Building a Social Media Platform Capable of Scaling to a Million Users Overnight

Harry Day

School of Computer Science

University of Manchester

Supervisor: Dr. Gareth Henshall

2025

Abstract

This is a brief summary of your dissertation. It should outline the research question, methodology, results, and conclusions.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivation

In today's increasingly digital-centred landscape, social media can transform lives and connect us in ways our ancestors could never have imagined. Social media enables friends and family to share their lives, connects customers to businesses directly, allows fans to interact with their sports teams and players, and helps employees find new jobs. Social media has allowed us to connect more and "play a positive role in strengthening the relationship between friends" (Chen et al. 2016).

One such social network that has seen consistent growth is LinkedIn. This professional social network connects professionals across all industries with their peers and colleagues. Whilst it is the gold standard, it would be impossible for the network to cater to the needs of each industry individually. This project aims to create a new social media platform enabling computer science students and software engineers to show off their personal projects

to their network, giving them a single place to direct friends, colleagues and recruiters to display their skills and engineering and creative skills. In the software industry, personal projects and past achievements are more valuable than academic grades as they show a willingness to learn and interest in software outside of work, leading to a more overall picture of a candidate than just an assessment centre.

New social media platforms often start slow but eventually hit a critical mass where growth becomes exponential, as with BeReal. In just two years, this pandemic-born platform exploded to an active user base of over 70 million (Curry 2025). One of the key focuses of this project, then, should be to create a highly scalable social media platform that can cope with the demands of millions of users overnight.

1.2 Aims and Objectives

This project aims to build a highly scalable web application, Omni, which enables users to share their personal projects online in a single space, creating an online portfolio they can share with friends, family, colleagues, and recruiters. Formally:

- Create a highly rated (in user feedback) web application that is both easy to use and pleasant to view. This application should enable users to see projects that others have posted.
- To serve this web application, create a highly scalable backend using a microservices architecture capable of scaling from 0 to millions of

requests per minute.

- Create an API that enables third parties to interact with the platform, backed by scalable microservices.
- Create a database and schema allowing sharding to enable horizontal data storage scaling.
- Follow industry standards for microservices, system design, Kubernetes, and software engineering.

1.3 Scope and Limitations

As this project primarily focuses on creating a highly scalable backend for a web platform, the majority of the focus will be on this section of the project. The front-end website shown to users will contain the minimum viable product to display the backend features but will not have much front-end 'magic' to enhance the user experience. Additionally, the platform does not contain the attributes commonly associated with social media, such as likes, follows and comments. This is an effort to combat "fakeness" on social networks: following someone to boost your social status, posting low-effort and topical content to receive vast numbers of likes. This social network intends to function closer to an online blog or portfolio, allowing software engineers to show off the cool projects they are working on without the fear of 'creators' dominating the platform.

The platform will also not have any algorithmic recommendations, although this would be interesting to explore in the future. Much of the success

of modern social media can be associated with the algorithmic suggestion of content for users to consume, as it "directly impacts user satisfaction, engagement, and retention" (Chen & Huang 2024). Despite the benefits, this machine learning algorithm could be a project within itself and falls outside the scope of building a scalable web platform.

1.4 Dissertation Structure

Chapter 2

Background

Lorem ipsum dolar

Chapter 3

Design

3.1 Architecture

Omni will use a microservices architecture. This enables a highly distributed and scalable system that can respond dynamically to changes in incoming requests. The approach should make use of common industry standards and technology as well as be portable across cloud providers. The aim is to build Omni cloud-natively so that it could scale exponentially in the case of a "viral" moment. mni will be primarily designed using the microservices architecture. This enables a highly distributed and scalable system, which is able to respond to changes in incoming requests dynamically. The approach should make use of common industry standards and technology as well as be portable across cloud providers. The aim is to build Omni cloud-natively so that it could scale exponentially in the case of a "viral" moment.

3.1.1 Kubernetes and Containerisation

Kubernetes is an orchestration tool which enables the management of pods across clusters. A cluster is one or more nodes (physical machines) connected together, possibly across different data centres or even regions, which run workloads. Figure 3.1 shows the architecture of a Kubernetes cluster[1].

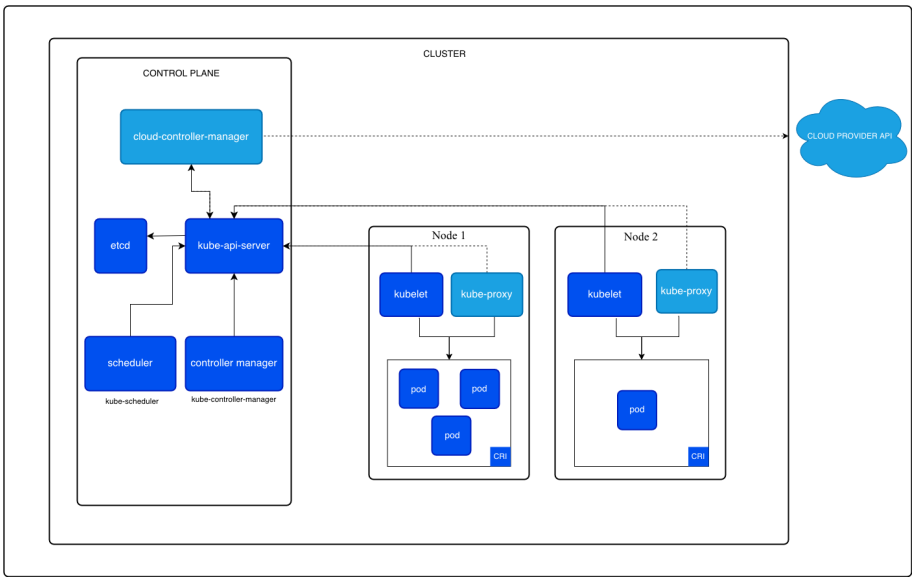

Figure 3.1: Kubernetes Architecture

Kubernetes manages the lifecycle of pods using the industry standard health check endpoints:

- `/healthz` - General health of the pod
- `/readyz` - Whether the pod is ready to receive requests
- `/livez` - Whether the pod is alive and should be running

[1] 'Kubernetes cluster components' by The Kubernetes Authors, from `https://kubernetes.io/docs/concepts/architecture/`, licensed under CC BY 4.0.

Using these endpoints, Kubernetes deployments can monitor the pods (a pod is a running unit usually containing one but sometimes more containers) they have spawned and, if any of them die, for example, if they crash, auto-heal and spawn a replacement. Deployments can also be configured to scale up the number of pods they run when certain limits are reached. The possibilities are endless, but some common examples could be CPU usage on a particular node or the number of requests received per time period.

This is what makes Kubernetes super powerful. Combining auto-healing and management of ephemeral pods with autoscaling leads to solutions that are highly adaptable to any scenario. For this reason, Omni will be designed to run natively in Kubernetes, meaning that containerised applications will need to be created to run the platform.

Containerisation is the process of modifying an application to run within a container. A container is a self-contained module that can be run without needing to worry about its dependencies or setup. For example, if an application has a dependency on a particular version of Linux with a specific library installed, containerisation will mean that the user running the container can execute the application on any supported OS without needing to worry about the dependencies.

The other added benefit of containers is that the execution environment only needs to contain the compiled binary. Rather than the host machine needing to compile the binary or a CI/CD pipeline creating many binaries for various operating systems, we can use one container, which contains only the compiled binary for the container's OS.

