up the original simplified view that most users might view as being more than adequate.

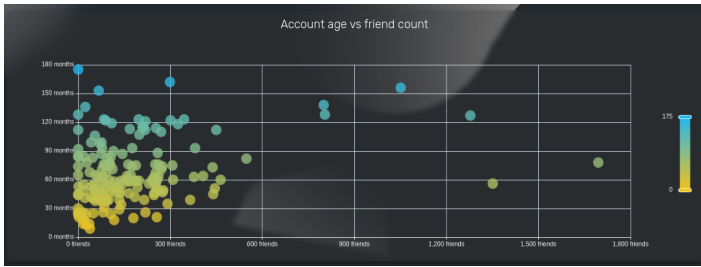


Account creation histogram

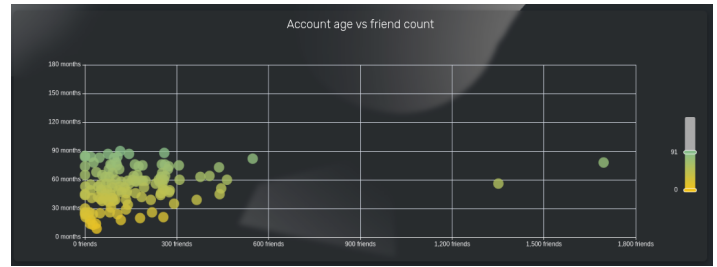


Statistics based on account age and creation month

To finish off this section there's a graph showcasing the relation between a user's account age and their overall friend count as well as a card showing the friend with the highest friend count. On most graphs the relation between older accounts and accounts who have more friends can be seen although there are often outliers with some very old users having little to no friends. Although users that have no friends might appear to be impossible it's merely due to the fact that when requesting friends from the Steam API for a user who has strict privacy settings enabled no friends are listed. On the right hand side there's a slider that allows for the filtering of data points based on the account age which can be used if the graph is too crowded for a user's preference.



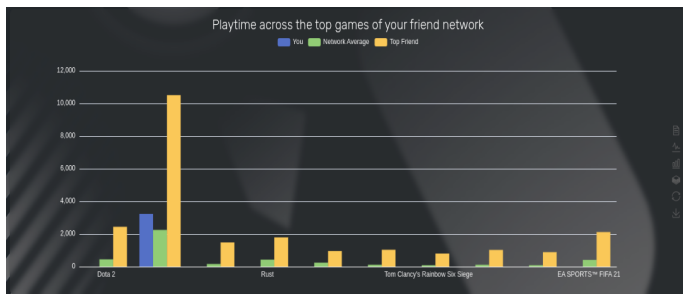
Account age vs friend count graph



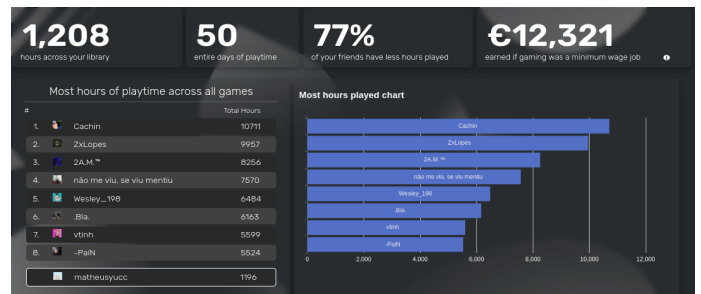
Filtered account age vs friend count graph

The next section offers an insight into the most popular games played by the friend network (in terms of total hours played) alongside statistics based on the playtime of the target user across their Steam library. My favourite metric is the amount of money earned if gaming was a minimum wage job given the number of hours that a user has spent playing games. I was inspired to create this metric based on SteamDBs 'pile of shame' which gives interesting substitutes for which people could have spent their time doing instead of playing games. I feel like these two metrics aren't exactly insults but they do poke fun at the idea that this time spent playing games was wasted which I would most definitely disagree with but again highlighting what could've been done instead of gaming is interesting, especially when bringing up the fact that a considerable amount of money could've been earned.

The next two elements rank and chart the users in the network with the highest amount of hours played across their library. Often, I noticed that with networks of 40-80 friends the first place user had a considerable number more hours than second place but with larger networks the overall distribution tended to be a lot more evenly spaced out. After having seen so many I'm not surprised at all seeing users with 20,000+ hours which as Gladwell suggests could've been spent achieving mastery in two separate skills[14].

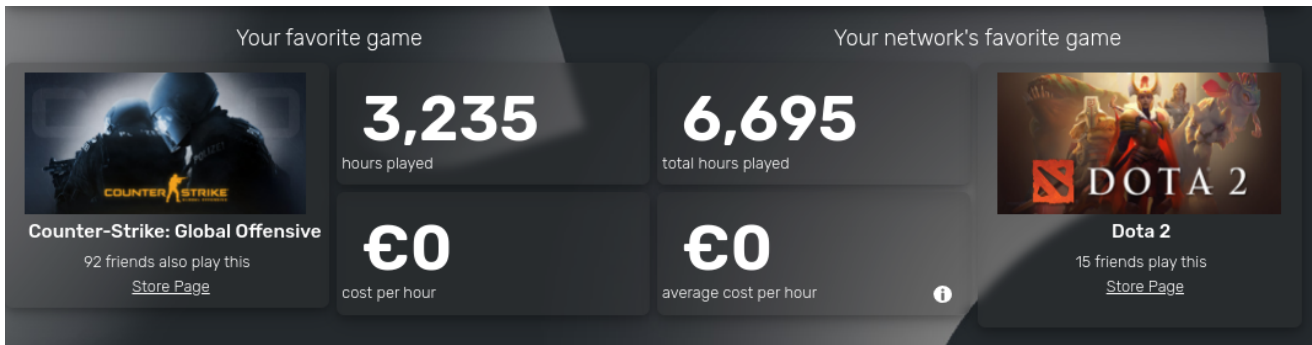


The most played games of the network



The network's users with the most total hours played on Steam

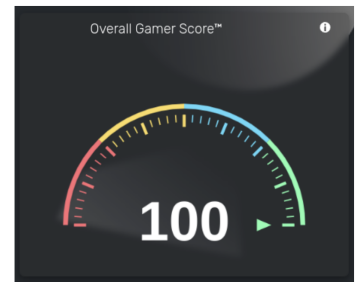
The final statistics displayed in relation to games played by the network are the hours played and cost per hour (total hours played/cost of the game) for both the user's and network's favourite games. I always found it interesting to calculate how much I spent per hour for some of my favourite games and some friends also thought it'd be an interesting metric to show. The game details such as full name, Steam store URL and header image isn't retrieved from datastore to save on the overall size of the crawls document and instead are retrieved from the Steam API when the page is loading. However, it's actually retrieved through an endpoint on the frontend service but this is simply a proxy to overcome issues with CORS that come with interacting with the Steam API through a website.



Your and your network's favourite game showcase. In both cases these games are free.

The overall gamer score metric is a Neo specific metric that's meant purely for bragging rights. Essentially, it's a ranking ranging from 1 to 100 of a user's account based on the total number of hours played and their friend count.

Finally, there's a link to the crawl's interactive graph page. Details on why this graph isn't included on this page will follow in the interactive graph page section. This has a simple video preview of an example graph just so it's not a simple link that might be overlooked which would be a real shame as the interactive graph is one of the major features of Neo.

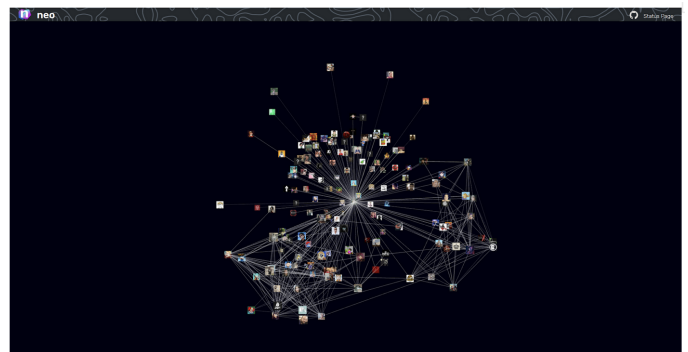


Gamer score

Interactive Graph Page

The choice to move the interactive graph from the graph page to its own standalone page was mostly due to performance and user experience reasons. The 3D graph is generated and rendered using ThreeJS, a Javascript library used to create and display animated graphics on the web[15]. During development I had the interactive graph placed at the bottom of the page and due to some unforeseen circumstances regarding its rendering and subtleties with my linux system the performance was extremely poor. Simply by moving the interactive graph to its own page there was a huge performance boost to both pages. An added benefit of this switch has to do with the fact that the graph is best viewed in full screen (covers the entire web page, not the traditional entire screen version of fullscreen) and attempting to force it to render at any smaller size can be quite a mess. Having the graph on the bottom of the page essentially consumed vertical scrolling capabilities and without using the browsers scrollbar made it appear to the user that they were stuck viewing the graph. Again, with the interactive graph on its own standalone page no vertical scrolling is necessary so this isn't an issue anymore. Another benefit to a standalone page for a graph is that dual user crawls can use this same page for displaying the shortest total distance between two users in a graph format where possible.

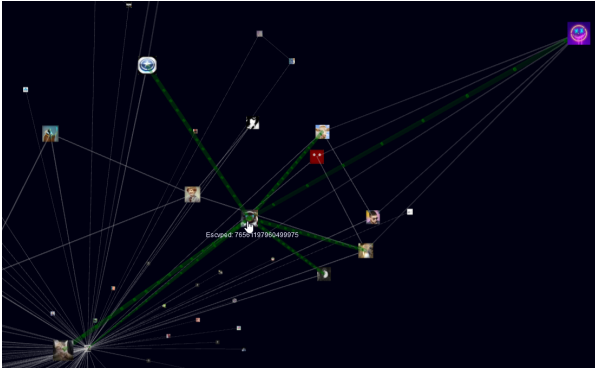
Using ThreeJS for rendering the graph made the whole development a pleasure to have gone through. The library has some very nice features for dynamically rendering nodes and edges with built-in concise and simple methods. I found that accomplishing my goals with styling of edges and nodes when highlighting the shortest distance, zooming in on selected users, highlighting friends of highlighted users and obtaining the optimal spacing of nodes based on their friend counts to be quite a straightforward experience. Nowhere along the way did I feel like I was writing code that went against what ThreeJS offers and I found extending my own features to be a great experience as I had close to no ThreeJS exclusive concepts to understand. There are two contexts where an interactive graph is used which I will



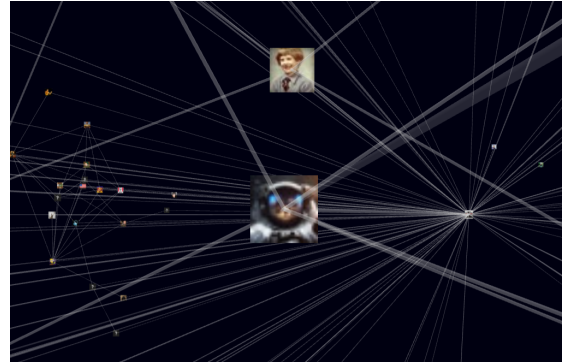
Single user interactive graph page

detail in the following sections: a graph for a single target crawl, a graph for two user crawls where there exists a shortest distance between the two target users.

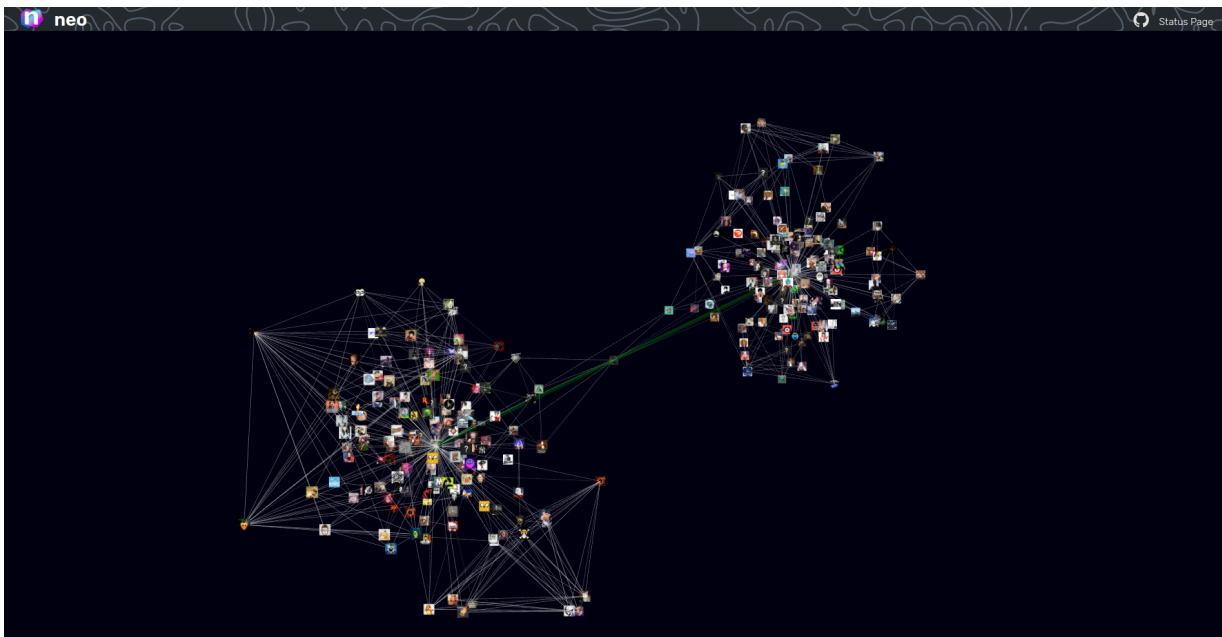
For single user crawls no inherent highlighting is required when there is no input from the user. When a user is hovered all of its connecting links (friend connections) are highlighted clearly with a thick green outline. This helps to differentiate the friend connections of a specified user over all of the other non-descript connections that also clutter up the surrounding area on very crowded graphs. Another feature or trait that I've tried to follow in the graph page has been the ability to view the real users' Steam account if they're displayed on the website. In the interactive graph when a user is clicked a simple animation zooms in on the user and when completed opens a tab to this user's real Steam account. This gives the displayed data more of a feeling of it being real as anyone can click on a user and verify that these displayed friends are actually shown in the graph as well.



