Tampereen ammattikorkeakoulu

Full-Stack Development of a SaaS Application

A Case Study of a Structured Training Platform

Kha Truong

THESIS
May 2025

ICT Engineering
Software Engineering
Embedded Systems and Electronics

ABSTRACT

Tampere University of Applied Sciences
Degree Programme in ICT Engineering
Software Engineering
Embedded Systems and Electronics

Kha Truong
Full-Stack Development of a SaaS Application
A Case Study of a Strength Training Platform

Bachelor's thesis 73 pages, appendices 8 pages
May 2025

Abstract

This thesis explores the development of niche Software as a Service (SaaS) applications through the creation of Replytics, a structured workout tracking platform tailored to weightlifters and bodybuilders. Replytics incorporates a gamified leveling system aligned with user-defined goals and provides comprehensive tracking of weightlifting exercises to enhance performance.

Developing scalable and efficient SaaS applications requires a balance between performance, maintainability, and user experience in resource-constrained environments. Replytics addresses these challenges by leveraging modern lightweight technologies, including the Bun runtime and Hono framework, to build a high-performance backend with a RESTful API for seamless integration. A user-centric frontend complements this with a responsive interface and structured workout logging.

This thesis serves as a practical case study for developing niche SaaS platforms, offering insights into efficient development strategies and the advantages of adopting emerging web technologies.

Key words: Bun runtime, Full-stack development, Gamified leveling system, Hono framework, performance benchmarking, RESTful API, SaaS architecture

---

TABLE OF CONTENT

ABBREVIATIONS

API – Application Programming Interface
BASH – Bourne Again Shell (Unix shell and command language)
CDN – Content Delivery Network
CI/CD – Continuous Integration / Continuous Deployment
CPU – Central Processing Unit
CRUD – Create, Read, Update, Delete (basic database operations)
CSS – Cascading Style Sheets
DB – Database
DNS – Domain Name System
DOM – Document Object Model
GDPR – General Data Protection Regulation
HTTP – Hypertext Transfer Protocol
HTTPS – Hypertext Transfer Protocol Secure
ID – Identifier
IDE – Integrated Development Environment
IP – Internet Protocol
JS - JavaScript
JSON – JavaScript Object Notation
JWT – JSON Web Token
LAMP – Linux, Apache, MySQL, PHP (traditional full-stack web development stack)
MERN – MongoDB, Express, React, Node (modern JavaScript full-stack architecture)
RAM – Random Access Memory
SaaS – Software as a Service
T3 Stack – TypeScript, TailwindCSS, tRPC, Next.js (modern full-stack web development stack)
UI – User Interface
UX – User Experience

1 INTRODUCTION

Software as a Service (SaaS) has transformed how applications are developed and delivered, providing scalable and cost-effective solutions for businesses. SaaS enables companies to reduce costs while improving flexibility and accessibility, making it an increasingly dominant model in modern software development (Netguru, 2023). This chapter provides an overview of SaaS, its significance, and the scope of this thesis.

1.1 Background and Motivation

As cloud computing continues to shape the future of information technology, businesses are increasingly adopting cloud-based solutions to remain competitive. Driven by the growth of SaaS, IaaS, and PaaS models, the cloud market is rapidly expanding toward more scalable and efficient digital infrastructures (Yasrab, 2018). This technological shift has also transformed the fitness industry, where SaaS-based applications are enhancing user experience, performance tracking, and workout optimization.

While numerous fitness tracking applications exist, many remain either too generalized or lacking depth in key strength training metrics. Most mainstream fitness apps are designed for a broad consumer base, prioritizing features such as step tracking, calorie counting, and general workout logging rather than specialized tools for progressive overload, structured strength progression, or in-depth performance analytics (Allous, 2024). Even niche weightlifting apps often focus primarily on tracking, but lack deeper engagement features like gamification, social elements, or adaptive workout recommendations.

From my own experience and discussions with fellow lifters, many find it easier to log workouts manually using spreadsheets or notes to track long-term strength progression. While the sample size was limited (n = 15), the targeted survey offered initial insight into current app usage and feature preferences.

The responses appeared to align with initial assumptions, as 46.7 percent of respondents had never used a strength training app, and only 13.3 percent used

one regularly. Despite the availability of fitness apps, users reported relying on tools like spreadsheets or notes to track their progress, with an average agreement rating of 3.6 out of 5. When asked about desired features, respondents most frequently selected personalized progression plans (73.3%), adaptive workout recommendations (73.3%), and advanced analytics (40%). These results highlight common gaps in generalized fitness apps.