So, for Omni, the application(s) should be containerised and ready for

deployment to a Kubernetes cluster in any of the major cloud providers (as well as bare-metal solutions). Kubernetes is a perfect fit for the microservices architecture described in Section 1: It simplifies the management of individual containers and their lifecycles and provides auto-scaling, rollbacks, service discovery, security and portability. Because of this, Kubernetes is ideal for maintaining scalable and reliable microservice-backed platforms.

3.2 Database

Omni's database needs to scale horizontally across many machines to handle potentially hundreds of thousands or even millions of users. Database sharding will be needed to "build scalable, fault-tolerant, and efficient database architectures" (Shethiya 2025). In order to accomplish successful database sharding, one of the problems that needs to be solved is how to locate data. Formally:

> Given N nodes (where N is sufficiently large), how can a backend locate a specific piece of data whilst preventing a worst case $O(N)$.

Ideally, we want data retrieval to be of the order $O(1)$, where a backend knows precisely where to look to find a piece of data that we want to retrieve.

3.2.1 Snowflake IDs

Luckily, this problem has been solved for us. Engineers at Twitter (now X) faced this same problem and came up with an intuitive solution: snowflake

IDs. Snowflake IDs are a type of ID that not only uniquely describes a piece of data but also its locality: where it is stored (X 2010). This enables data to be sharded across many database instances whilst retaining a look-up time with a complexity of $O(1)$. Snowflake IDs are unique 64-bit integers with the following properties:

- The first bit is locked to 0, which ensures that all snowflake IDs are positive. Treating them as signed or unsigned does not matter.
- The following 41 bits are dedicated to be a timestamp. This is the time that the ID was created and is based on a custom epoch (rather than the UNIX epoch) to increase the range.
- Following the timestamp, each ID has a 10-bit Node ID. This represents the location where a piece of data has been stored.
- Finally, we have a 12-bit Sequence ID. This prevents collisions between IDs created for the same Node ID and Timestamp.

3.1 shows the final structure of the 64-bit Snowflake ID.

1 bit unused	41 bit timestamp	10 bit node id	12 bit sequence id

Table 3.1: Snowflake ID Structure

Using these Snowflake IDs, it becomes obvious how to store data across many nodes while retaining an efficient retrieval mechanism. A backend only needs to inspect the bits representing the snowflake's node to determine where to route its request. Using 10 bits for this value also allows us to scale up to 1024 nodes, which should be plenty for even the largest social media networks.

When using Snowflake IDs, the key concern is ensuring that only a single object in a given backend can create the IDs and that each backend is assigned a unique Node ID (even if multiple Node IDs are stored in the same database shard). This prevents ID collisions during generation, rather than using a 128-bit ID like the one specified in the RFC 4122 UUID specification (Leach et al. 2005).

3.2.2 Database Implementation

A classic SQL database will suit our needs since the data we will store in a social media network is structured, where the relationships between users, posts, and comments are known. The added flexibility of a NoSQL database, whilst attractive for sharding, is not warranted for such a platform, and a more strict and structured scheme should bring benefits in the long run compared to developer experience and stability.

Given this directive, some natural choices are SQLite and MySQL/MariaDB. They are both robust and straightforward; however, MySQL/MariaDB is better suited for this particular application with a single (sharded) database. In lightweight/mobile applications, SQLite is the preferred option.

As mentioned above in Section 3.2.1, the database will be sharded so that the load of multiple backends accessing data from the database is balanced across multiple servers. This allows databases to be located in separate regions and balances the load across multiple servers. Another option to balance the load on the database is to use read replicas, where there is one primary node and many read replicas, which are a copy of the primary node. This setup is

perfect for read-heavy applications, as read requests are distributed across the replicas. It also brings the benefit of higher availability, as all servers have the same database, so if the primary node fails, other nodes can be promoted to receive write requests. Replication does bring a write bottleneck where only one server can process writes to maintain ACID compliance.

Overall, sharding is the better option for the Omni platform due to the write bottleneck and the potentially massive amount of data that will be stored in the database. A future improvement would be to use a hybrid approach, where some things, like the user profile data, which is not expected to receive many writes, are stored in a replicated database, whereas post and comment data is stored in a sharded database to prevent a bottleneck when writing.

3.3 Backend

Omni has a couple of requirements for the backend:

- The backend needs to be able to service both users interacting with the frontend as well as via an API
- The backend must be able to scale to meet the needs of the incoming requests

Considering these requirements, the backend must be designed to scale efficiently. A social media platform expects to receive a lot more read requests than write requests. Splitting the backend into multiple microservices that can scale to demands independently puts the platform in the best position

to maintain availability and reliability. One way to achieve this is to use separate microservices for read and write requests. This allows us to scale the read service to use many more instances than the write service during periods of high usage.

By adopting this approach, we can also split other sections of the application into separate microservices. For example, we could use an authentication service that is only responsible for logging users in and providing them with a JWT token for future requests. There is no limit to the granularity of the microservices used. However, there will likely be a limit of diminishing returns where further breaking down microservices does not lead to significant performance improvements compared to the effort spent creating them.

3.3.1 Omni Microservices

Based on the discussion in Section 3.3 above, Omni's backend is split into three different microservices, built using GoLang:

- OmniRead, which will handle purely read API requests with no side effects
- OmniWrite, which will handle any API request that modifies the database
- OmniAuth, which will handle authenticating a user when they log in as well as provisioning JWT tokens

Figure 3.2 shows the architecture of the Omni platform, including the microservices and database.

We do not want separate API endpoints for each service. To route requests to the correct microservice, we will use a layer 4 load balancer. Because requests are balanced based on their HTTP method and path (GET requests are for OmniRead, /login requests are for OmniAuth, and everything else is for OmniWrite), a layer 4 load balancer, which operates at the application layer, is preferred over a lower-level layer 7 load balancer, which operates on TCP/UDP packets.

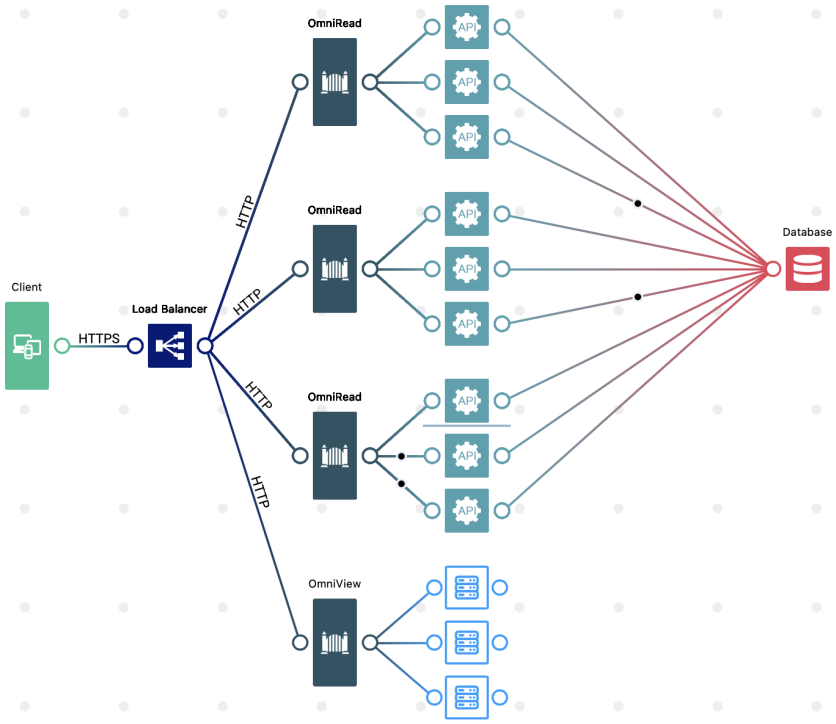

Figure 3.2: System Design of Omni Platform

3.4 Frontend

Similar to how multiple microservices serve the backend, the website will be served by a separate microservice that is only responsible for rendering page content. Any data the frontend service (OmniView) requires will be retrieved by calling the backend inside the Kubernetes cluster. Kubernetes allows pods inside the same cluster to communicate with each other via standard HTTP requests. As the website is not the main focus of this project, and a social media platform inherently does not have much onscreen interaction (instead, they are traditionally comprised of many pages linked together), the website will be built with a combination of HTMX (served via the GoLang microservice), TailWindCSS for styling, and AlpineJS for the limited amounts of more complex interaction that is required.