Links highlighted when a user is hovered over



When a user is clicked an animation zooms in



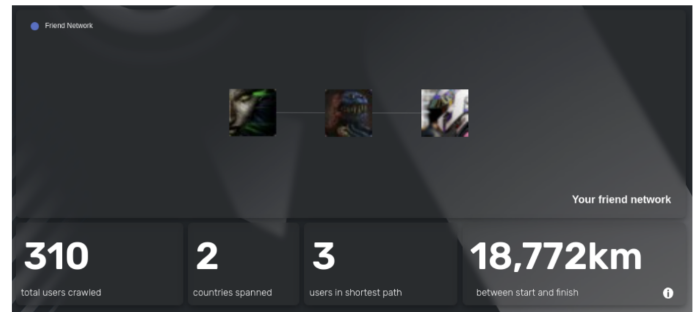
Shortest distance between two users. Highlighted green path shows the shortest distance

The second context that the interactive graph page is used is for displaying the shortest distance between two users if it exists. This requires the same representation as a single user graph but in addition to this the shortest distance should be clearly highlighted to the user as that's the main feature that the dual user crawl is meant to display. Implementing this

functionality was thankfully quite simple and effective with minimal effort which is all that I could have hoped for. The path is highlighted using a distinctive green highlight which serves to clearly differentiate itself from all other edges in the graph.

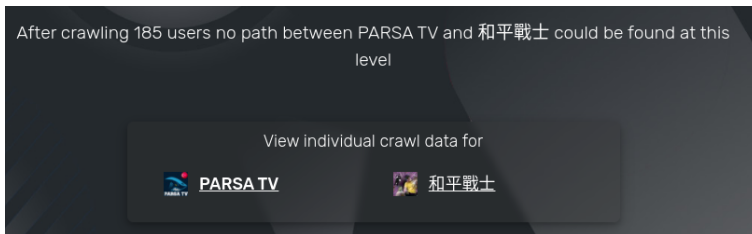
Shortest Distance Page

The shortest distance page is the equivalent crawl page for dual user crawls. It's where the user is forwarded to after a dual user crawl has successfully been executed and is available for retrieval from the datastore service. It is however much simpler than the graph page and does not need to display an in-depth analysis based on the two combined friend networks. With this in mind the page is quite simple, first there is a simple 2D graph representation of the shortest distance chain and some metrics based on this and the total network crawled.

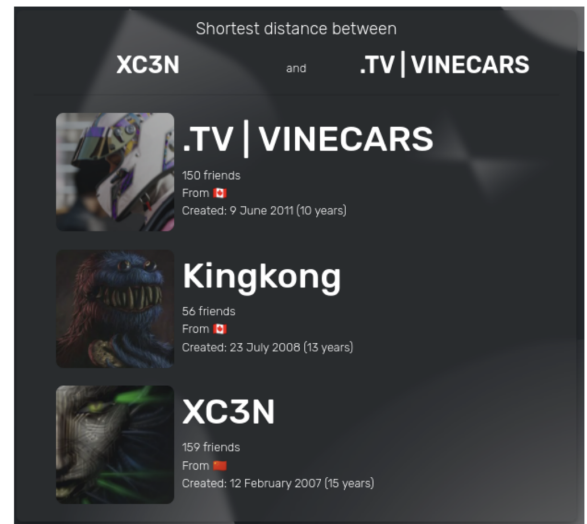


Statistics based on the shortest distance

Following this there's the main representation of the chain of friends that connect the two target users. Simplified versions of the user cards seen on the graph page are used for these users and as per usual their Steam profiles can be found by clicking on their profile pictures. The client side Javascript powering this page is very simple overall and is just tasked with rendering data received in an appealing manner. One of the advantages with the approach taken during a dual user crawl is that both users involved have their own individual user crawls generated which means the crawl for each user can be viewed as if they were requested as a single user. When no shortest distance can be found this is expressed to the user and links to both user's individual crawls is displayed



Unsuccessful shortest distance crawl view



List view of all users found in the shortest distance path

Common

Common is a shared library that all microservices import for common dependencies like DTOs (data transfer objects) for API endpoints, common utility functions and other shared data structures. It quickly became apparent that some common dependencies were required across three separate services and an external library was needed to avoid code duplication which ultimately and inevitably always leads to headaches in the future when only some of many copies of a function are updated causing very hard to find inconsistencies in behaviour in a distributed system. To curb this potential problem a shared library is used and imported to all services. Once a change is pushed to this common repo it can be pulled in for all other services as it acts as a single source of truth for shared data structures and current DTOs that are

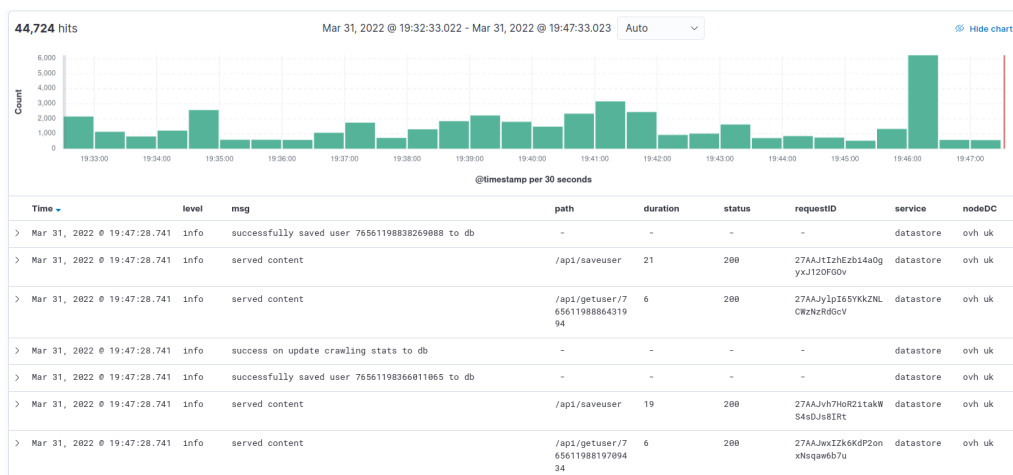
required when inter-service communication occurs which is quite often in the crawler service which heavily relies on datastore.

Infrastructure

Logging

As I will mention plenty of times during this report I first came into contact with the ELK stack during my work placement. Given this prior experience I felt that it would be a great choice for Neo. The ELK stack consists of Elasticsearch (log persistence), Logstash (log aggregation) and Kibana (log analysis and browsing) which when combined make up a powerful and efficient logging solution. The only other part of the logging infrastructure is Filebeat which is deployed as a sidecar to all microservice containers. The sidecar design pattern[16] allows for the main microservice to not concern itself with how logs are shipped to the central Logstash instance which serves to further decouple these two elements that each service offers. This means that in the future if an alternative to Filebeat was to be used then the process of swapping it out would not impact the codebase for any service.

Overall I found that once setup and initialised using the ELK stack was very pleasant. I've not even scratched the surface in terms of utilising the wide variety of features offered by Kibana but I am content with using it for what I know which is simple log analysis and browsing. Being able to filter based on the many fields such as the level, service, datacenter or region was extremely useful when debugging issues with deployed services the odd time when they happened. I felt more empowered debugging multiple cloud microservices using only Kibana compared to running them on my own computer and browsing the raw text files with the use of CTRL + F.



Web interface for Kibana

Messaging/Queue System

As an exception to the rule, I had been in brief contact with Apache's Kafka during my placement which is a very popular and performant messaging system but I instead opted to use RabbitMQ. Although Kafka itself is extremely powerful, offers more features and has more support online for troubleshooting it's simply too much for what I plan to use out of a messaging system. Therefore, RabbitMQ was by far the best contender from the remaining messaging/queue systems

that I researched. In its most basic sense RabbitMQ acts as a queue storing jobs that are waiting to be processed by crawler instances. Various other features offered by RabbitMQ serve to define itself as a great choice and manual/automatic acknowledgements of jobs is certainly one of them. Should a job be consumed from the queue and the microservice crashes RabbitMQ handles this failure and adds the jobs back into the queue as they were consumed from the queue but not acknowledged as having been completed. By manually acknowledging job completions at the microservice level after they've been fully crawled successfully this ensures that no matter what, no jobs will get lost or discarded in the system even if multiple crawler instances lose connectivity due to redeployments or crashes. All or just one crawler instance can even be redeployed as crawls are being executed and no jobs or progress will be lost. Enabling deployments that cause little to no noticeable disruption to the service as a whole is another key principle that must be ensured when architecting a system that excels in its continuous deployment capabilities.

Although I've described Kafka as being a system that's far too complex to be considered as an optimal choice for Neo I sometimes found myself thinking the same thing but to a lesser degree for RabbitMQ. Although many popular applications and examples of RabbitMQ opt for a model where there are many thousands of concurrent consumers and publishers for queues with low traffic my implementation of RabbitMQ consisted of only a handful of consumers and publishers with an insanely disproportionate amount of traffic being consumed and published by these machines. This certainly doesn't fall outside of the proposed applications listed by RabbitMQ but I often found myself disregarding many configuration suggestions proposed by online guides and tutorials. This resilience ensured by RabbitMQ is invaluable as otherwise redeployments would only be possible when no crawl is currently being executed by the system which would be a major issue.

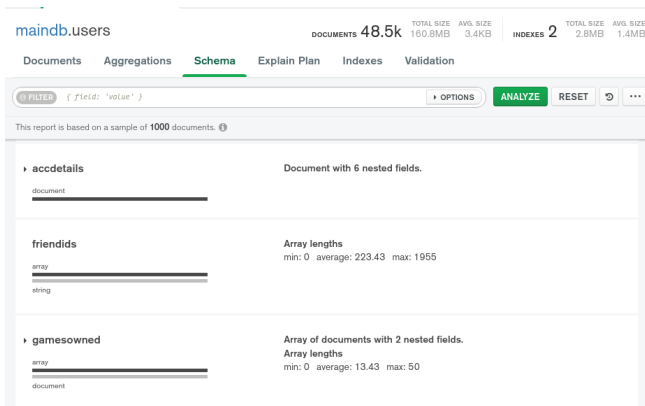
Surprisingly RabbitMQ is extremely efficient. My instance is being hosted on a simple 2GB RAM and 1 core S1-2 OVH system. Its memory usage when the queue had an unusually large backlog of 560,000 jobs only stood at roughly 500mb. RabbitMQ's ability to store some jobs on disk instead of having all jobs in memory is a lifesaver in this situation and it's also extremely efficient when it comes to CPU usage even at peak load. Initially I thought that such a piece of infrastructure as important as the database would require a much more expensive server but I was happily surprised by how performant it was.

Data Persistence

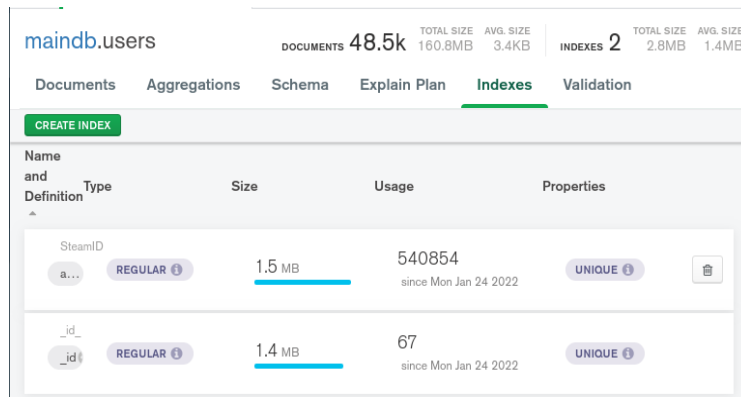
A combination of MongoDB and PostgreSQL was ultimately chosen for Neo. Initially only MongoDB was planned to be used but through some complications with the overall size of the documents used to store completed crawls. I had used MongoDB in the past and was very comfortable with writing schemas and constructing efficient queries with the use of user defined indexes so overall I had a pleasant time interacting with it. It's a noSQL based database so the barrier to entry with getting started with it is nowhere near comparable with SQL based databases which require what I would consider a formal education in order to write efficient queries and construct correct schemas. MongoDB traditionally doesn't enforce a schema on its documents but given the nature of using a strongly typed language like Go and the inherent increase in performance gained when applying strict schemas to all documents I opted to have them.

During development I had the pleasure of using Mongo's Compass GUI to interact with my deployed MongoDB instance. It's an amazing application that is superior in every way to interacting with a MongoDB instance through the lightweight CLI Mongod. It offers schema validation, insights into internal execution times for latencies and index management, all of which I made use of during development to increase performance of queries which can often be performed dozens of times per second. Creating specific indexes on document fields that are commonly used in queries helped to speed up queries and slightly decrease overall query times although these changes are usually far more apparent when databases have several times more documents than I currently have. The explain schema feature was quite interesting to use as it takes a random slice of 1000 documents from a database and analyses them in terms of minimum, maximum, average values for things like the number of friends per user. For example, I had absolutely no intuition for the average number of

friends for users or average number of games owned and I'm able to not only easily see these values in Compass but also the distributions of these values.