Given the limited sample size and targeted distribution, these findings are not meant to generalize but do reflect trends observed in this niche community. The survey was created and distributed using Google Forms and included both multiple-choice and Likert-scale questions. A full list of questions and answers is provided in Appendix 4.

1.2 Purpose of the Thesis

The purpose of this thesis is to examine the development of a SaaS application through the case study of Replytics, a strength training platform designed for weightlifters and bodybuilders. The research aims to explore the challenges and best practices in SaaS architecture, performance optimization, and scalable application design. By analyzing the design choices and technical implementations of Replytics, this thesis provides insights into the development of modern, high-performance SaaS solutions.

1.3 Research Questions and Objectives

Research Question

This thesis aims to explore the following primary research questions:

> **How can a niche SaaS application be designed and implemented for scalability, performance and user retention?**

To answer this, the research is guided by the following sub-questions:

- What are the key architectural principles of SaaS applications?
- How do established tech stacks (e.g., MERN) compare to emerging alternatives like Bun and Hono?

- What are the best practices for handling high-frequency workout data logging and retrieval in a SaaS fitness platform?

These questions serve as the foundation for the research and are addressed in later chapters.

Research Objectives

The objective of this thesis is to analyze and demonstrate the development of a scalable SaaS application by focusing on the following goals:

- To design and implement scalable SaaS architecture for a structured workout tracking platform.
- To compare modern web technologies (React, Bun, Hono, MongoDB) with conventional SaaS stacks.
- To identify best practices for handling real-time workout data logging and retrieval in a SaaS fitness application.

These objectives provide a structured approach to understanding the challenges and solutions in developing a modern, scalable SaaS application.

2 UNDERSTANDING SAAS: Definition and Architecture

Software as a Service (SaaS) is a cloud-based model for delivering applications, allowing users to access software over the internet instead of installing it locally. Unlike traditional software, where processing occurs on the user's device, SaaS centralizes application logic in the cloud, enabling scalability and seamless cross-device access (Cloudflare, n.d.).

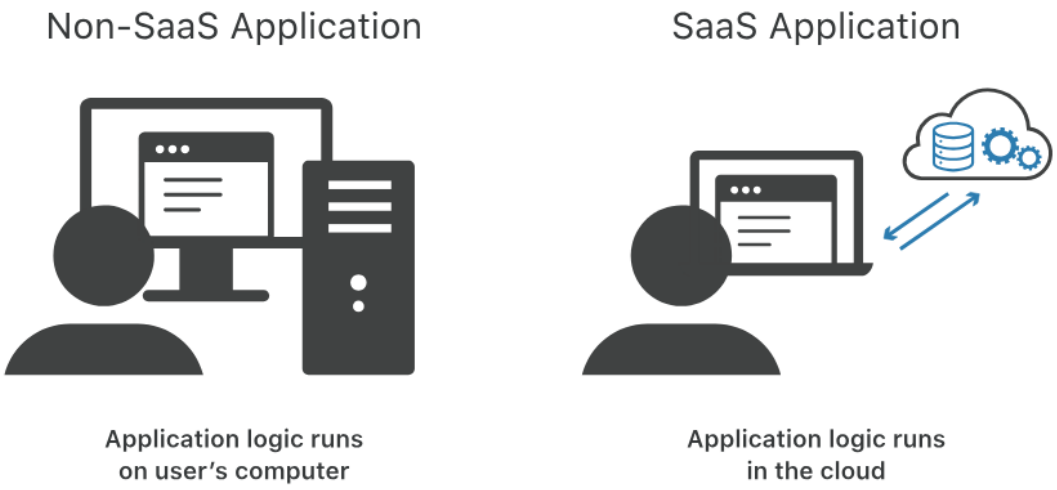

Figure 2-1. SaaS vs. Non-SaaS Application Models (Cloudflare n.d.).

As shown in Figure 2-1, traditional software applications execute the core logic directly on the user's device, whereas SaaS applications transfer this logic to cloud infrastructure. This architectural difference enables SaaS platforms to offer greater scalability, easier updates, and cross-device accessibility.

2.1 SaaS Architecture and Key Components

SaaS applications are built on scalable, multi-tenant architectures that enable efficient resource sharing while ensuring security and data isolation (Amazon Web Services, n.d.). A well-designed SaaS system must address performance, scalability, load balancing and security concerns as multiple users interact with the same infrastructure.

2.1.1 Multi-Tenancy

Multi-tenancy is a core principle of SaaS architecture, enabling multiple customers (tenants) to share the same application instance while keeping their data isolated. This approach allows SaaS providers to maximize resource efficiency, reduce infrastructure costs, and scale applications seamlessly.