3.4.1 UI Design

3.5 Tech Stack

The Omni microservices will be built using GoLang. This is a fast language with excellent concurrency features, making it ideal for web servers and APIs. The SQLc library will automatically generate the database layer from the SQL migrations and queries to connect to the database from within Go. Letting a code generation framework create this code automatically ensures that the database and the microservices will always be in sync, and any errors caused by new migrations will be flagged as type errors during building.

As mentioned in Section 3.4, the website will use HTMX, TailwindCSS,

and AlpineJS. HTMX is a lightweight JavaScript framework that can be used for client-side interactivity as an alternative to ReactJS or NextJS. Using HTMX, one can dynamically update portions of the page by serving partial HTML snippets from the webserver. This can be used, for example, to add a new post to a list of posts dynamically.

MariaDB is the database of choice for the reasons mentioned in Section 3.2.2, and Kubernetes will be used to orchestrate the workloads being run. I have built a custom Kubernetes cluster running across 3 Raspberry Pis which will be used for testing, but there is nothing stopping the same deployments to be used in a cloud-based Kubernetes environment like GKE, AKS or EKS (from Google Cloud, Microsoft Azure and Amazon Web Services respectively).

Chapter 4

Implementation

This chapter will describe in detail the technical implementation of the Omni platform.

4.1 Configuration

Configuration is an important part of any production application. It is any data required by the application for it to run. Configuration may need to change, and it should be able to do so without requiring developers to rebuild the application. Allowing configuration to be ephemeral enables companies with separate teams that manage a platform's operations to make configuration changes without the need for intervention by a software developer.

Configuration items should be passed to the application via its environment rather than compiled into its binary. One method for specifying configuration is through a configuration file. This file (normally JSON or YAML) contains key-value pairs of data for the application to consume at start-up. Some

frameworks even enable the file to be monitored for changes so that the most up-to-date values can be used without requiring a restart.

Another method preferred in Kubernetes and containerised environments is using environment variables. These are variables set in the bash environment that the application is executed from. The application can then read the values of these variables and use them for its own configuration.

Kubernetes allows environment variables to be specified in the workload manifests for each application. This means that if the configuration changes, a new manifest can be loaded to bring new pods online with the updated config. Once the new pods are available and responding to requests, the old pods can be scaled down and removed.

```
1 func main() {
2   ctx := context.Background()
3   cfg, err := env.ParseAs[config.DatabaseConfig]()
4   if err != nil {
5     panic(err)
6   }
7
8   // Start-Up Code...
9 }
```

Listing 4.1: Example of Parsing Environment Variables

The second method will be used as Omni is designed for Kubernetes. To enable the use of environment variables for configuration, an open-source package called Caarlos0/env will automatically pull these values from the environment (shown in Listing 4.1). Defined in the applications are different configuration objects with the required configuration items included, along

with some special tags that indicate the name of the related environment variable. On start-up, a call to the package automatically creates the configuration objects based on the values from the environment. The config objects can then be passed around the application to the sections of code that require them.

4.2 Observability

Logging and observability are vital to the ongoing stability of any production system. "The distributed nature of microservices also introduces complexity, making it challenging to ensure the reliability, performance, and security of the system" (Chinamanagonda 2022). Clear and consistent logs are crucial for developers to be able to debug faulty platforms quickly.

Logging is not the only facet of observability. However, metrics and tracing are also key to building a holistic view of a platform's health. Metrics are the key numerical data points representing a system's health and behaviour over time. Metrics often include requests per second, latency and CPU usage. These can be easily aggregated, structured, and visualised in top-level dashboards to provide a glanceable overview of how a platform is performing. They are the first level of reporting an operations team will monitor.

Traces, on the other hand, record the path that requests take through a platform. They often capture the logs each system produces along the way and measure latencies across the system. Traces are ideal for identifying bottlenecks and pinpointing failures within a larger microservices architecture. Tracing is often handled by a framework called Open Telemetry , a

standardised way of collecting and exporting traces from many different applications.

Finally, logs are the lowest level of the three fundamental observability pillars. Logs are immutable, and applications emit timestamped events to process incoming requests. Developers with a fine-grained understanding of how the system functions best use logs, which are extremely useful for pinpointing the exact line where a system has failed.

4.2.1 Observability in Omni

In Omni, the Go standard library's structured logger emits logs, allowing each log to contain extra information, such as variables. The structured logger can output logs in various formats, the most common being text-based (for console logs) and JSON-based (for consumption into an observability pipeline).

```
if errors.Is(err, sql.ErrNoRows) {
  logger.InfoContext(
        ctx, "entity not found",
        slog.Any("id", id)
    )
  http.Error(
        w, "entity not found",
        http.StatusNotFound
    )
  return true
}
```

Listing 4.2: Example of Structured Logging

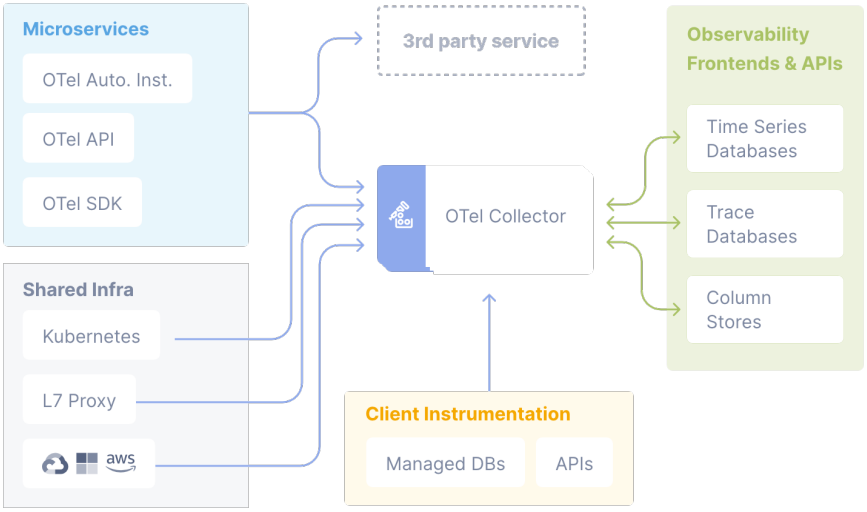

Figure 4.1: OpenTelemetry Reference Architecture.

Whilst tracing using a framework such as OpenTelemetry would be helpful in a larger production environment (Figure 4.1 shows the architecture of OpenTelemetry[1]), Omni currently only has four microservices. Tracing has been ignored for the initial product but could easily be retrofitted around the existing logging infrastructure. An OpenTel Exporter would then consume these traces from many different applications and publish them to a single location, for example, Prometheus . The Kubernetes cluster provides top-level metrics covering basic needs, while load balancers provide more in-depth metrics, such as requests per second.