Explain schema feature on the users database



Index management feature on the users table. Not the usage of the custom accdetails.steamid index as opposed to the default _id index

In total four databases (equivalent to an SQL table) are used for crawling statuses, game details, shortest distance crawls and users. The crawling status database holds crawling statuses which represent the current state of all crawls. The games database is used to associate a Steam's internal game ID to its name. This database is used to avoid duplication in user's documents where instead of listing a game's name and its associated ID in the owned games section only an ID needs to be listed as the name can later be retrieved from the games database. The shortest distance crawls database holds all of the data served when a dual user crawl is executed which includes the main users involved, the shortest path if it exists and other metadata that's required. The final database holds all of the users that are crawled and if Neo is continually used will inevitably end up being the largest database out of them all as more users are crawled. Currently 13,640,000 users can comfortably be stored on the MongoDB instance without any sharding across other servers necessary.

One important database is missing from this list and that's the data that's served for individual user crawls. Although many tweaks were made to this document the overall size of larger crawls inevitably were set to exceed the internal 16MB document limit imposed by MongoDB. This is a hard limit imposed by MongoDB and the alternatives like breaking up this document into smaller chunks were considered but I saw no way that it would end in anything other than a horribly confusing overall schema with documents linking to each other in a linked list fashion. To remedy this situation a PostgreSQL instance is used for persisting the crawl data documents. PostgreSQL doesn't impose any document size limit (that is realistically an issue for any size crawl generated) and the single schema and queries used is very simple which bypasses any complex foreign key and normalisation efforts that are normally required when using an SQL based database. Each record only needs to store a crawl ID and its associated processed graph data. Using pgAdmin 4 as a GUI for my deployed PostgreSQL instance like with Mongo Compass was a pleasure to use for the few times that I had to dive in and troubleshoot issues early on when getting acquainted with working on a PostgreSQL database.

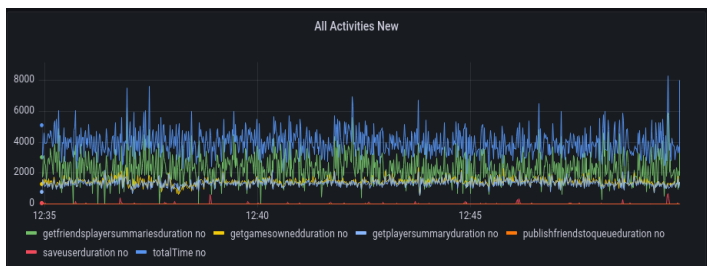
Metrics

Informative, intuitive and expressive metrics are integral to the success of any distributed system as they allow you to see at a glance the overall state of many services through many visual and numeric metrics. Grafana is a very popular metrics analytics and visualisation application that I also used during placement and had no second thoughts about using it for Neo. Only considering the end product of the many metrics boards that I've created I'd consider it a perfect fit

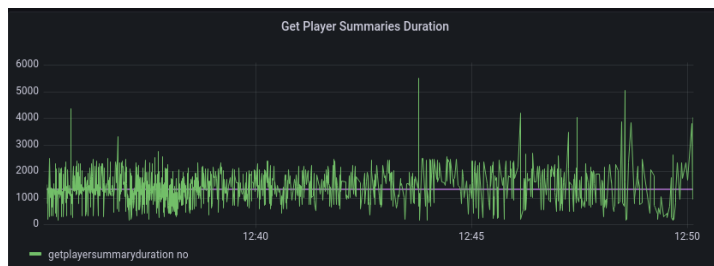
for my needs and it would take a lot for me to consider using an alternative for any future project. Grafana does not operate on its own and for Neo I chose InfluxDB as a data persistence solution as its bindings for Go are quite popular and I've written queries in the past and dreaded learning a new query language for any other alternative. Using Grafana and InfluxDB only required metrics publishing code changes in my microservices where the InfluxDB instance would be interacted with. These changes were minor and overall I found that I had little trouble getting acquainted with the process of creating buckets (similar to SQL tables), writing specific data to its associated bucket, writing Flux queries (InfluxDB's query language) and then formatting and organising graphs constructed with these queries in Grafana.

A considerable amount of thought was put into the datastore and crawler metrics pages in relation to what data I wanted to be able to view at a glance. In 'Sight Reliability Engineering'[18] there are two key metrics in particular that are specifically mentioned that I believe should be monitored if anything at all is monitored. Those are the latencies (P50-P99 and the distribution of those requests) and the traffic to any particular endpoint. With the P50, P90 and P99 (50th percentile, 90th percentile and 99th percentile) latencies at a glance a numerical value can be given that tells developers that most of the time requests wait for x, but 10% or 1% of the time these noticeably longer latencies are being recorded. In systems where network calls lead to further network calls in a waterfall fashion one slow call in the stack causes widespread issues as it is similar with how in security how you're only as strong as your weakest link but in this case you're only as fast as your slowest request. With a visual graph showing the distribution this serves as a powerful visual aid when the numerical figures seem too abstract. After creating this page and getting a solid foundation of metrics to log for each endpoint I largely copied this format for metrics from the crawler service.

For pieces of infrastructure such as MongoDB and RabbitMQ I was able to download premade dashboards online for monitoring them with the help of exporter applications to extract metrics from them using Prometheus and Mongo-Exporter. These dashboards were of great use to me and enabled me to spend my time working on more important things instead of writing dozens of queries to visualise the many core metrics that required monitoring for these two applications.



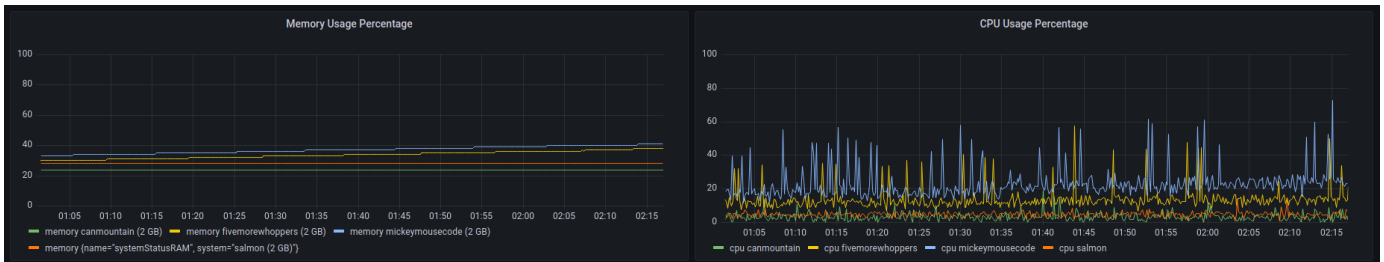
Crawler graph displaying the total durations for each activity executed during the processing of one job



Duration of calls to GetPlayerSummary (Steam API) in crawler. Durations above the purple line indicate they had to wait to retrieve an API key to avoid rate limiting



Typical series of metrics that are shown for each endpoint in all microservices. A scatter plot of all latencies, P50/90/99 latencies and a latency distribution histogram



System stats metrics. Track CPU and memory usage for all microservices



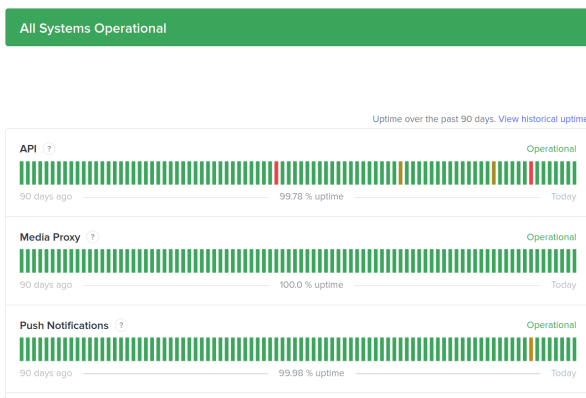
Key MongoDB metrics that I follow on my premade MongoDB board. The query operations board is the most important as it illustrates the current total load on the database



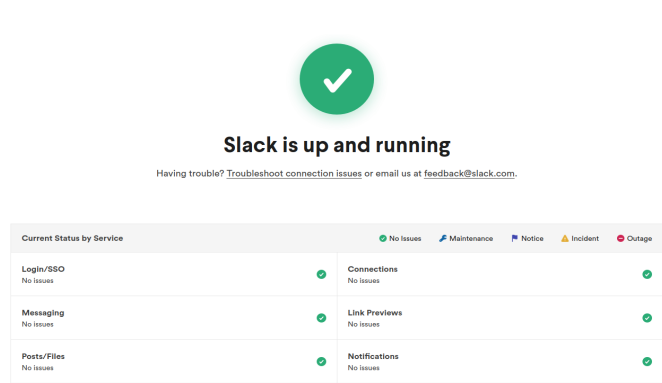
Key metrics on RabbitMQ premade board. Input/s is jobs being published per second. Output/s is the jobs being completed per second, this indicates the total throughput of the system in terms of jobs

Uptime Monitoring

Whenever one of my favourite applications like Discord seems to be having connectivity issues I always check their standalone uptime status site to see if the system itself is having issues or if it's just my connection. I found that Discord status site's graphics for the uptime percentage overtime was quite nice and a great visual indicator as to the overall stability of the system overtime. They also display what appears to be an overall API response time chart plotting response time vs time which I also find interesting to look at as it fluctuates over time. Although I'm not a user of the product, Slack's uptime status site is great in its ability to express the status of their services in simple terms. I looked into various open source solutions for creating your own status page but most either were not self-hosted and were not flexible enough in their configurations or had amazing interfaces but required manual updates for the data.



Discord status uptime site[11]

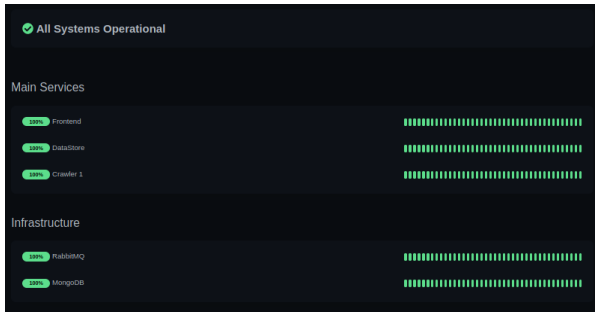


Slack status uptime site[10]

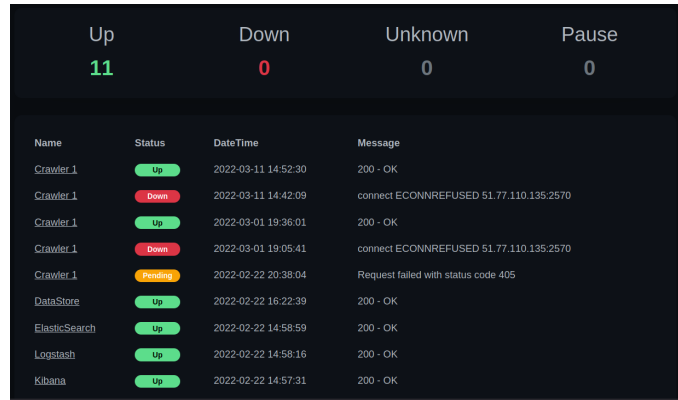
I then came across Uptime Kuma. It's self-hosted, allows for health checks through TCP, HTTP(s), ping and even DNS. The flexibility is amazing and simple authentication, returned content validation and automatic retries are all integrated in an intuitive manner when setting up new monitors. Since it's self-hosted there is no implicit rate limit set on the intervals that health checks can be executed at and there is no limit to the amount that can be run at any given time. It's also got a huge variety of third party integrations to applications like Discord, Microsoft Teams, Slack and Email where updates on changing availability of monitored services can be sent to in real time. I receive push notifications from a discord channel where Uptime Kuma sends messages should a service ever crash to ensure that these incidents do not happen silently.

In addition to checking the availability of a particular target it also tracks the most recent, average over a 24 hour and 30 day span response time alongside its total percentage uptime over the last 30 days.

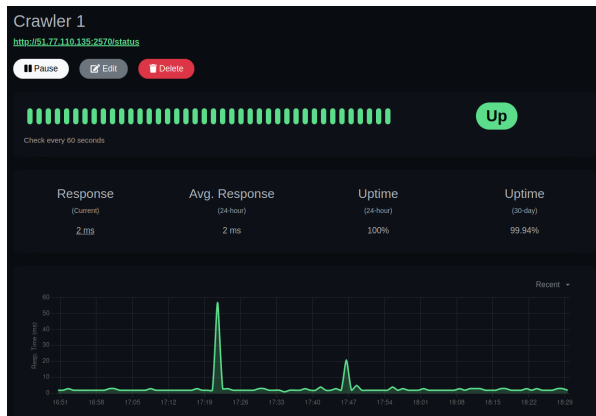
The simple status page view is linked from the navigation bar on the frontend alongside a link to the Github repository. Overall I'm very pleased to have stumbled across Uptime Kuma and I think it makes an invaluable contribution to the overall presentation of Neo. The uptime percentage alone provides a level of assurance that the stability of the services is of utmost importance as I'm committed to be open and transparent about them even if incidents occur and uptime suffers as a result.



