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

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.

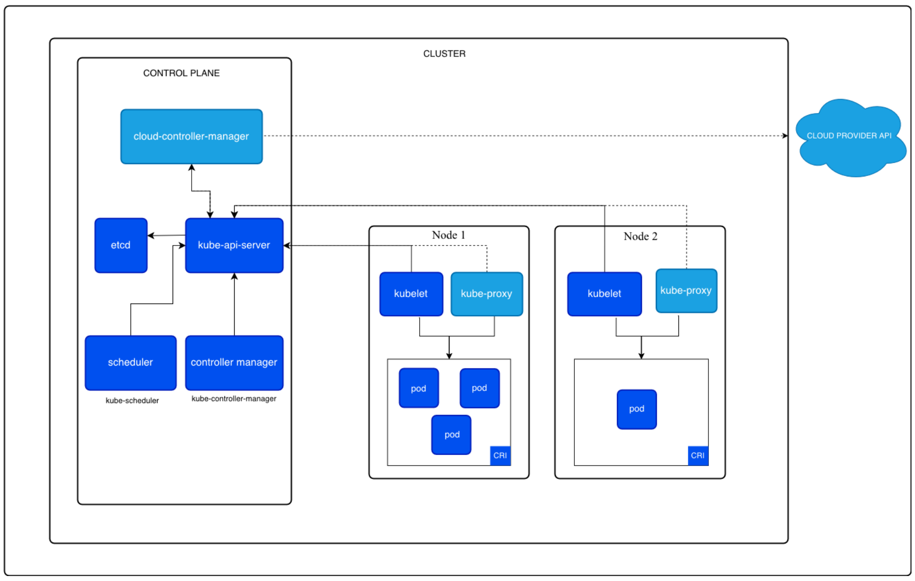

Figure 3.1: Kubernetes Architecture

Kubernetes manages the lifecycle of pods using the industry standard health check endpoints `/healthz`, `/readyz` and `/livez`. These endpoints for an application tell Kubernetes whether a pod is running (`/livez`), ready to respond to requests, possibly checking dependencies like a database (`/readyz`), and the general health of a pod through `/healthz`.

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. This means that I will need to create containerised applications 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 certain 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 exactly 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 simple and robust; 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

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.

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, garbage-collected 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, you 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 Database

Figure 4.1 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.

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

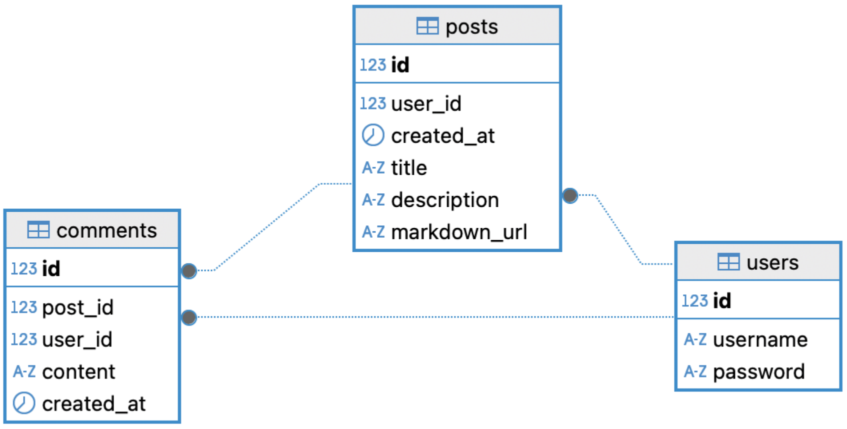

Figure 4.1: Database Entity Relationship Diagram

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.

Chapter 5

Testing and User Feedback

Lorem ipsum dolar

Chapter 6

Reflection and Evaluation

Lorem ipsum dolar

Bibliography

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.

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

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*

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).

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

Chapter 7

Appendix

Lorem ipsum dolar