[1] 'OpenTelemetry Reference Architecture' by OpenTelemetry Authors, from `https://opentelemetry.io/docs/`, licensed under CC BY 4.0.

4.3 Database

Data is at the centre of every social media platform. The first step in creating Omni should be to build the database in which its data will be stored. To ensure the database is reproducible on multiple servers (for example, when creating a new database shard), database migrations must be written and applied iteratively on top of one another until the final database structure has been formed. There are many tools and libraries for applying database migrations to a database, but since the microservices will be written in GoLang, the GoMigrate tool is a natural fit.

4.3.1 Database Tables

Figure 4.2 shows the final design of the tables. The Omni platform has two main entities that are easy to see, and a third arises during the implementation.

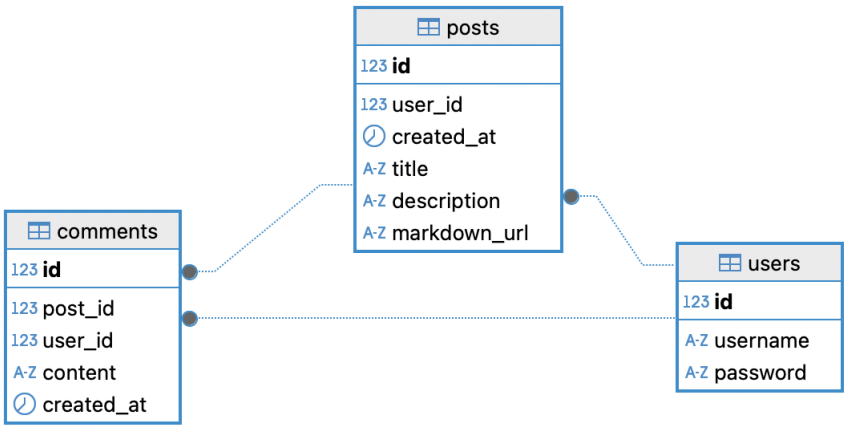

Figure 4.2: Database Entity Relationship Diagram

Information about each user needs to be stored, so a user table is naturally

required. This includes any information about the user, such as username and password hash. In the MVP of Omni, there is not much additional information stored about each user, but we could feasibly expand to store much more data for new features, such as storing a user's birthday. In the future, if a hybrid approach for database distribution is used, a separate table to hold authentication data may be better to store this on a replicated database cluster rather than a sharded one.

The second easy-to-see table holds information about each post. It must store the creator's ID, title, description, link to the markdown file, timestamp, and more. Posts will have a one-to-many relationship with users, where one user may create many posts, but each post is always associated with just one user.

The final table is not as obvious to spot but still crucial: a table to store comments in. Storing this in the posts table does not make sense in a structured SQL database. Adopting this approach could work in a NoSQL setup, but for popular posts, the document size would grow too big and slow down reading the post for everybody. By storing the comments in a separate table, we can use pagination in our SQL queries to limit the number of comments we return to a user simultaneously. This limits the load on the microservices, ensuring they are not overwhelmed by a very popular post with many comments.

4.3.2 Discussion on Sharding

As discussed in Section 3.2, the design of the database and Snowflake IDs enable easy sharding of the database, whilst maintaining fast access times by backend microservices. A single database has been used to maintain simplicity for the initial implementation and testing of the platform. The only difference this has in the code is that the logic for selecting which database node to retrieve data from is removed (as only one node contains all data).

4.4 Backend Microservices

This section discusses how the backend microservices were created and how they interact with the broader environment. Some patterns have been shared across services to enable as much code reuse as possible.

4.4.1 Database Access

All three backend services require some level of database access. OmniAuth and OmniRead require only read access, whereas OmniWrite will read and write data.

All the applications (currently) interact with the same database, although, as mentioned in Section 3.2.2, future improvements could include using separate databases for things like authentication. Because the database is shared, we can utilise the same database layer in each application, reducing the amount of code we need to write.

The SQLc Library

SQLc has been used to generate the database layer automatically. This code generation library takes as input database migrations and all the queries to be run against the database. It then compiles these inputs into type-safe code. There are multiple benefits to this approach:

- Most of the code generated is boilerplate developers must write by hand or copy from documentation. It does not solve novel problems.
- The code generated is type-safe and handles embedding objects from joins for a more traditional object-orientated programming style.
- The compiler flags breaking changes to the database via migrations when the database layer has been regenerated, but the business logic has not yet been modified.

Using this approach, SQLc has generated objects for users, posts, and comments, as well as other objects used when retrieving smaller data sections from each table. Helpfully, the library generates an interface used within the code to access the database and an implementation of the interface for accessing the database. However, this interface enables developers to write custom implementations, for example, to be used during unit testing.

```
type Post struct {
  ID          int64     `json:"id"`
  UserID      int64     `json:"user_id"`
  CreatedAt   time.Time `json:"created_at"`
  Title       string    `json:"title"`
  MarkdownUrl string    `json:"markdown_url"`
```

```
  Description string    `json:"description"`
}
```

Listing 4.3: The Generated GoLang Post Struct

All the backend services share this database layer as there is no need to write the queries separately for each one, as operations like retrieving a user profile happen in all the services.

Managing Database URLs

In order for the services to make calls to the database, they need to know where to send requests. This is defined by a URL that is passed to each application through the environment, allowing for on-the-fly changes to the configuration without having to rebuild each application. The start-up function parses this URL and then creates a database connection using Go's built-in database/SQL package. Finally, a Querier object (generated by SQLc) is initialised from the database connection object. This is the object which will be passed to each service's business logic so that database requests can be made.

4.4.2 Authentication

Authentication is an important part of any platform. It ensures that users can only access data and perform tasks they are authorised to do. Rather than using an authentication library, Omni uses a simple, custom authentication implementation, where users log in using a username and password.

Sign Up

When a user signs up to Omni, they enter a unique username and a password of their choice. This information is sent to the backend, where the password is hashed using the bcrypt algorithm developed by Provos & Mazieres. The GoLang package crypto/bcrypt handles this. The password hash is then stored in the database along with the rest of the user's details.

Bcrypt

The Bcrypt algorithm stores all the information required to verify a password attempt in the hash.

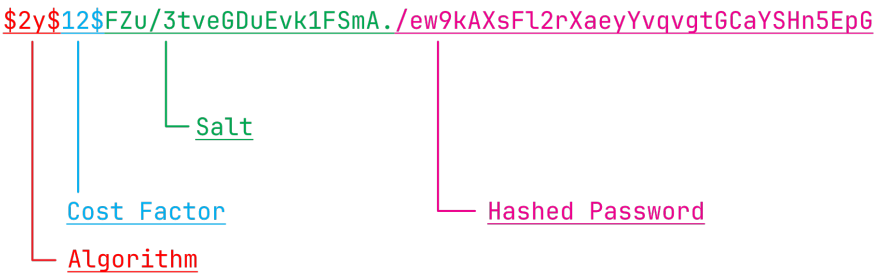

Figure 4.3: Sections of the Bcrypt Hash

The first section of the hash contains the algorithm used to create it. Bcrypt uses a version of the Blowfish cypher, denoted by the hash `$2b$`. The second section is the cost factor used in the algorithm. The default of 10 is used for Omni, but in production, this should be increased to 12 or higher. Generally, the larger the value of the cost factor, the slower it is to check a password. While counter-intuitive, this is the desired behaviour, as the slower it is to check a password, the harder it is for a brute-force algorithm to check

the most common passwords. The third section of the hash is the salt used; this is stored as plain text so that it can be used to derive the encryption key for later use. Finally, the encrypted password is stored as the last part of the hash, which is compared against a potential password.