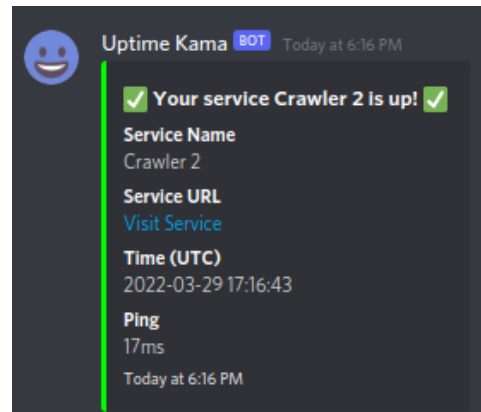
A portion of the user' view of the status page



Admin view of the change in status of all monitors



Admin view of the history of a particular monitor



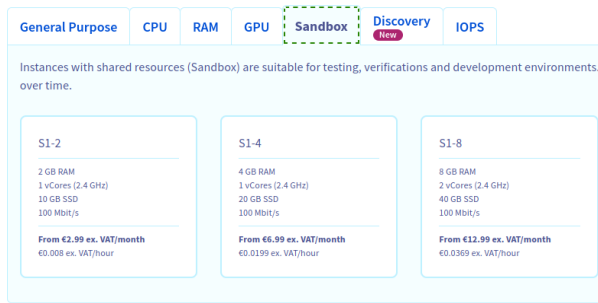
An example notification received on Discord from Uptime Kuma

Hosting

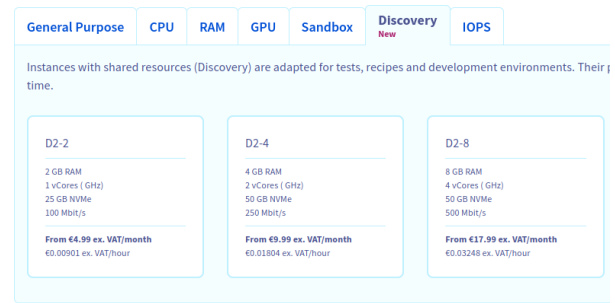
Through my research I found OVH, a French cloud hosting provider who offered plenty of tiers that would suit my needs in terms of compute power and price and I had read many stories online from happy customers to show that they've got a solid track record of not disappointing their customers.

Shown below was the main selection of instances that I used for Neo. My own microservices have a very small footprint and are able to be hosted on S1-2 instances (2GB ram, 1 core) with plenty of room to spare even under an artificially induced heavy load during load testing which I will cover in more detail in the evaluation section. The logging

infrastructure is especially critical and therefore I opted to use a D2-4 instance (4GB ram, 2 cores) for hosting as I had often created horrendous bottlenecks for the entire project hosting the stack with just one gigabyte of ram available.



OVH's sandbox tier instances. S1-2 tier instances were used for all microservices



OVH's discovery tier instances. A D2-4 instance was used for the ELK stack instance

Oracle provides an extremely generous free tier that I took full advantage of. In total I was able to leverage 24GB of RAM and 4 (slightly underpowered ARM based) cores on a single instance which hosts both databases to ensure that resource starvation does not occur no matter the load on the databases. Their approach is more closely aligned with AWS so setup was slightly more involved as they take a much more strict approach to overall security on instances. A result of this was that all ports that I wanted to use to communicate with on the instance had to be manually whitelisted to enable access from remote instances

Service	Cloud Provider	Compute Power	Network Speed (Mbp/s)
Frontend	OVH	2GB RAM 1 core	100
DataStore	OVH	2GB RAM 1 core	100
Crawler 1	OVH	2GB RAM 1 core	100
Crawler 2	OVH	2GB RAM 1 core	100
Uptime Kuma	OVH	2GB RAM 1 core	100
Prometheus, RabbitMQ	OVH	2GB RAM 1 core	100
Grafana, InfluxDB, Mongo Exporter	OVH	2GB RAM 1 core	100
ElasticSearch, Logstash, Kibana	OVH	4GB RAM 2 cores	250
MongoDB, PostgreSQL	Oracle	24GB RAM 4 cores	250

I also purchased a domain for the project, neofyp.com. Remembering IPV4 addresses is quite hard and inconvenient for people so being able to just remember neofyp instead of one is what I'd consider the bare minimum for any website. In addition to the main domain for the frontend there exist several subdomains for other components such as datastore.neofyp.com, logs.neofyp.com and status.neofyp.com that are often visited through web browsers. The process of purchasing a domain and updating the DNS records on the provider's website is extremely simple and in most instances I was able to navigate to the new domain/subdomain within minutes which was great given that sometimes this process can take up to a few hours depending on how fast the appropriate DNS servers are updated. One important thing to note is that status.neofyp.com (which is Uptime Kuma) is hosted on its own instance isolated from all faults that

could be caused during maintenance/hardware issues on any other Neo service. This makes the situation of certain services going down which then as a consequence causes the status page to also go down which invalidates the basic need of a status page to be used when there seems to be a fault for a service.

Testing

It cannot be understated how integral testing is in modern software. Robust, efficient and adequate test suites are required for all projects that do not wish to cause their contributors endless hours of fruitless debugging. In an ideal test suite a broad range of testing methods should be undertaken ranging from service tests to integration/end to end tests. To say that I relied heavily on testing during development would be an understatement. However, placing so much trust in tests isn't always a wise decision because simply having tests is not enough. Tests must adequately cover a variety of cases and reliably test all applications of codebases to ensure that future additions do not cause regressions or detrimental unforeseen consequences to surrounding features. Through diligence and discipline in writing comprehensive testing suites I'm able to with ease count on one hand the number of times that my testing failed to catch regressions and other major bugs that were not previously covered by tests.

The main difference between service tests and integration tests is that only integration tests are allowed to interact with the 'outside world'. Michael Feathers lays out what I consider to be the most important rules that must never be broken when writing service tests "A test is not a unit test if: if talks to the database, it communicates across a network, it touches the file system, it can't run at the same time as any other of your unit tests, You have to do special things like things to your environment (such as editing config files) to run it" [9]. Service tests need only to test the actual functionality of a piece of code, not the integration with other pieces of code (that already should have their own existing service tests) across networks or filesystems. Service tests must be reliable and allowing access to unreliable dependencies such as file systems or networks goes against everything that service tests stand for. Testing the integrations between pieces of code is the task for integration tests and integration tests alone. By relying on unreliable integrations between the tested code the probability for failure inherently increases which to a degree is always expected with integration tests. Although this is not to say that integration tests are not expected to pass nearly all of the time. Every effort should be made to rectify any issues if integration tests are failing on even a semi-regular basis as this indicates that the same issues must be occurring in production seemingly without any acknowledgement or even plans to fix them.

I will add that while my development usually consisted of implementing a new feature and then adding testing for it I found it useful in some situations to opt for a more test-driven development approach. Test-driven development places heavy emphasis on comprehensive test suites as before any new code is implemented tests are constructed which act as the minimum viable conditions that must be met. I often opted for this approach when implementing minor changes to important processes that were already implemented as it gave me a much greater sense of trust in myself to not introduce regressions that would go unnoticed without adequate testing in place.

Given the nature of a distributed system and the strict requirement for service tests that forbids them from communicating with anything such as a filesystem or network, careful thought must be given to the codebase to enable thorough and comprehensive testing without breaking these rules. In a general use language such as Java the practice of dependency injection is extremely simple and commonplace. Dependency injection in the context of testing allows for dependencies or methods to easily be replaced by what are known as stubs/mocks. In a service test anything that isn't directly part of the specific function being tested is placed with stubs that hijack any invocations that may occur and return a specified result that's manually written in the test. By following the rules set out by Feathers testing a system that interacts with other services over a network is impossible.

However, in go practising dependency injection requires a slightly more hands on approach as what I'd consider to be moderate changes to a codebase must be made although the end certainly justifies the means in this context. Essentially

all functions that will require some kind of stubbing for testing are added as methods to a struct which is similar to associating methods to an object in Java. In all services this struct is referred to as the Controller. During testing a mock version of this controller is used where various methods are stubbed to return exact values instead whereas the real application does not have these methods stubbed.

```

type CntrInterface interface {
    // MongoDB related functions
    InsertOne(ctx context.Context, collection *mongo.Collection, bson []byte) (*mongo.InsertOneResult, error)
    UpdateCrawlingStatus(ctx context.Context, collection *mongo.Collection, crawlingStatus common.CrawlingStatus) (bool, error)
    GetUser(ctx context.Context, steamID string) (common.UserDocument, error)
    GetCrawlingStatusFromDBFromCrawlID(ctx context.Context, crawlID string) (common.CrawlingStatus, error)
    HasUserBeenCrawledBeforeAtLevel(ctx context.Context, level int, steamID string) (string, error)
    GetUsernames(ctx context.Context, steamIDs []string) (map[string]string, error)
    InsertGame(ctx context.Context, game common.BareGameInfo) (bool, error)
    GetDetailsForGames(ctx context.Context, IDList []int) ([]common.BareGameInfo, error)
    SaveShortestDistance(ctx context.Context, shortestDistanceInfo datastructures.ShortestDistanceInfo) (bool, error)
    GetShortestDistanceInfo(ctx context.Context, crawlIDs []string) (bool, datastructures.ShortestDistanceInfo, error)
    GetNMostRecentFinishedCrawls(ctx context.Context, amount int64) ([]common.CrawlingStatus, error)
    GetNMostRecentFinishedShortestDistanceCrawls(ctx context.Context, amount int64) ([]datastructures.ShortestDistanceInfo, error)
    GetTotalDocumentsInCollection(ctx context.Context, collectionName string) (int64, error)
    // Postgresql related functions
    SaveProcessedGraphData(crawlID string, graphData common.UsersGraphData) (bool, error)
    GetProcessedGraphData(crawlID string) (common.UsersGraphData, error)
    DoesProcessedGraphDataExist(crawlID string) (bool, error)
}

```

Associated methods of the Controller in the datastore service. These are all of the methods in the service that interact with a network and subsequently must be stubbed for service tests

One of the smallest yet most common differences to my project that I picked up during my work placement was the practice of having test function's names describe exactly what's being tested, which often is the length of an entire sentence. Normally having function names that are this long could only be considered to be bad practice but for tests I now see it as the complete opposite. This minor detail was quite confusing to me at first glance but quickly became apparent as to why it was done when I had to trawl through text logs of failed test suites and instantly, just from reading the failing test name I knew exactly what had failed without having to dig any further. Another great design pattern that I picked up is the arrange, act and assert (or given, when, then) testing pattern that's used to visually divide a test into its three main components. Initially all the dependencies and stubs are created in the arrange/given stage. Then the code being tested is actually called in the act/when stage. Finally assertions are made on the returned values in the assert/then stage.

In addition to service tests integration testing is used. Integration tests are not bound by the same rules as service tests and do interact over a network with other components. In the case of Neo integration tests spin up a local version of the service and any external services called are the deployed production instances. Typically with major software products there exists two separate environments; integration and production. They would both be as close as possible to each other in everything aside from the actual data being stored in the databases and the users of each environment. In this setup integration tests target other services that are deployed in integration and any testing data that is created is written to integration databases that does not affect the production application. However, with this approach comes a lot more responsibility for the

```

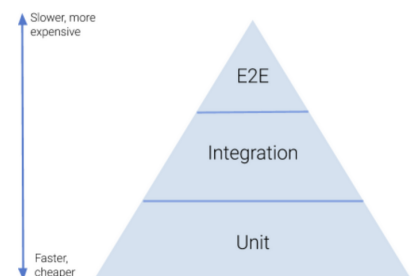
func TestGetUserReturnsInvalidResponseWhenGetUseFromDBReturnsAnError(t *testing.T) {
    // Initialise the server and mock controller that will have
    // some of its methods stubbed for the purpose of this test
    mockController, serverPort := initServerAndDependencies()
    getUserError := errors.New("couldn't get user")
    // When the GetUser method on Controller is invoked in the form:
    // GetUser(anything, anything of type string)
    // return an empty userDocument and the specified error getUserError
    mockController.On("GetUser",
        mock.Anything,
        mock.AnythingOfType("string")).
        Return(common.UserDocument{}, getUserError)

    authHeader := http.Header{}
    // Set the Authentication header to allow for access
    // to this authentication-required endpoint
    authHeader.Set("Authentication", os.Getenv("AUTH_KEY"))
    res, err := util.GetAndRead(
        fmt.Sprintf(
            "http://localhost:%d/api/getuser/%s",
            serverPort,
            testUser.AccDetails.SteamID),
        []http.Header{authHeader})
    if err != nil {
        log.Fatal(err)
    }

    // Assert that conditions have been met. In this case that
    // the returned HTTP body contains the beginning of the default
    // error message "Give the code monkeys this ID:"
    assert.Contains(t, string(res), defaultPanicErrorMessageStarter)
}

```

An annotated example service test. Notice how the act, arrange and assert design pattern is followed (emphasised in this specific example for clarity)

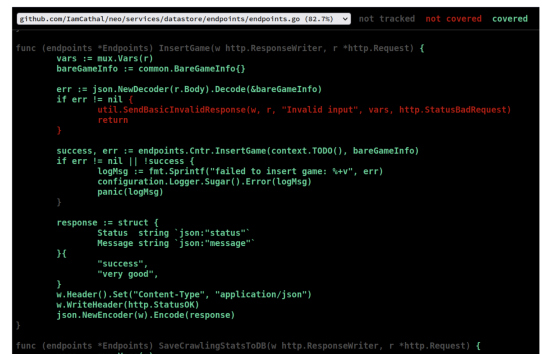


The Testing pyramid [Leung]

developers: a replica set of all services only used for testing must be maintained.

My testing mindset was based on the popular testing pyramid methodology. This states that there should be a significant amount more fast and simple unit/service tests in comparison to much more complex and 'flakey' integration tests. Integration tests are often considered flakey as without any code changes they are not always guaranteed to pass as they interact with external dependencies like filesystems and networks whereas service tests never do. With regards to the runtime of test suites in terms of service and integration tests it's the inverse of the 'cost'. Service tests should be easily run as fast as possible to ensure that it's not a pain for developers to stop and quickly test their code. As the time required for running test suites increases the tendency to not run them also increases which ultimately is never a good thing. Integration tests often require heavy initialisation of real dependencies and are slowed down by network delays for requests unlike service tests which ultimately mean they're runtime is significantly longer. However, by prioritising the quick service tests by running them first overall wastes less time when failures are caught by them which is a good thing for everyone. For testing I always try to positively and negatively test. Positive and negative testing aims to validate that an application can handle input that is desired and input that is not desired[8]. As a consequence of this approach to testing the coverage usually ends up being quite high and inversely by trying to maximise coverage positive and negative testing usually happens in some form (if the code itself is well written and handles undesired outcomes/inputs).