Checking a password involves deriving the encryption key from the given algorithm, cost factor and salt, using that key to encrypt the given password and then comparing it against the valid password hash. If the hashes match, the password is valid.

JSON Web Tokens

JWTs are the industry standard for ongoing authentication. They can contain many standard or custom 'claims', which the application uses to determine whether the JWT is valid. In particular, Omni uses the `exp` and `sub` claims (standing for expires at and subject, respectively).

After a JWT body is generated, applications hash it using a secret key, which is appended to the JWT to ensure that the body is not modified by anyone other than the authorised applications. Figure 4.4 shows the three sections of a JWT.

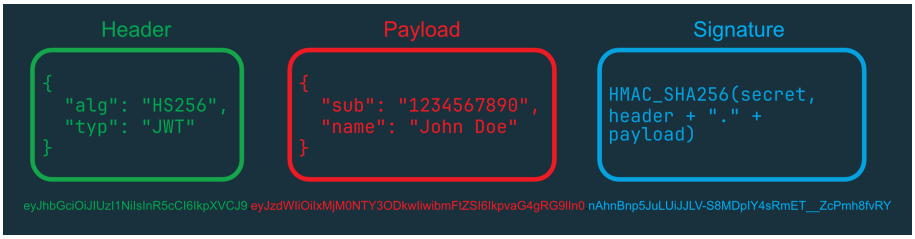

Figure 4.4: Sections of a JSON Web Token

In subsequent requests to restricted endpoints, clients attach their JWT

in the Authorization header of the HTTP request. The relevant backend, which handles requests to that particular endpoint, can then verify the claims in the JWT. Based on the subject claim of the JWT, the backend can decide if the client making the request is authorised to do so for the particular action they are performing. For example, a user can delete a post they have created but cannot delete posts from other users.

4.4.3 Middleware

Middleware is software that sits between the client request and the final logic that handles that request. It can achieve various effects, from modifying the request to rejecting it altogether (due to a lack of authentication, for example).

Middleware is crucial in modern web services, even more so in a microservices architecture. Omni utilises middleware for safety, stability, and functionality.

The most crucial and ubiquitous of Omni's middleware is the logging middleware. This middleware records each incoming request as a log, and after the request is processed, it logs the amount of time taken to serve that request. These logs follow a structured format, allowing external monitoring tools to use them to build a better understanding of how long requests take to process on average and which types of requests take the longest. This allows bottlenecks to be identified. Appendix A shows the implementation of this simple yet powerful middleware.

Another important middleware that Omni takes advantage of is MaxBytes-

Reader. This middleware sets the maximum number of bytes that can be read from the body of an HTTP request to be 1 MiB. The middleware prevents DDOS attacks using vast HTTP request bodies, where the application may run out of memory trying to decode all the information in the body. For all the Omni endpoints, the expected amount of data should never exceed 1 MiB, so any request with a body larger than that should be rejected with an error.

4.4.4 OmniAuth

The simplest of the backend services is OmniAuth, which authenticates an existing user to access restricted endpoints. OmniAuth serves just one endpoint: `POST /api/login`. The handler expects the request to have a JSON body with the fields `username` and `password`. The username is used to retrieve the user from the database. When the user does not exist, the endpoint returns a 404 Not Found HTTP error. The returned data from the database includes the Bcrypt hash of the user's password.

As mentioned above in Section 4.4.2, the crypto/bcrypt package compares the password sent in the request to the hashed password stored in the database. If the passwords match, the authentication request is valid. If they do not match, the request returns a 401 Unauthorised HTTP error.

Given that the passwords match, OmniAuth will create a JSON Web Token (JWT) (code shown in Listing 4.4) that can be used to authenticate the user for subsequent requests to restricted endpoints.

```
1 claims := &jwt.RegisteredClaims{
2   ExpiresAt: jwt.NewNumericDate(time.Now().Add(time.Hour *
     24)),
```

```
  Subject:   fmt.Sprintf("%d", id.Id().ToInt()),
}
token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
tokenString, err := token.SignedString(a.secretKey)
```

Listing 4.4: Example of Creating a JWT

4.4.5 OmniRead

OmniRead, as described in Section 3.3, is responsible for handling any HTTP Get requests. This allows us to scale read requests separately from the scaling of backend services that handle write requests. In the initial version of Omni, the data that users need to be able to read is:

- Posts
- Users
- Comments

However, we have more than three endpoints, as users may want to access different data sections differently. For example, a user may want to retrieve all the posts a user has created and the most recent posts by any user.

All the API endpoints served by OmniRead are unauthorised, meaning that anybody can access them without a JWT token. This simplifies the service regarding the middleware required for each endpoint and its runtime configuration in the Kubernetes environment.

For some endpoints that could return infinite amounts of data, such as retrieving comments on a post or posts by a particular user, paging is needed.

This ensures that the database and the backend service are not overwhelmed by a denial-of-service attack, where a malicious request can cause the entire service to hang or run out of memory while processing it. Paging ensures that the data returned has a maximum size of ten items. Users who wish to see more items can request the next page, which will return the following ten items.

Achieving this paging in the application layer is trivial, but it still leaves the attack vector open. In this scenario, the application layer would still require all the data from the database, even if it only returned a subset to the user. Luckily, popular SQL databases already have this paging feature built-in through the `LIMIT` and `OFFSET` keywords. `LIMIT`, as the name suggests, limits the number of rows of data returned, and `OFFSET` skips the number of rows specified. For example, with a `LIMIT` of 10 and `OFFSET` of 0, the first 10 rows will be returned. With an `OFFSET` of 10, rows 11-20 are returned.

```
1 -- name: GetPostsPaged :many
2 SELECT
3     users.username,
4     sqlc.embed(posts),
5     CEIL(COUNT(*) OVER() / 10.0) AS total_pages
6 FROM posts
7 JOIN users ON posts.user_id = users.id
8 ORDER BY posts.created_at DESC
9 LIMIT 10 OFFSET ?;
```

Listing 4.5: Example of SQLc Query with LIMIT and OFFSET

Other paging strategies also exist. For example, in the context of comments, a user may prefer to page by time, finding all the comments before or after a

timestamp. This paging application is more useful for other applications that use the API, perhaps for research purposes. In contrast, users interacting with the Omni platform via the website will likely want to read the most recent comments on a post or the highest-rated comments if a liking system is introduced. The `LIMIT` and `OFFSET` approach is more efficient for these cases when compared to the client specifying a timestamp or other parameter to find comments before or after, with different sorts applied to the query.

OmniRead returns data in the response body in a JSON format. JSON is the standard format for transmitting data in APIs on the web. Whilst other more efficient formats exist, such as ProtoBuf , JSON is ubiquitous and well-supported by most languages' standard libraries. GoLang has powerful, built-in tooling to encode and decode to and from the JSON format into standard GoLang structs. As part of a set of helper functions for all the backend services, Omni has a helper function that takes an object and an HTTP Response Writer object and writes the JSON encoding of the object to the response along with a 200 Success HTTP code. This helper function is used across all the backend services to transmit data to the request sender.

```
1 func MarshallToResponse(
2     ctx context.Context, logger *slog.Logger,
3     w http.ResponseWriter, v interface{}
4 ){
5   b, err := json.Marshal(v)
6   if err != nil {
7     w.WriteHeader(http.StatusInternalServerError)
8     return
9   }
```

```
10
11    w.Write(b)
12 }
```

Listing 4.6: Example of Writing JSON to the Response

Conforming to the HTTP Specification

Throughout the OmniRead service (and all the backend services), Omni attempts to conform as much as possible to the uniform HTTP interface defined by Fielding et al.. In practice, this means reflecting as much information as possible through HTTP Status Codes and adopting the correct HTTP verbs. Some examples include 200 Success and 201 No Content when requests are successful, with the former indicating there is some returned content in the body of the response, whilst the latter means nothing was returned. When a request is malformed or otherwise incorrect, status codes in the range 4XX are returned depending on the type of error. Finally, the service returns 500 Internal Server Error and 503 Service Unavailable errors when the backend encounters errors it cannot recover from, such as the database being down.

Consumers of the API should be able to use HTTP Status Codes as the first port of call to immediately know if a request was successful, rather than the way that some more modern JavaScript frameworks hide errors inside of 200 Success messages. The practice of hiding errors inside responses that return with a 200 code can lead to poor error handling and unexpected fatal crashes.

As discussed by Richardson & Ruby, "when the method information isn't found in the HTTP method, the interface stops being uniform." It is

important to conform as much as possible to the specifications that underpin the modern web; otherwise, the tooling and developers working on the web will have to work harder to achieve the same baseline of safety.

4.4.6 OmniWrite

OmniWrite handles requests that could modify the database, such as creating a post or adding a new user. For each class of resource, user, post, and comment, OmniWrite handles three endpoints:

- POST - Creates a new object
- PUT - Updates an existing object by its ID
- DELETE - Deletes an existing object by its ID

The implementation for each class is mostly the same, with the only difference being the data associated with each resource in the class. For example, the implementation of creating a new comment or post is almost the same, except for the data expected in the request body and the database calls made.

The general skeleton for the implementation of each handler for this service follows the following blueprint:

1. For PUT and DELETE requests, parse the ID parameter from the request's path and check if the resource exists in the database.
2. For POST and PUT requests, decode the request's body to find the new parameters.
3. Call the relevant function in the database layer.

4. Check for errors and return the response to the client.

Before modifying the database, the handler must also verify that the caller provided a valid JWT for the request. The only exception is when creating a new user (as the user will not yet have created their password or possess a JWT).

A JWT is valid for a PUT or DELETE request if the resource owner matches the user ID in the subject field and the token has not expired. The only other thing that complicates the request handlers in OmniWrite is the amount of places where errors can arise. The error handling logic is responsible for over half of the code in each handler, ensuring that each response receives the correct HTTP status code, depending on the type of error.

4.5 Load Balancer

Many backend services are required to handle the load for a platform to scale efficiently. Requests should be distributed evenly across each service for a platform to work optimally. A load balancer is required to achieve optimal performance. Omni has a custom, configurable load balancer that serves different paths to different backends.

4.5.1 Note on Load Balancing in the Final Version of Omni

The original plan was to use a custom load balancer for the Omni platform to prevent vendor lock-in to a particular cloud service's load balancer. However,

after carefully reviewing the Kubernetes documentation, particularly the relatively new Gateway API, the decision was made not to deploy the custom load balancer but rather to use the Gateway API to provision a load balancer from the cluster provider. This decision allows Omni to remain cloud-portable, allowing the Gateway API to automatically provision a cloud provider's load balancer.

Some work was needed to enable a load balancer in a custom-built, bare-metal Kubernetes cluster such as the one Omni was tested on (2x Raspberry Pis); however, MetalLB and NGINX Gateway Fabric were successfully implemented and used for testing.

4.5.2 Configuration

The Omni load balancer begins with the configuration. The configuration defines which load balancing algorithm to use (for example, the round-robin algorithm) and which paths the load balancer should expose.

The load balancer ingests this configuration file (written in YAML), on start-up and creates internal maps for each path. It also exposes the standard liveness endpoints `/livez` and `readyz`. `readyz` responds with a healthy status code when a backend exists and is ready to serve responses for each path defined in the configuration.

4.5.3 The Load-Balancing Pool

Two other endpoints are also exposed: `addz` and `removez`. The backends use these endpoints to register and unregister themselves from the load-balancing

pool. When a backend adds itself to the load-balancing pool, a reverse proxy is created for the backend's IP address and added to the pool. When a client sends a request via the load balancer, the new backend will be available for the selected algorithm to choose from.

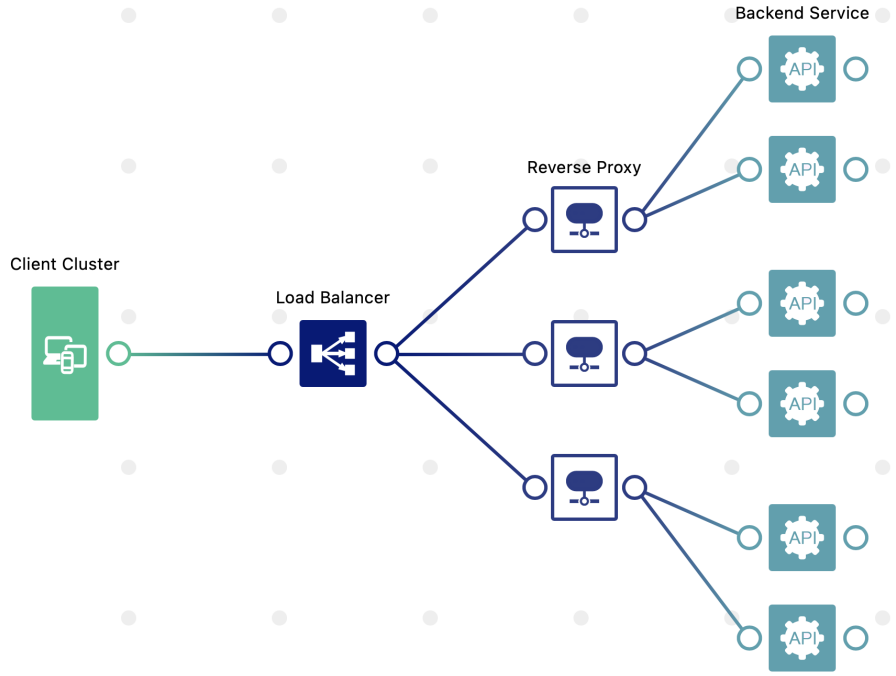

Figure 4.5: Load Balancer Design Overview

Once the chosen algorithm has picked a backend to serve the user's request, the load balancer forwards the request to the backend using its direct IP address. The request is also modified to include the `X-Real-IP` and `X-Proxy` headers to indicate to the backend nodes that the load balancer has forwarded this request and from whom the request originated.

The load balancer is also responsible for regularly checking that the registered backends are still alive and healthy. When a backend does not

respond before the timeout, it is removed from the healthy pool of nodes until it responds again or is removed from the load-balancing pool altogether.