Coverage can mean everything or nothing when it comes to writing comprehensive tests. I believe coverage is important but is certainly not the only metric that should be followed in measuring the value of test suites. For every new feature, no matter how large or insignificant, I aimed to have good coverage not only in terms of the percentage of lines covered but also for the real world situations that the feature would face. There were some exceptions to this rule with particularly hard to test features like websocket endpoints or retry mechanisms when interacting with the actual Steam API. While coverage is not tracked publicly on the repository as it is with some projects I did personally monitor it and increase coverage when I felt it was necessary.



```
func (endpoints *Endpoints) InsertGame(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    bareGameInfo := common.BareGameInfo{
        err := json.NewDecoder(r.Body).Decode(&bareGameInfo)
        if err != nil {
            w.WriteHeader(http.StatusBadRequest)
            return
        }
        success, err := endpoints.Ctr.InsertGame(context.TODO(), bareGameInfo)
        if err != nil || !success {
            logMsg := fmt.Sprintf("failed to insert game: %v", err)
            configuration.Logger.Sugar().Error(logMsg)
            panic(logMsg)
        }
        response := struct {
            Status string `json:"status"`
            Message string `json:"message"`
        }{
            "success",
            "very good",
        }
        w.Header().Set("Content-Type", "application/json")
        w.WriteHeader(http.StatusOK)
        json.NewEncoder(w).Encode(response)
    }
}

func (endpoints *Endpoints) SaveCrawlingStatsToDB(w http.ResponseWriter, r *http.Request) {
```

An example of the coverage report indicating in green the lines that were executed and in red the lines that were not

The frontend service is the exception with regards to extensive testing of client side Javascript code. The reason for this is that in order to be able to test Javascript code, expensive and heavy dependencies must be used to emulate a web browser which has a runtime capable of executing Javascript. In the end I decided that testing this mostly CRUD and utility based code was not worth the investment required. However, for the very few endpoints that are not used purely to serve static content/templates there is adequate testing.

Security

Unfortunately given the current landscape with cybersecurity any internet facing application is forced to protect itself against an ever growing number of attacks from bad actors from all over the globe. By default access to all of the servers used for hosting through SSH is restricted to public key authentication as the only method of access. This renders brute force attempts at guessing passwords to gain access to them impossible. The only method of gaining entry to any of these servers is for an attacker to somehow exfiltrate my private key which is only stored on my own computer. This section will be detailing the myriad of measures taken in both my own services at a code level to the securing of infrastructure as a whole.

All user input is considered malicious until it has been validated and sanitised. This happens on both the client and server side for one very important reason. Validating and sanitising input on just the client side might stop invalid inputs from users who are not malicious but attackers will quickly craft requests directly to the server without having to go through the client side checks which then invalidates any filtering executed on the client side. Robust testing of these

validation and sanitisation methods through a range of malicious inputs must also be taken to ensure that you're comfortably able to rely on them, simply having good coverage for service tests on these validation functions isn't enough. In addition to this no input is rendered directly to the page in ways that would allow attackers to execute any kind of Javascript injection.

The nature of the application itself simplifies the security requirements as no user management is handled and no sensitive data is ever stored or handled. The only pieces of data that could be considered sensitive are the Steam API keys that are used by crawler instances but users would never be able to access them by any means. However, there are other issues that must be taken care of in relation to security and that stems from the fact that while the Steam API requires their own issued API keys for access Neo does not. Therefore I had to pay careful consideration to the fact that I should not be allowing Neo to become a proxy for the official Steam API for those who either had no API keys themselves or wished to abuse the use of my own. While Neo technically can be used as a very an extremely rudimentary proxy for retrieving some data from the Steam API it would not be worth anyone's time to try due to the measures that I've taken specifically to stop this. A specific example of this is the get crawling user endpoint in datastore. For the crawling page basic account details are displayed for the target user, these details are taken directly from the main users database but not through the typical get user endpoint. Instead, the frontend can only request individual users through the get crawling user endpoint when a crawl (where they are the target) is currently being executed. This allows for the retrieval of these account details without compromising on the specific design patterns implemented to make any use of Neo as a Steam API proxy as tough as possible.

As few API endpoints as possible do not require authentication to further discourage users attempting to use Neo as a proxy to the Steam API. Many endpoints are designed to retrieve data that is saved directly from the Steam API and are only able to be accessed when a secret is included in the authentication header. This secret is known to the crawler instances that require access to authenticated endpoints (in the datastore service) and there is no chance of it leaking. I considered using some sort of JWT (Javascript web token) solution which is a popular authentication approach that involves tokens that are signed with a secret or public/private key pair. However this is most definitely overkill for the simple task of accessing endpoints and adds unnecessary complexity. The chosen secret is of sufficient length (more than 32 alphanumeric and symbol characters) so brute forcing entry is not possible. Unsuccessful attempts to access authorization-required endpoints are flagged in the logs for monitoring purposes.

Another practice that I follow religiously throughout all of my projects that are internet facing is returning as little information as possible through error messages. For example, in a situation where a query causes an unrecoverable failure from within the application it's very important for the developer to see exactly what that error message says but to allow a user to see this error message is asking for trouble. Instead of informing the user that this carefully maliciously crafted input failed to pass the table schema in a database the user should only be informed that their input was invalid. By returning a set list of general error messages end users of the application cannot gain any information that might be leaked from reading actual error messages like perhaps the specific version of a database that's being used or other small pieces of information that should not be known by users. With this practice it's important to emphasise that the actual error messages are not being thrown away, they're logged in full and it's only the user who sees this vague error message. Another popular example of this is informing the user during a login attempt that although a password might be correct the associated username is not. This is wrong on so many levels and reveals more to the user than they should ever know. Therefore simply informing the user that either the username or password (but not specifying which) is incorrect is always the better option.

On a slightly related note I specifically chose to not configure the microservices to automatically restart if they should ever crash. Docker, and as an extension Docker Compose offers functionality to restart any containers no matter which exit code is returned when the main process running inside of a container exits. The reason behind this choice is due to the fact that if an attacker caused an error that crashed a microservice and repeatedly kept on doing this then the cycle of restarts and crashes is nearly something that I'd consider to be worse than a complete outage. During this time I would also be receiving an avalanche of notifications from Uptime Kuma informing me every time the service lost and

regained connectivity. This situation could also just as easily (and more likely) be caused by misconfiguration on my behalf where a microservice fails to initialise all its dependencies which leaves it unable to function normally and is subsequently stopped. This would cause an even faster cycle of restarts as the initialisation process has been optimised to take as little time as possible.

Documentation

Go has some strong semi-native support for documentation through both the official Go language server Gopls and go.pkg.dev. One of the many features of Gopls is extracting documentation in the format of comments in Go code among many other helpful features. The Go team emphasise that documentation must be “well-written and accurate, but it also must be easy to write and to maintain”[5] and to ensure that Go provides users with the tools to make that a reality they created Godoc, a tool that parses comments in source code and renders rich and informative HTML documents that can be viewed on pkg.go.dev for anyone to browse. This enables me as a developer to only write comments in my code for functions and data structures and automatically get this documentation rendered as a web page and hosted for me.

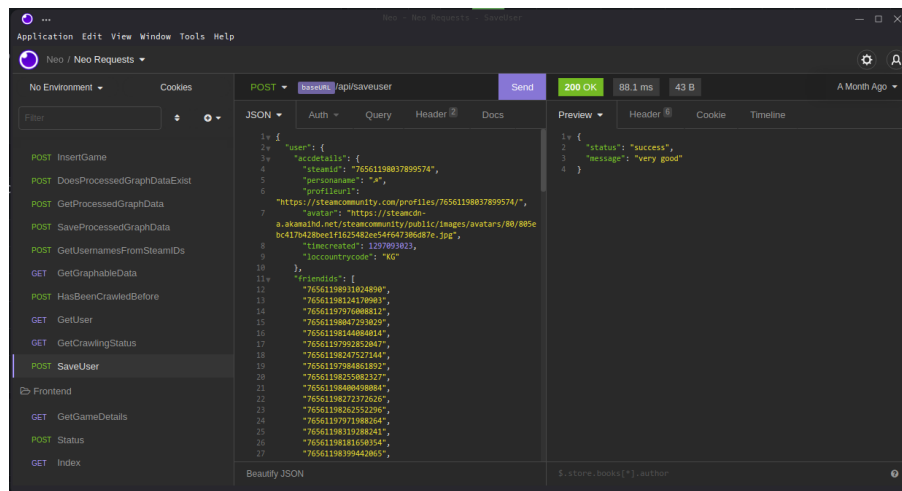
```
// SaveUserDTO is the input schema for saving users to the database. It takes
// the original crawl target user (that initially caused this crawl) and the
// current user to be saved
type SaveUserDTO struct {
    OriginalCrawlTarget string `json:"originalcrawltarget"`
    CrawlID             string `json:"crawlid"`
    CurrentLevel        int    `json:"currentlevel"`
    MaxLevel            int    `json:"maxlevel"`
    User                 common.UserDocument `json:"user"`
    GamesOwnedFull      []common.GameInfoDocument `json:"gamesownedfull"`
}
```

```
type SaveUserDTO
type SaveUserDTO struct {
    OriginalCrawlTarget string `json:"originalcrawltarget"`
    CrawlID             string `json:"crawlid"`
    CurrentLevel        int    `json:"currentlevel"`
    MaxLevel            int    `json:"maxlevel"`
    User                 common.UserDocument `json:"user"`
    GamesOwnedFull      []common.GameInfoDocument `json:"gamesownedfull"`
}
SaveUserDTO is the input schema for saving users to the database. It takes the original crawl target user (that initially caused this crawl) and the current user to be saved
```

Source code for the SaveUser object returned by the datastore /saveuser endpoint

Pkg.go.dev web view of the SaveUserDTO object

In addition to documentation written for the internals of Neo services I also used Insomnia in some ways as API documentation. Insomnia is an open source alternative to Postman, an API client used for testing REST, GRPC and GraphQL APIs. Creating rich documentation for all 37 endpoints across all of my services would be a massive undertaking but with Insomnia I can manually interact with endpoints and also use it as some form of documentation as to how endpoints function with regards to inputs and outputs. Many of the endpoints are quite similar to each and only differ based on the input data that they receive and return. However this isn't very useful for myself as I can read the source code for endpoints and quickly determine how to interact with it as I've written so many of them.



An example of manual testing of the save user API endpoint through the Insomnia client

Development

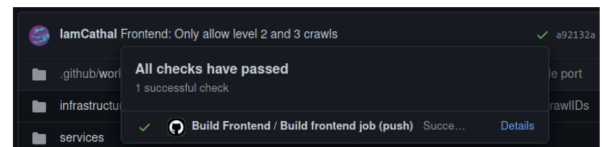
I've used Git for years on all my personal projects and college assignments and I couldn't see myself using anything else for Neo. Although I chose git like I had for all my previous projects I stuck to a strict commit formatting system that would enable me to quickly track the overall changes happening to each service that I was committing to. All commits to service x would be in the format "X: bump common and Go mod tidy" which enabled me to later extract this metadata easily from all commit messages and display them in this report to show which services had the most commits during the course of development.

I also opted to not use feature branches during development and chose to push everything to the master branch. Normally on a project with multiple contributors this would never be a good choice but for my workflow it was a good decision. The rationale behind using feature branches is to keep master "clean" and without half-baked changes as it's often the branch that's used for deployments. My master branch is used for deployments but deployments are not triggered automatically on pushes to master. Deployments are manually triggered only when I've completed and manually tested a feature locally, so if the master branch is failing the test suite it's not going to affect production. Therefore, although it's good practice my master branch isn't required to always be passing the test suites.

CI/CD (Continuous Integration And Continuous Deployment)

Wikipedia states that CI/CD "bridges the gaps between development and operation activities and teams by enforcing automation in building, testing and deployment of applications"[6]. Efficient and automated testing alongside simple deployments are the core principles that CI/CD aims to ensure.

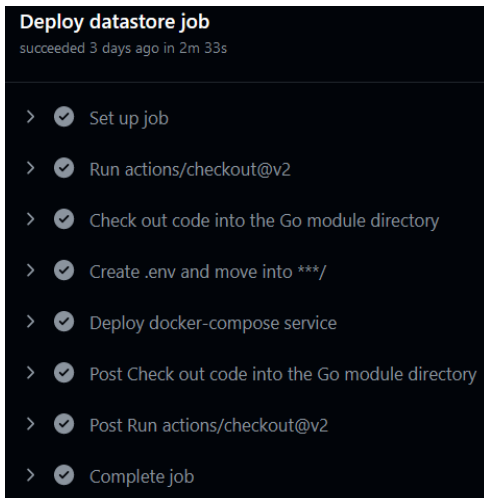
Automated testing is implemented through Github actions workflows after commits are pushed. Actions allows for test suites to run on remote servers running any operating system architecture ranging from from mac os, windows to linux. For each commit the changes are analysed and each main service where any changes have occurred has its automated test and build suite triggered. This saves unnecessary test suites being run for services where no change has been committed. Should the build fail an email is immediately sent to me and if the build succeeds then no further input is required. Automated testing is integral to the success of any software project as it's inevitable that either through negligence or forgetfulness that at some point a serious bug could make it through to production if tests were not run even just once. I also set up all service tests to run before every commit was made by making use of git's pre-commit hooks. Before git itself commits my changes it runs service tests and if the suite does not pass then it's not committed unless it's explicitly told to commit regardless of the test suite status. This is a useful feature but pre-commit hooks are not shared in a repository so the odd time when I was developing on my laptop I wasn't able to use the hook unless I explicitly added it again myself.