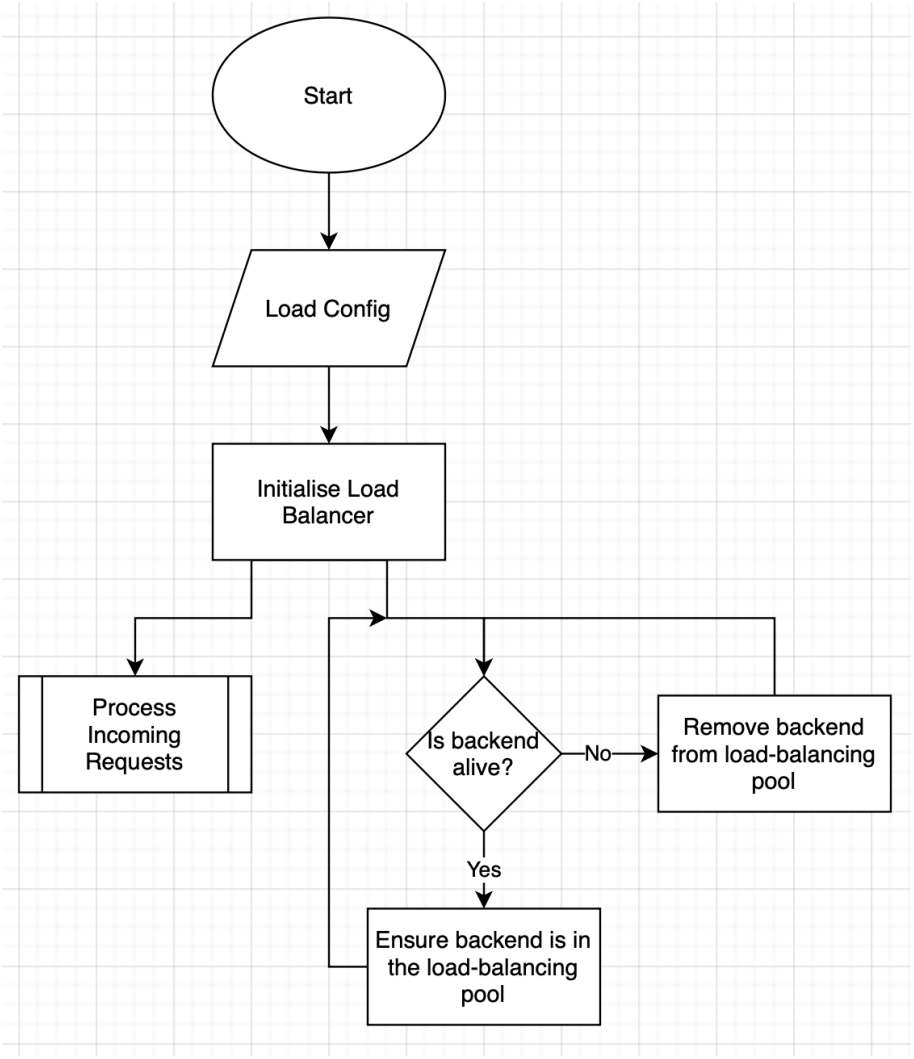

Figure 4.6: Load Balancer Flow

4.5.4 Load-Balancing Algorithms

Many different load-balancing algorithms exist, and this report does not discuss the benefits and drawbacks of each. For the first implementation of

the load balancer, a simple round-robin algorithm was implemented; however, much more complex and efficient approaches are available, such as the least connections and least response time algorithms. The study by Sharma et al. into the performance of different load-balancing algorithms is particularly informative.

Chapter 5

Testing and User Feedback

Testing is an important facet of any programming. It ensures that the code is functional and free from logical errors. Test-driven development (TDD) is the practice of writing unit tests before implementing a feature or bug fix, allowing developers to define exactly how a specific function is expected to work given a set of inputs before focusing on how the function achieves the correct outputs. Defect rates have been shown to decrease by around 50% when following TDD, compared to a traditional style of testing, where tests are written after the fact (Maximilien & Williams 2003).

5.1 Unit Testing

Unit tests are a fundamental form of testing for all applications. They are small, focused tests that verify the correctness of individual components of a system. Each test checks whether a specific function produces the expected outputs, given a set of inputs. Unit tests are reusable by design, building into

a suite of tests that are quick to run over time and ensure regressions do not occur. Regression tests are particularly valuable as they increase confidence that new changes do not break existing features unexpectedly. The field of regression testing and minimisation of regression tests is vast. However, much work has been done to automate the selection of regression tests to keep overall costs (time and energy) down whilst still providing a comprehensive suite of tests which aim to cover as much of the application as possible (Wong et al. 1997).

Omni utilises unit tests to cover most of the codebase, aiming for code coverage of over 70% but achieving close to 100% across the key, testable areas of the codebase (shown in Figure 5.1).

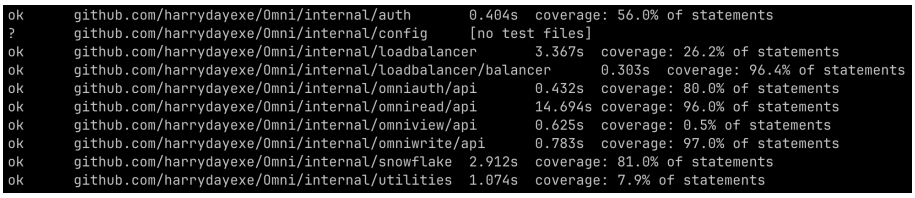

Figure 5.1: Omni Test Coverage Report

Demanding code coverage requirements (such as enforcing all code to be at least 70% covered) is less effective and even detrimental to the actual defect rates of codebases. Instead, it is more effective to encourage practitioners to focus on more in-depth code coverage metrics such as Modified Condition-Decision (MC/DC) coverage, which is much more effective at highlighting faults before they become user-facing errors (Hemmati 2015).

5.2 Integration Testing

The second important aspect of testing is the use of integration tests. Unit tests, whilst powerful and robust, are inherently limited in scope. To keep them self-contained and fast to execute, they should not interact with external services such as databases or APIs. This limitation is where integration tests shine.

As the name implies, the design of an integration test allows them to test how different sections (or units) of the system integrate. Integration can include how an application interacts with a real database or external API. Integration tests do not have the same expectations about runtime and, in some cases, can take minutes or even hours to execute more complex scenarios. However, they should still enforce the same requirements around repeatability and reusability, again to build up a suite of test cases for a product to pass before an update ships to market.

Whereas unit tests should be run consistently by the engineer throughout the development lifecycle, integration tests are usually only run after an engineer is confident in their work, often by a CI/CD tool.

5.2.1 Testcontainers

Engineers often face the challenge of ensuring that the integration tests they write are repeatable. For example, a database needs to be reset to a common starting point before each run of the tests to ensure consistency across iterations.

In 2015, an open-source project called Testcontainers was released to the

public, intending to solve just that issue. Taking the example of an integration test that interacts with a database, instead of running the tests against a permanent database, which would need to be reset and modified after each run, the Testcontainers framework allows the creation of a database inside a container. Each integration test can be run in parallel, connecting to different containers, ensuring that side effects from one test do not affect another.

Because each test can define its dependencies in a container(s), the setup for each dependency is explicitly set in code and portable for any developer to run. This allows integration tests to be run locally on an engineer's computer or in a larger CI/CD pipeline.

Adopting Testcontainers for integration tests requires just a few lines of code to configure the container in each test case. It brings all the benefits that ensure tests are repeatable, consistent and not flaky.

OmniRead utilises Testcontainers for integration testing between the request handlers and the underlying database logic, verifying that the queries written are accurate and valid. A unit test cannot check the SQL queries themselves as the database layer is mocked to ensure speed when running the test suite. Figure 5.2[1] shows the containers created as part of the integration test suite, demonstrating the parallelisation possible due to each test running on a fundamentally different database.

[1]The ryuk container runs for the duration of the test suite and is responsible for killing any containers that are not terminated by the test suite itself (for example if one of the tests fails early)

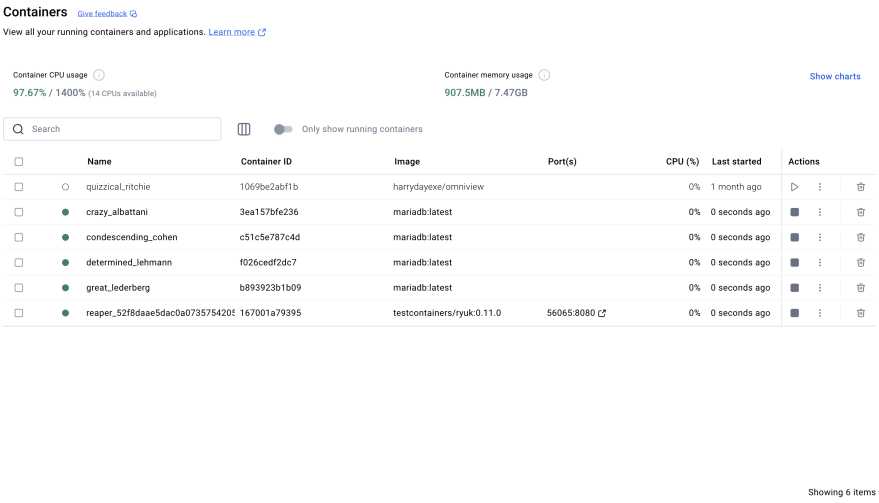

Figure 5.2: Testcontainers Running during the Integration Tests

5.2.2 Testing of the Load Balancer

Integration tests were needed to test the load balancers' ability to add and remove backends based on their health checks. However Testcontainers also uses these health checks to ensure a container is ready before allowing a test to run. In order to combat this, a custom health check container was built which exposes its actual health check on the non-standard endpoint `testcontainersz`. Using the non-standard endpoint allows the standard endpoints `healthz`, `livez`, and `readyz` to be configured for testing purposes, precisely what is needed for the load balancer integration tests.

The code for the health check tester application can be found at GitHub and the image is available on DockerHub for anyone to use.

Chapter 6

Reflection and Evaluation

Lorem ipsum dolar

Bibliography

Authors, P. (2025), 'Prometheus', `https://prometheus.io`. Accessed: 2025-04-22.

Caarlos0 (2025), 'Env', `https://github.com/caarlos0/env`. Accessed: 2025-04-22.

Chen, L., Zhu, J., Tang, Y., Fung, G., Wong, W. & Li, Z. (2016), The strength of social networks - connecting people and enhancing relationship, *in* '2016 IEEE 20th International Conference on Computer Supported Cooperative Work in Design (CSCWD)', pp. 470–475.

Chen, Y. & Huang, J. (2024), 'Effective content recommendation in new media: Leveraging algorithmic approaches', *IEEE Access* **12**, 90561–90570.

Chinamanagonda, S. (2022), 'Observability in microservices architectures-advanced observability tools for microservices environments'.

Curry, D. (2025), 'Bereal revenue and usage statistics (2025)', `https://www.businessofapps.com/data/bereal-statistics/`. Accessed: 2025-04-09.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. & Berners-Lee, T. (1999), 'Rfc2616: Hypertext transfer protocol–http/1.1'.

Foundation, C. N. C. (2025), 'Opentelemetry', `https://opentelemetry.io`. Accessed: 2025-04-22.

Google (2025*a*), 'Golang crypto/bcrypt package', `https://pkg.go.dev/golang.org/x/crypto/bcrypt`. Accessed: 2025-04-09.

Google (2025*b*), 'Golang database/sql package', `https://pkg.go.dev/database/sql`. Accessed: 2025-04-09.

Google (2025*c*), 'Protocol buffers', `https://protobuf.dev`. Accessed: 2025-04-22.

Hemmati, H. (2015), How effective are code coverage criteria?, *in* '2015 IEEE International Conference on Software Quality, Reliability and Security', IEEE, pp. 151–156.

Hugo (2025), 'Metallb', `https://metallb.io`. Accessed: 2025-04-24.

Jar, A. (2025), 'Testcontainers', `https://testcontainers.com`. Accessed: 2025-04-24.

Leach, P. J., Salz, R. & Mealling, M. H. (2005), 'A Universally Unique IDentifier (UUID) URN Namespace', RFC 4122.
URL: *https://www.rfc-editor.org/info/rfc4122*

Maximilien, E. M. & Williams, L. (2003), Assessing test-driven development at ibm, *in* '25th International Conference on Software Engineering, 2003. Proceedings.', IEEE, pp. 564–569.

NGINX (2025), 'Nginx gateway fabric', `https://docs.nginx.com/nginx-gateway-fabric/`. Accessed: 2025-04-24.

Provos, N. & Mazieres, D. (1999), A future-adaptable password scheme., *in* 'USENIX annual technical conference, FREENIX track', Vol. 1999, pp. 81–91.

Richardson, L. & Ruby, S. (2008), *RESTful web services*, " O'Reilly Media, Inc.".

Sharma, S., Singh, S. & Sharma, M. (2008), 'Performance analysis of load balancing algorithms', *World academy of science, engineering and technology* **38**(3), 269–272.

Shethiya, A. S. (2025), 'Load balancing and database sharding strategies in sql server for large-scale web applications', *Journal of Selected Topics in Academic Research* **1**(1).

SQLc (2025), 'Sqlc home page', `https://sqlc.dev`. Accessed: 2025-04-09.

Wong, W. E., Horgan, J. R., London, S. & Agrawal, H. (1997), A study of effective regression testing in practice, *in* 'PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering', IEEE, pp. 264–274.

X (2010), 'Announcing snowflake', https://blog.x.com/engineering/en_us/a/2010/announcing-snowflake. Accessed: 2025-04-12.

Appendix A

Middleware Implementations

```go
package middleware

import (
  "log/slog"
  "net/http"
  "time"
)

// middleware is a function that wraps http.Handlers
// proving functionality before and after execution
// of the h handler.
type Middleware func(h http.Handler) http.Handler

type WrappedWriter struct {
  http.ResponseWriter
  statusCode int
}

```

```
19 func (w *WrappedWriter) WriteHeader(statusCode int) {
20   w.ResponseWriter.WriteHeader(statusCode)
21   w.statusCode = statusCode
22 }
23
24 // NewLoggingMiddleware returns middleware which logs
     incoming requests
25 func NewLoggingMiddleware(logger *slog.Logger) Middleware {
26   return func(next http.Handler) http.Handler {
27     return http.HandlerFunc(
28       func(w http.ResponseWriter, r *http.Request) {
29         start := time.Now()
30
31         wrapped := &WrappedWriter{
32           ResponseWriter: w,
33           statusCode:     http.StatusOK,
34         }
35
36         logger.InfoContext(r.Context(),
37           "handling incoming request",
38           slog.String("method", r.Method),
39           slog.String("path", r.URL.Path),
40         )
41
42         next.ServeHTTP(wrapped, r)
43
44         logger.InfoContext(r.Context(),
45           "finished handling request",
46           slog.String("method", r.Method),
47           slog.String("path", r.URL.Path),
```

```
            slog.Int("status_code", wrapped.statusCode),
            slog.Duration("duration", time.Since(start)),
          )
        })
  }
}
```

Listing A.1: Logging Middleware Implementation

```
package middleware

import "net/http"

func NewMaxBytesReader() Middleware {
  return func(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *
    http.Request) {
      http.MaxBytesReader(w, r.Body, 1048576)
      next.ServeHTTP(w, r)
    })
  }
}
```

Listing A.2: Max Bytes Middleware Implementation