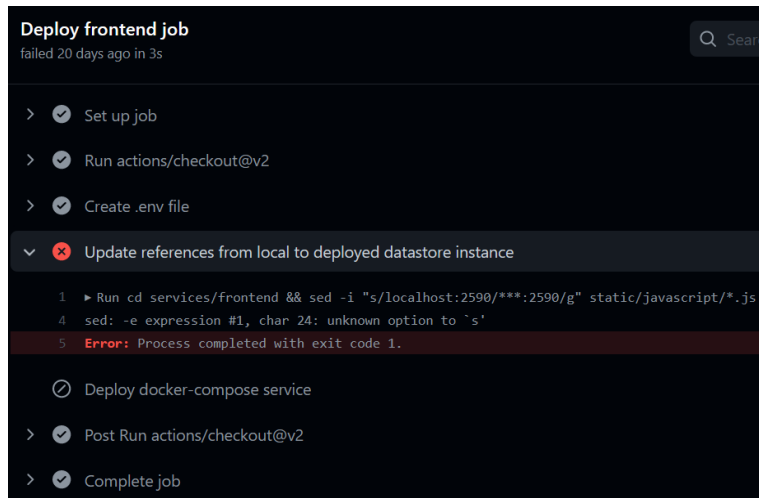
Automated frontend test suite pass

With my approach in development with regards to only using one branch for development, having automated deployments would not be very useful. However, simple one-click deployments are enabled for all services through Github actions. Again, with deployments I'm made aware of failures immediately through either the web interface, email alerts or Kuma (uptime monitoring service). Github actions displays a graphical list of all steps in a workflow to quickly show failures in certain steps and the logs taken directly from the build server. This is particularly useful as deployments are being executed across multiple remote servers instead of my own. Github develops self-hosted runners, software that allows github actions to interact with remote servers for building and testing. The service is run on all servers in the background and in the workflow files the specified servers are targeted i.e for deployments of the frontend service the unique ID of the frontend server is used. With self-hosted runners managing an effective continuous deployment

operation is seamless and I rarely have to invest time into troubleshooting my deployments for issues that aren't caused by the services themselves.



A successful build



An unsuccessful build which clearly shows the failed step and the reported error message from the logs

To enable quick and simple deployments all microservices utilise .env files to store secrets for things like InfluxDB bucket tokens, Steam API keys, database credentials and other configuration details that should not be hardcoded. By doing this changing minor details like database credentials the only variables that have to be updated are in the Github secrets manager and it's distributed to all services on their next deployment. This also serves to hide any sensitive keys or credentials needed by the application from anyone reading the public source code on Github. This also means that all a service requires is this single configuration file, there's no sticky sessions or local caches that must be built up over time to function properly. Therefore deployment scripts are quite simple as the only steps that must be taken are as follows: pull the most recent commit, create a local .env file and concatenate the required secrets into the file, execute the docker file which builds the code and then deploy the services with docker compose.

All services make use of multi-stage docker builds to enable the size of the containers to shrink by a factor of roughly 10x[20]. Typically what would happen is a rather large base image would be used and on top of this the go runtime would reside which means that the final image size can often be as large as 700MB, for comparison the current crawler service image size stands at 78MB. Docker builds images in layers and although the large base image might be required during building it is not required at runtime. Therefore with a multi-stage build approach before the final image is created the binary that has just been built is extracted and that itself is the entire image and the large base image is discarded. No shell exists inside of this container and the go application can function normally with a drastically reduced image size. In this case shipping these binaries is not a concern so this reduced image size isn't as advantageous as it could be but it's a great approach to take regardless. Another advantage of this is that should an attacker be able to exploit the application to gain shell access there will be no shell for them to access.

Downtime of a service when it's being redeployed is only as long as it takes to initialise. I'm managing my builds and deployments by building the new service in the background and only once the container has been built is the now outdated running container killed. This enables services to have roughly 1-2 seconds of downtime instead of stopping the currently running container while the new container is being built. Reducing the initialisation time of services was also a challenge but significant progress was made by parallelizing the various tasks that needed to be initialised but did not depend on each other, similar to the task group work carried out on the crawl loop. With the crawler service this reduced the time from initially starting the service to the service being able to handle requests from roughly 6 to under 2 seconds reliably.

Nearly all configuration details that should be configurable are set through environment variables to enable quick tweaks to be made without endless code changes being required wherever a hardcoded string is mentioned. To further aid in this design pattern each microservice has a list of environment variables that they expect to be set and a further list of service specific environment variables that need to be set. In the base root directory for all services the general environment variables are listed that all services rely upon. Common things such as tokens for accessing metrics buckets and the secret authentication passphrase are listed here. In addition to this service specific environment variables are listed in the base directory of each service detailing the specific environment variables that must be set for this particular service. However, these lists do have many variables that must be set and I sometimes forgot to include all of them when setting up a new service and therefore in each configuration submodule for each service there is a utility function that asserts that not only the default environment variables that all services must set have been set but the service specific variables are also passed in. This ensures that unexpected behaviour such as metrics not being uploaded because the token does not exist which happened before I implemented this utility function does not happen again.

Services

All neo microservices

Configuration

All services must have the following environment variables

Variable	Description
API_PORT	The port to host the application on
LOG_PATH	Where to write logs to
NODE_NAME	Unique name for this instance
NODE_DC	Datacenter for this instance
SYSTEM_STATS_BUCKET	Bucket name for system stats metrics
SYSTEM_STATS_BUCKET_TOKEN	Bucket token for system stats metrics
ENDPOINT_LATENCIES_BUCKET	Bucket name for endpoint latencies
ENDPOINT_LATENCIES_BUCKET_TOKEN	Bucket token for endpoint latencies metrics
ORG	Org name for grafana
INFLUXDB_URL	Full URL for connecting to influxDB
AUTH_KEY	Authentication key used by services for authorized only endpoints

General service environment variable list

Datastore

Build Datastore passing Deploy Datastore passing

Datastore is the thin client that sits in front of the database

Configuration

Datastore expects the following variables to be set in .env

Datastore specific

Variable	Description
MONGODB_USER	MongoDB account username
MONGODB_PASSWORD	MongoDB account password
MONGO_INSTANCE_IP	IP for the MongoDB instance
DB_NAME	Database name for the stored data
USER_COLLECTION	Collection name for the user data
CRAWLING_STATS_COLLECTION	Collection name for the crawling stats
SHORTEST_DISTANCE_COLLECTION	Collection name for shortest distance info
POSTGRES_USER	Username for postgres worker account
POSTGRES_PASSWORD	Password for postgres worker account
POSTGRES_DB	DB name for postgres saved graphs table
POSTGRES_INSTANCE_IP	IP for the postgres instance

Datastore specific environment variables

Languages

All microservices are written in Go and a standard HTML, CSS and (vanilla) Javascript approach is taken for the web interface. Given the fact that I've been using Go for roughly three years now and that the language itself was specifically developed by Google to be used in programming for large and scalable servers it's a great match. Aside from my comfort with using the language there are a plethora of native drivers for my planned pieces of infrastructure like RabbitMQ, MongoDB and InfluxDB and many great examples online of microservice patterns using the language with minimal dependencies required no less. Go strives to be simple and doesn't encourage reliance on a myriad of external dependencies unlike the ecosystems that exist around NodeJS and Java. Testing and running microservices is as simple as using "go test" and "go run". No external dependencies required which is a breath of fresh air after having worked with Spring powered Java microservices where I had no clue how the multiple layers of abstractions worked to host a simple service. An added benefit is Go's ability to build binaries for any platform and that single binary is enough to run the application. No external dependencies need to be installed on remote servers, as long as the binary has been targeted for the correct architecture (macos, windows or arm) then it works seamlessly. Overall I don't see myself using any other language in its place and many of the downfalls of the language that some developers might point out like

strong typing and lack of generics I see as necessary steps that were taken in the pursuit of a language that strives to differentiate itself from the rest instead of trying to imitate them.

I'm not interested in securing a position as either a full stack or frontend developer as my focus is entirely placed on backend development. For the web interface I was very much against using a framework such as React, Angular or Vue mostly due to the fact that I already had my work cut out for me in developing the microservices and time spent getting acquainted with a brand new frontend framework would only significantly serve to hinder progress on the project as a whole for the benefit of being able to say that I used a new framework. I've used HTML, CSS and Javascript extensively on my own personal site and I've developed a styling and layout using Bootstrap that I've grown quite fond towards. Therefore being able to transfer over this experience would provide a massive boost to my productivity while also allowing me to provide the best experience to users as this website wouldn't be my first with this approach. Referring back to the main project objectives, the website is not my main focus in terms of using new technologies and pushing my boundaries so I saw no issues with the approach that I took.

Evaluation

Evaluation will be given based on the proposed project and personal objectives set out in the abstract based on my own views and further evaluation will be given based on some users' views on the project based on a simple questionnaire that I distributed. From an objective point of view I am confident in saying that Neo excels in terms of many criteria because it's powered by a distributed system which is exactly what I set out to achieve. With a project of its scale simply the ability to independently test and deploy individual services has been invaluable

Encountered Issues and Solutions

As with most projects with any level of planning and preparation some non-trivial issues occurred during development. In this section I will be discussing the impact of some of these issues, the solutions/workarounds that were implemented and if better planning on my behalf could've realistically prevented them.

Excessively large user documents

Initially all data received for a given user was saved in its document in MongoDB. While this granted the most freedom for exploring the data on the frontend it was quickly causing a huge bottleneck with the system in terms of the required bandwidth for saving and serving these documents. Shaving down this document from an average of 460kb to 3.4kb (a reduction by roughly a factor of 115x) took a lot of careful planning. Only the minimum amount of identifying data was to be stored for each user in an effort to eradicate any duplication that was not strictly necessary. This included opting to store only the IDs of all friends instead of account details for all friends (which includes ID, username profile URL, avatar etc). The same approach was taken when storing a user's owned games. Before an object detailing a games full name, store link, default cost, ID, total playtime metrics for linux, mac and windows and other metadata fields was being stored whereas now only the unique ID for a game and the user's total playtime across all platforms is saved. A separate table was then created for the sole purpose of mapping game IDs to their titles. In both approaches the minimum amount of uniquely identifying information is saved and duplicated multiple times and with the owned games approach the non essential metadata such as its title resides in another table where no duplication exists. This massive reduction in storage requirements comes at the cost of some extra database calls when querying for information but it's ultimately the only way that this problem could've been tackled. In terms of code changes required this compromise was extremely simple only requiring some filtering and slicing of existing data before it was sent to the datastore service to be saved.

In hindsight this would have not been particularly hard to foresee but there were many more pressing issues and topics to research during my testing phase. In addition to this the final schema for user documents wasn't known when I began development. Any planning that I could've done might've been wasted given the shifting priorities during the early stages

of development. Overall this could've been a massive issue and a devastating bottleneck for the system but the compromises taken did an excellent job at providing a fairly optimal solution without any major changes required to the operation of the main crawling loop. I could have chosen to not use PostgreSQL and instead stored the crawl documents in a compressed format in MongoDB which would not exceed the 16MB document size limit but this compromise would mean that the data would be unreadable when browsing the data with a GUI application such as Compass. I often relied heavily on Compass during testing for debugging and troubleshooting and not being able to browse data using it would not be a step I'd be willing to take just for the purpose of removing my dependency on PostgreSQL.

DOCUMENTS	43	TOTAL SIZE	19.4MB	AVG. SIZE	460.8KB
-----------	----	------------	--------	-----------	---------

User document in its original form

DOCUMENTS	51.3k	TOTAL SIZE	171.5MB	AVG. SIZE	3.4KB
-----------	-------	------------	---------	-----------	-------

Final version of the user document

Accessing the Steam API as few times as possible per user crawled

All information from the system is retrieved through the Steam API which is rate limited. With a finite amount of calls permitted per day the importance of retrieving all of an individual's crawl data in the least amount of calls to the API as possible is highlighted. This isn't exactly a show stopping issue, it's more or less an issue that if not taken care of will cause lengthy crawl times that are extremely inconvenient for end users. During my early testing of the Steam API to determine my ability to extract all of the information that I believed was required I wasn't retrieving all of the information that I ended up collecting in the finished project. I wouldn't class this as feature creep but more or less growing ambitions during development when I realised I could vastly increase the range of data collected per user which would significantly increase the value of the final product. Although the current system is not perfect in terms of querying the Steam API as few times as possible its approach is optimal given the actual design requirements that I strived for. The most impacting approach taken to minimise Steam API calls came about when I realised that the endpoint for retrieving account information (username, profile URL, avatar etc.) accepted anywhere up to 100 steam IDs in one API call. Although I wasn't making 100 separate calls beforehand which would've led to on average a 100x reduction in calls I opted to account information for every friend of the target user during the main crawl loop which ultimately allowed for greater detail in the logs for the main crawling loop.

Initially my planned approach was to make the crawl loop as fast and require as few API calls as possible but my priorities changed and making more calls made my life a lot easier. Being able to get account information for 100 users at a time is quite convenient but by no means necessary in the grand scheme of things.

Accidental leaking of sensitive files

The frontend acts as a regular microservice and as a file server. This is to serve the static files such as webpages, Javascript files and images required for the web interface. All seemed fine and it was only a week after having it deployed properly did I think to use a tool such as Nikto, a web server scanner. Nikto scans a web server for common vulnerabilities and misconfigurations which can lead to the leaking of sensitive information. Unknown to me when I had enabled the frontend to act as a file server I did not limit the directories from which content could be requested from which evidently meant that the entire instance's file system was accessible through the frontend.

The most sensitive files included were the log file, /etc/passwd and .env for the frontend which had secrets such as influxDB bucket tokens. Thankfully because of the precautions I had taken with regards to security I was able to quickly rotate all sensitive keys exposed, nuke the instance, reinstall the necessary dependencies and redeploy the frontend within 30 minutes without any lasting damage after having fixed this major loophole by only allowing retrieval of files in

whitelisted directories. This was most definitely the biggest mistake that I made during development but it thankfully had no lasting consequences. In hindsight it was quite irresponsible of the tutorial I was following to not mention this glaring security issue anywhere even though the fix can be implemented in less than a handful of lines. After this incident I regularly scanned the other microservices using Nikto. I had planned to use a vulnerability scanner during my research stage but I should've ensured that this measure was taken before exposing my services to the internet.

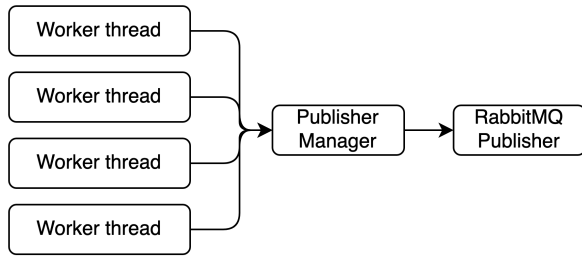
```
cathal@gomney:~/Desktop/nikto/program$ ./nikto.pl -h http://51.89.231.193/
- Nikto v2.1.6
-----
+ Target IP:          51.89.231.193
+ Target Hostname:   51.89.231.193
+ Target Port:       80
+ Start Time:        2021-12-07 15:26:23 (GMT0)
-----
+ Server: No banner retrieved
+ The anti-clickjacking X-Frame-Options header is not present.
+ The X-Content-Type-Options header is not set. This could allow the user agent to render the content of the site in a different
fashion to the MIME type.
+ No CGI Directories found (use '-C all' to force check all possible dirs)
+ OSVDB-3092: /log.txt: This might be interesting.
+ OSVDB-3092: /etc/passwd: An '/etc/passwd' file is available via the web site.
+ /.env: .env file found. The .env file may contain credentials.
+ /README.md: Readme Found
+ 8121 requests: 0 error(s) and 6 item(s) reported on remote host
+ End Time:          2021-12-07 15:35:28 (GMT0) (545 seconds)
-----
+ 1 host(s) tested
```

Nikto report scan of frontend exposing sensitive files

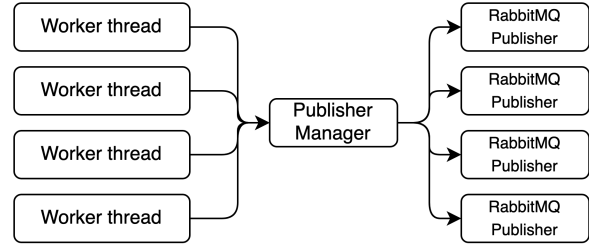
Issues arising from high throughput concurrent workloads

A number of issues occurred due to the scale of concurrent operations being executed on crawler instances ranging from the ulimit being exceeded for open file descriptors, imposed timeouts from my network's router and malformed jobs being published to RabbitMQ. All of these issues were fatal when they occurred and rate limiting measures had to be applied across the crawler service. In linux everything is a file, including open TCP sockets. A combination of resource leaks with regards to TCP connections not being closed and a high throughput of requests quickly led to the maximum allotted amount of open file descriptors being exceeded. This was solved mostly with fixing resource leaks by ensuring all connections were properly closed. Timeouts being imposed by my network's router was an awfully strange issue to occur and to diagnose but ultimately it became clear that there had to be a limit on the amount of network requests that a crawler instance could make at any given time. A mechanism was then put in place to limit the amount of concurrent requests that could be initiated at any given time. The effects of this measure on the overall throughput is negligible nearly all of the time and while it might add a few more milliseconds to some requests at peak throughput it's a necessary measure that must be taken.

The last main issue was with publishing jobs to RabbitMQ. The diagram below shows the architecture before and after a fix was implemented. Before every worker thread was consuming concurrently (which is thread safe) but they were all publishing to the same single publisher (which is not safe for unregulated access even from one thread.. Very rarely two jobs were published at nearly the same time and because of the "queue" nature of publishing instead of two jobs being sent correctly as [job 1 headers][job 1 body] [job 2 headers][job2 body] the order of the packets were mixed up and malformed jobs were being published where it appeared as [job 1 headers][job 2 headers]. This was a very hard issue to track down when it happened on a very rare occasion when the system was publishing under particularly heavy load. Thankfully I was saved by a single archived mailing list reply from 2014 discussing my exact issue and the cause behind it. The implementing solution involved maintaining a pool of publisher connections that could be used in a round robin fashion to ensure that two jobs were not being published at once in a manner that would lead to the same issues arising that the new implementation was designed to alleviate.



Crawler one publisher diagram



Crawler multiple publisher pool diagram

In hindsight I expected some issues with the amount of concurrent operations happening but I wasn't able to specifically plan for any of these specific issues that occurred. These issues are ones that particularly require past experience to be able to effectively plan for.

Degradation of performance for large graphs

Perhaps what I'd consider to be the biggest disappointment of the final product is the inability of ThreeJS to reliably and efficiently render graphs when there are usually around 500 or more users. However, I cannot directly blame this on threeJS directly as it's simply a bottleneck that ThreeJS does not advertise as being able to solve. Unfortunately when it comes to rendering graphs at the standard that ThreeJS does for the web there's nothing that's better at an all around solution. During testing I found that for a rather large graph with roughly 1,200 users ThreeJS outright failed to render it and threw an obscure error that appeared often during testing when I had the graph data formatting incorrect but this time the format was definitely correct. Even if errors like this didn't arise I believe that the performance would've been quite poor and wouldn't have provided a worthwhile experience for users.

During my testing with ThreeJS the types of examples I had running did many a comparable amount of nodes as the types that broke in the final version but the associated metadata and interaction features (like edge highlighting and zoom in on clicks) weren't present which I suspect only lessened the performance when they were fully implemented. However, graphs of this size are not common for the most part and I believe as a compromise in the future I could implement some graph shrinking techniques in an effort to remove some nodes that do not have much of an impact such as nodes with only one connected edge. Finding a good compromise between removing nodes and keeping the overall integrity of what the graph is trying to express would be quite a challenge so I most definitely would not try to rush any implementation into production any time soon without a lot of planning and evaluation.

Caching Strategies

On slower internet connections the time between first loading the graph and the successful download of the entire crawls data is often noticeable. Currently on each page load this document must be requested and redownloaded over the network which is less than ideal. Initially I thought after some very brief research that adding specific cache control headers to content that once a user has downloaded a crawls data once could easily and seamlessly be cached locally by browsers for a specified amount of time. Ideally this would allow for the user to refresh the page many times after the initial download and instantly be able to load the large document from cache and also reduce load on the datastore service induced from serving large files repeatedly. However, due to the complex nature of how different browsers handle cache internally it quickly became apparent that this would not be doable as I initially planned. The mechanisms for caching in browsers is quite complex and is hyper tuned for maximum effectiveness in ways that work against my implementations. Actions like refreshes do not respect cached content and even with the correct 'Cache-Control' headers present with appropriate expiration timers set are seemingly redundant.

With a much deeper understanding of how major browsers handle caching internally I believe I could've come to a working solution that fit my original goal but the outcome of this in-depth research would not be as beneficial as many other areas that I could invest my time into for the project. I believe that even though my initial research was extremely brief it's quite deceptive that this simple topic has so many anomalies with how it's handled by browsers. The absence of this functionality is unfortunate but its impact would have not been noticed by many users so in this situation I can accept failure.

Memory Leaks

Memory leaks occur when a resource is allocated memory for use and when it is no longer needed its memory is not properly released leading to it residing in memory for longer than is needed. If this repeatedly occurs then over time the memory usage of an application will rise as more and more objects that should be deallocated reside in memory where they serve no use to the application. In Go memory leaks are often caused by resources such as ReadStreams (HTTP request bodies) not being closed after they have been used which leads to them residing in memory for quite some time or in the worst case indefinitely. Although I usually am able to spot situations where memory leaks may occur with regular Go code I've been using lots of new dependencies such as InfluxDB which I've never used before and therefore increases the chances of a memory leak occurring due to improper management of their resources. This ultimately was what happened as in all services but notably the datastore service memory usage seemed to steadily increase as more crawls were being executed. I estimated that over the course of a day with regular use this would increase the default memory usage on the instance from 23% to roughly 60%. The simple solution would be to periodically restart the service to negate this but that's an approach I'd only take after failing to find the root cause of the leaks for days on end.

Investigation was quite difficult. Although there exists some tools that come installed with Go for profiling memory and CPU usage I found that the experience (of at least the ways that I was using them) was quite hostile towards beginners. I didn't make much progress through using tools such as pprof memory profiling but thankfully I already had an idea as to where the memory leaks might've been coming from when I began investigating. With my experience with Go I had a good understanding of things in the standard library that caused memory leaks when mismanaged so the chances of the leaks coming from new dependencies that I was working with was quite likely. Through isolating the suspected culprits in a long process of executing crawls and monitoring the memory usage I was able to find the main culprits. The memory leaks were occurring due to: certain failover mechanisms for HTTP request retries that didn't close the request bodies after reading them, multiple unnecessary initialisations of new metrics clients and other resources not being closed when interacting with PostgreSQL. The InfluxDB and PostgreSQL issues were confirmed when I took the time to properly learn from the official documentation and saw that I wasn't properly handling my resources despite my best efforts to follow them when I initially wrote the code.

I would rule these memory leaks as a product of negligence on my behalf. Although I did always consult official documentation when I was first writing code that interacted with InfluxDB and PostgreSQL, sometimes the examples given were quite different to what I was trying to produce. This turned the situation into a process where I was unsure what pieces of code were required and which were not which inevitably led to some errors in judgement on my behalf.

Metrics

The process of creating new and interesting graphs on Grafana is as follows: write the base queries for InfluxDB in Flux query language and then paste them into Grafana and style the graphs. The hardest part is writing the queries even though InfluxDB does offer an alternative which is more of a visual builder which I did opt for a handful of times. However, a lot of my more complex and interesting graph ideas that I would've loved to implement simply were out of reach when it came to attempting to write them in Flux. Like I'll say for a few things in my evaluation, if I wasn't a one

man team creating this distributed system I could've comfortably spent the time getting properly adjusted to building complex Flux queries that would've resulted in beautiful graphs. Overall the quality of the metrics is still great and definitely usable but I feel that I could've done a better job had I been given more time.

Logging

A similar story can be said with my use of Kibana. Although I find that it's an invaluable tool that I can fully utilise to my full advantage I find that it can offer so much more but I never had the chance to properly invest my time into being able to capitalise on all of its many features. This however is not by any means an outright disadvantage for the project that I didn't use all of its features but more or less just a note that even more could be possible in the future with some more research. For all of the purposes that I did use Kibana for I found that it was nothing but a pleasure to interact with and not once did I ever find myself in a situation where I found myself regretting choosing it for Neo. As a testament to the effectiveness of the ELK stack the only times (after initially setting up and initialising a new server) where I had to manually SSH into a server were due to issues involving updating the operating systems.

Despite its reliance on both Elasticsearch and Logstash I would most definitely consider using it for a personal project of mine that until now has used simple text files for logging. After having worked with the ELK stack for the duration of this project I'm much more of a strong supporter of the stack and I'm happy to have gained such vital experience in deploying and maintaining such an important piece of infrastructure instead of just interacting with it as I had before on my work placement.

Testing

Given that I am only a single developer with a limited budget each month for hosting (that would essentially double overnight if I followed this practice) maintaining both an environment for integration and production isn't viable. As a disadvantage of this decision, integration tests are constrained in certain ways. For example, a typical test might crawl a single user and then retrieve this user and assert some properties. However currently this can be done once where the user will be crawled and saved to the database then returned. On subsequent requests to crawl this user a crawl will not be initiated as it has already been crawled and the saved user will instead be returned meaning the crawling component of the test will not be executed. In an integration environment the database would be wiped on initialisation of each test suite run to ensure that no caching or existing data would have an effect.

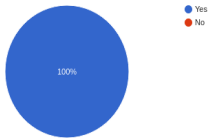
The utility functions required to facilitate things like this would also take a considerable amount of effort to implement and secure as methods to wipe databases and similar tasks would need to be executable from the services. The plethora of surrounding utilities like this that proper and robust integration tests demand is what I would consider to be a just price for the advantages that they bring.

Survey

As another means of collecting feedback in a slightly more defined manner compared to having conversations with friends who've used Neo I set up a quick and simple survey. The survey covers experiences users had based on the home page, graph page and shortest distance pages. Given the timeframe that I had to produce, distribute and collect responses for my survey the questions aren't too detailed as allowing too many free text responses can lead to a lot of manual work to filter through them. The survey was distributed among 7 of my friends and classmates who offered to give their anonymous feedback on the application.

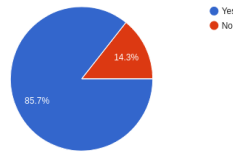
Without looking again did you notice that the landing page had live dynamic data?

7 responses



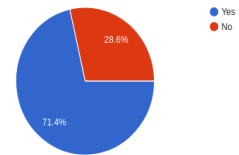
Were you able to start a crawl?

7 responses



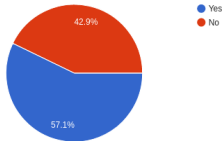
Did you read anything from the FAQ?

7 responses



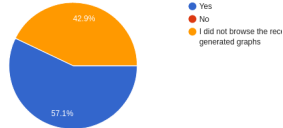
Did you browse any of the recently generated graphs?

7 responses



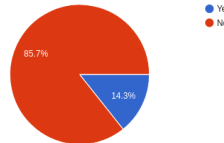
If yes, did you find they were nice interesting to browse or not?

7 responses



Did you browse any of the recently generated shortest distance graphs?

7 responses



If yes, did you find they were nice interesting to browse or not?

7 responses



With the home page I hoped that users would utilise the recently generated crawl feeds to explore other users' crawls and it seems like this was the case however the majority of users did not visit a listed shortest distance crawl. I feel like it's highlighted well enough but ultimately I believe it just suffers from being the last piece of content on the page which is inevitable as something must come last. I am thankful that virtually everyone who browsed them did find them useful in some context.

The next questions asked users about the metric and graph they felt they learned the most from and the ones they most enjoyed. Using a free text field was not the best option for this but it's clear just from reading the responses that the top scoring graphs from which users said they learned the most from were the graphs showing the network's favourite games and total hours played among the top friends. Both of these things are something that you'd never know just from playing Steam or viewing your friends' profiles so I'm not surprised that people enjoyed browsing them. I would agree with these answers as these two are also a few of my favourites. When it came to the metrics that users most enjoyed, statistics based on total hours/days played and the amount of money earned if gaming was a minimum wage job popped up quite frequently. I also have to agree that the minimum wage job metric is my favourite metric that I created purely due to how different it is to everything else. I left a separate question asking if users felt that there was any graph or metric that they felt didn't belong but this received no answers which is good to see that I stayed on topic with all of my additions to the graph page.



Project Management

Before starting this project I was never particularly a fan of writing up extensive to-do lists and drawing up proposed timetables/sprints for the work that I had planned to do. This was due in part to the fact that none of my personal projects have deadlines which got me into the habit of avoiding most forms of planning to do work in order to meet certain deadlines. However, with a project of this scale and one that would span the entire academic year I knew that in order to ensure that progress didn't stray from where it needed to be I had to break my old habits. In the first term my scheduling was roughly based on the hope that before Christmas I'd have the core of the backend developed with all of the pieces of data that I'd need for the finished project being retrieved. With this goal in mind I was able to work backwards and devise a simple list of all key milestones that had to be met in order for this deadline to be met. This list wasn't detailed at all but I did find that it did somewhat help overall with my pacing as sometimes I'd inevitably get sidetracked working on features that weren't critical or didn't progress the project towards the main goal set out for Christmas. By Christmas I had all of the core functionality needed to retrieve and process graph data working aside from the website.

For the second term I opted for a more structured approach with my project management. Within the first two weeks I had scheduled different tasks for each week-long segment which I treated as a sprint. It would be particularly disingenuous to say that I didn't find this quite difficult especially as I planned beyond 4 or 5 weeks in advance as realistically I felt I couldn't predict issues that might appear and impact all deadlines for the following sprints. Despite this I found that for the first few weeks at least that I enjoyed following this plan and knowing in my head what was coming next sprint. This was quite unusual for me as I've mentioned before most of the time on my personal project I might have a vague idea as to what I should do next but nothing is set in stone. This also helped me to not put off the tasks which I wasn't particularly fond of since I was forced to face them head on. I will admit that towards the end my sprint planning did slightly fall apart with some issues and other priorities that I had going on but those wouldn't have been possible for me to know about at the beginning of term. Comparing the two terms I liked certain aspects of having no sprint plans vs having them and although I don't see myself planning all of my time for future personal projects I might implement some new practices to some degree. I do however realise the necessity with real software projects to construct solid plans and reliably execute them but again this is just a college project and the standards and expectations are inherently different.

Hosting

I am very pleased with my choice of OVH as a main cloud provider for nearly all of my services and would happily use them again in the future. The cost of the project, functionality of their web interfaces and concise and no-nonsense billing practices are quite refreshing. Although I was not aware of any DDoS protections that OVH offered until it would've seemingly been too late I was pleasantly surprised to see that OVH automatically helped to mitigate a potential attack to one of my services. After having investigated this incident I only then realised that all infrastructure hosted on the platform is given free DDoS protection which is advertised as having the capacity to mitigate a massive 17 Terabit per second attack. I would like to note that there was no trace of any attack or even increase in traffic to the service in the logs or metrics during this timeframe. I believe that the attack was not directly targeting my specific services and was most likely against other OVH infrastructure. This is based on the fact that if this attack was directly targeting my infrastructure then my monitoring would've been able to pinpoint the exact impact and timings of an attack.

	customersupport@ovh.ie	[oc212692-ovh] End of attack on IP address 51.77.110.135	ovh.ie - https://www.ovh.ie Unit 12 The Courtyard Building, C...	06:52
	customersupport@ovh.ie	[oc212692-ovh] Detection of an attack on IP address 51.77.110.135	ovh.ie - https://www.ovh.ie Unit 12 The Courtyard Bu...	06:37

Email alerts from OVH informing me of an apparent DDoS attack on one of my services (server hosting a crawler instance)

Future Aspirations

In terms of future aspirations for the project the issues with the system's degraded ability to render large graphs springs to mind as one of the key items on my agenda should I continue development on the project. This issue in particular would require, as I've mentioned before, a considerable amount of thought and planning as to not undermine the integrity of the friend network being shown. Beyond this there are always new features and metrics that can be added to further extend the functionality of the project and given the broad range of analysis carried out on the graph page I see no reason why there aren't even more interesting ways to utilise the existing data that's being collected.

Regarding some continuous deployment strategies I'd be interested in changing my approach when it comes to how deployments are handled. Currently for a deployment each server which is targeted builds its own docker image and executes it which works perfectly fine. However, with services such as the crawler service where in the future there could be several instances it makes no sense to build the same image on multiple servers. During the average two minutes that

it takes to build an image the server's CPU rises to a near continuous 100% which can lead to degraded performance for the production instance hosted on that server as the image is built in the background to avoid downtime. Instead, managing a relatively high powered central build server that builds the latest images and then having each instance pull these new images would be much more efficient. This would also greatly benefit from the lightweight images enabled by using multi-stage docker builds to be served to each instance without requiring too much bandwidth. However, the current approach is more than adequate for the time being but if Neo was to scale further this new approach would inevitably be required to be implemented to avoid duplication of effort.

Researching the requirements for enabling TLS protection across the many services and subdomains would also serve to greatly benefit the project as a whole as more and more browsers warn users of any HTTP powered site. Although it should be noted that TLS protection is certainly not a requirement for this project as I've mentioned before because no personal or sensitive data is transferred during its use which would warrant TLS protection to avoid snooping by third parties which would compromise the privacy of any data sent to the site. The intricacies of implementing TLS protection for all services and subdomains are slightly intimidating and any steps towards doing so would require some manual testing and researching on my behalf to ensure that it would be a smooth transition.

In its current state Neo can crawl an estimated 1,120,000 new users per day, which in the context of the size of most level two crawls is a colossal amount of throughput. This estimated throughput is more than enough for the site to comfortably handle dozens of crawl requests per day. This figure of the overall throughput is only given as an estimate given the nature of the crawl loop where each individual job requires a variable amount of requests to the Steam API. As I've previously noted this was a specific design choice for various reasons but should the priorities of the system ever change I've planned an alternative approach (which only affects the ability to express some internal metadata for metrics) where each job would require the same amount of requests. This would on average increase the overall throughput of the system and the figure would be a known value instead of just an estimate.

However, it should be noted that in its current state I would happily maintain the project and continue hosting for anyone to use. While some minor things might not be perfect I am more than proud of what has been achieved and there was no major goal that I had in mind that wasn't eventually achieved in some sense.

Project Objectives

The most prominent ambition that I had from the very beginning even before I had the first idea for my project's implementation was that I wanted to create a project that was not only using a distributed system because it seemed interesting but because it would excel when it could leverage the many advantages of one. Deployments of individual services and the ability to scale horizontally with additional crawler instances is particularly effective. The overall throughput of the system is directly tied to the crawler service so I would suggest that for the foreseeable future no other service would become a bottleneck as traffic increases. Being able to run microservices on cheap cloud servers is also an inherent advantage as oftentimes the price/compute power ratio is much more expensive when it comes to single high power VPS instances as demand for them is not as strong. The only situation where I was forced to utilise a much more expensive VPS instance for hosting the ELK stack as investing in infrastructure instead of saving money where possible is particularly important.

One minor pain point with the project is its requirement for users to initiate crawls only with Steam IDs. I understand that the average user does not know their ID and instead needs to visit a service like SteamIDFinder in order to start a crawl which is less than ideal. However, I suspect that this service offered by SteamIDFinder is no easy task as I planned to implement this functionality from the very beginning and although I had a very rough testing script working I ultimately failed to come up with a robust and reliable solution that would've been up to the standard of the rest of the project.

From an end user's perspective all of my initial goals were met. Although my initial goals were quite broad and vague I'm happy to have exceeded what I initially set out to implement with many more features to enrich the service that I've created like the new user feeds, dynamic crawling status interface and plethora of interesting graphs and metrics. I consider myself decent at planning and fully acknowledge its importance but I often find that my best ideas happen in the spur of the moment which was definitely the case for a lot of features in Neo. All of the data that I wished I could've been able to retrieve from the Steam API was available to me so I wasn't limited at all in the analysis that I could undertake. One of the biggest contributing factors that led to me being able to implement so many new and interesting features was the amount of time that I was able to invest in developing new things instead of planning and having to work around issues that I didn't realise would happen during my research and testing phase. Although of course there were some days where I was only troubleshooting and fixing bugs its expected with any software project to some degree and I can happily look back and say that my diligent planning was extremely beneficial to the success of Neo.

Personal Objectives

I'm extremely proud of Neo and my experience gained as a software developer through working on it. All of my major objectives and ambitions for the project have been met and there isn't anything in particular that I regret having either done or not done. Irregardless of the project's success among real users I believe that this experience has been invaluable not just in terms of the many things I've learned but in terms of the new sense of confidence that I have in my abilities to plan and execute plans for a project of this scale which I previously thought was an undertaking simply too large for any single person like myself. This success can be pinned to a number of approaches taken to developing this project:

- What seemed like mountains of work was split into manageable chunks that not only made the project seem much more manageable but also regularly instilled a sense of confidence as I was able to regularly cross off items from my long list of components to plan and implement
- I was extremely driven by the main project idea and wanted to not only provide others with an enlightening experience when using the project but to also enjoy using the project myself.
- I thoroughly enjoyed working with new applications and overall there's nothing I love more than getting fully immersed in personal projects. The fact that this project had such a long timespan and would be a major contribution to my overall degree only served to add to my drive and enthusiasm for it.

From an objective standpoint I can say that the core principles of microservices were adhered to and no corners were cut. Initially I thought that eventually given my lack of experience I'd make some compromise for the sake of completing a feature which would directly contradict a core principle but I'm pleasantly surprised that it didn't happen. I can attribute this mostly to the amount of research that I put into learning about microservices alongside my experience during work placement. Having succeeded in adhering to these principles I can only imagine the potential issues and limitations that could have occurred if I didn't. Although I do not particularly love devops work I can appreciate how difficult it can be after having worked on Neo and I've gained a new insight into many issues that I wasn't aware of ever before such as DNS and security concerns when deploying critical pieces of infrastructure such as the ELK stack.

References

- [1] Newman, Sam. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015.
- [2] Fowler, Martin, *Microservices*, <https://martinfowler.com/articles/microservices.html>, 2014
- [3] Russel, Jeffrey, *Steam Graph Project*, <https://jrtechs.net/steam/>, 2019
- [4] Github, <https://github.com/home>, 2022

- [5] Go.Dev, *Godoc: documenting Go code*, <https://go.dev/blog/godoc>, 2011
- [6] Wikipedia, *CI/CD*, <https://en.wikipedia.org/wiki/CI/CD>, 2022
- [7] 7, Jessie. "The Test Pyramid. Thoughts on the test pyramid... | by Jessie 7." *Better Programming*, 20 May 2019, <https://betterprogramming.pub/the-test-pyramid-80d77535573>. Accessed 15 March 2022.
- [8] Hamilton, Thomas. "Positive Testing and Negative Testing with Examples." *Guru99*, 26 February 2022, <https://www.guru99.com/positive-and-negative-testing.html>. Accessed 15 March 2022.
- [9] Feathers, Michael. "A Set of Unit Testing Rules." *Artima*, 9 September 2005, <https://www.artima.com/weblogs/viewpost.jsp?thread=126923>. Accessed 15 March 2022.
- [10] System Status. *Slack System Status*, <https://status.slack.com/>. Accessed 15 March 2022.
- [11] System Status. *Discord Status*, <https://discordstatus.com/>. Accessed 15 March 2022.
- [12] Cathal O'Callaghan, *Portfolio website*, <https://cathaloc.dev/>. Accessed 15 March 2022.
- [13] Cloudflare, "Dns amplification: DDoS attack", <https://www.cloudflare.com/learning/ddos/dns-amplification-ddos-attack/>. Accessed 15 March 2022.
- [14] Gladwell, Malcolm. "Malcolm Gladwell." *Wikipedia*, https://en.wikipedia.org/wiki/Malcolm_Gladwell. Accessed 19 March 2022.
- [15] Three.js – Javascript 3D Library, <https://threejs.org/>. Accessed 19 March 2022.
- [16] "Sidecar pattern - Azure Architecture Center." *Microsoft Docs*, <https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar>. Accessed 19 March 2022.
- [17] "Slowloris (computer security)." *Wikipedia*, [https://en.wikipedia.org/wiki/Slowloris_\(computer_security\)](https://en.wikipedia.org/wiki/Slowloris_(computer_security)). Accessed 21 March 2022.
- [18] Beyer, Betsy, et al., editors. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Incorporated, 2016. Accessed 22 March 2022.
- [19] "Universally unique identifier." *Wikipedia*, https://en.wikipedia.org/wiki/Universally_unique_identifier. Accessed 23 March 2022.
- [20] Vaswani, Vikram. "Optimize Your Go Application for Production with Multi-Stage Builds and Bitnami Containers." *Bitnami Documentation*, 19 August 2019, <https://docs.bitnami.com/tutorials/optimize-docker-images-multistage-builds/>. Accessed 29 March 2022.
- [21] "Thundering herd problem." *Wikipedia*, https://en.wikipedia.org/wiki/Thundering_herd_problem. Accessed 26 March 2022.