Cloud providers like Microsoft Azure for example, offer built-in solutions for implementing multi-tenant SaaS applications, ensuring scalability, security, and load balancing (Microsoft Azure, 2024). Figure 2-2 illustrates a modern multi-tenant SaaS architecture deployed on Azure, highlighting key components such as traffic distribution, application orchestration, and database management.

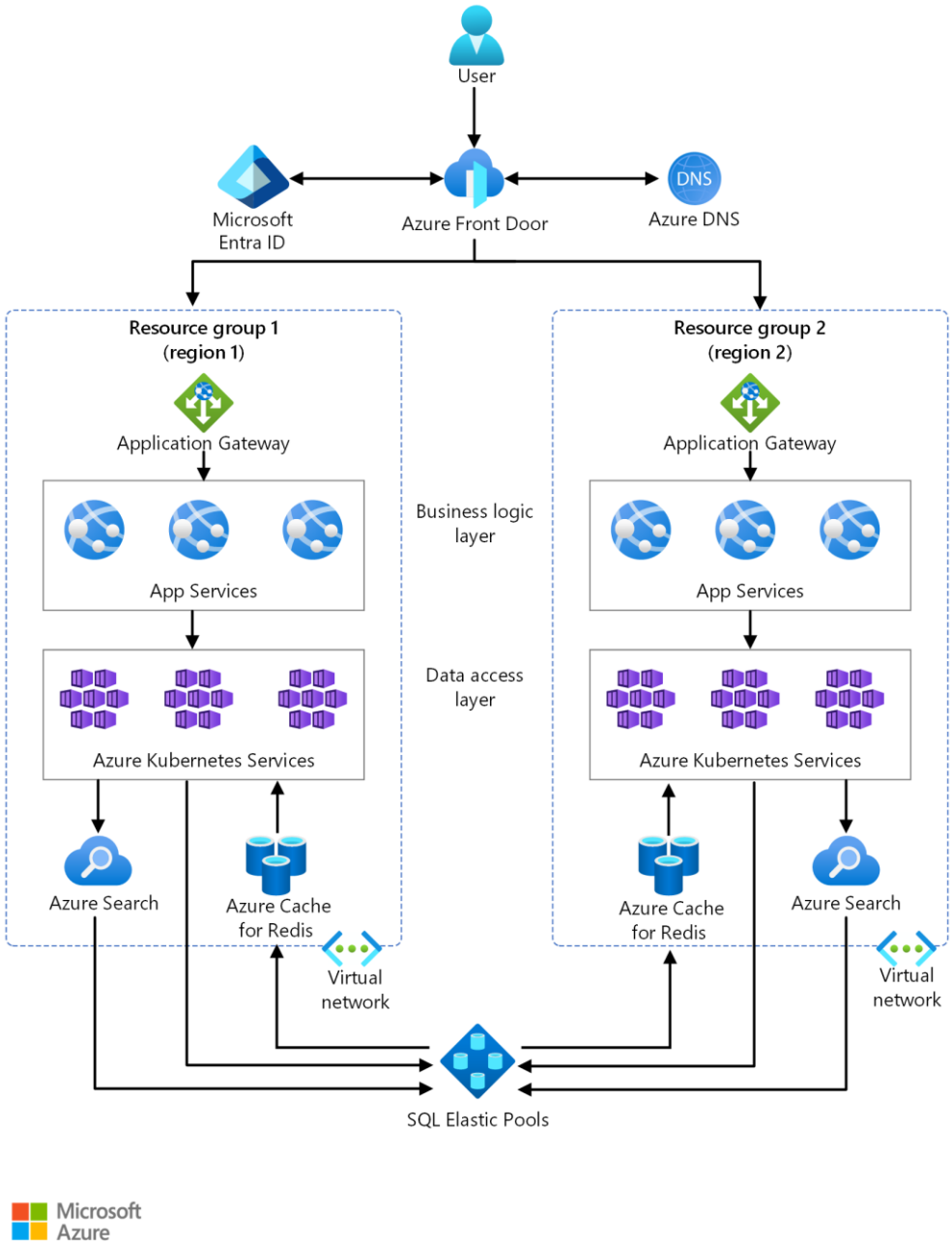

Figure 2-2. Multitenant SaaS architecture deployed on Microsoft Azure. (Microsoft Azure, 2024).

After the user request reaches Azure Front Door, traffic is routed to regional resource groups, each containing an Application Gateway and App Services that process requests dynamically.

- Application Gateway & Load Balancing: Ensures traffic is evenly distributed across multiple app services.

- Azure Kubernetes Services (AKS): Manages containerized workloads, allowing dynamic resource allocation per tenant.
- SQL Elastic Pools & Redis Cache: Provides efficient multi-tenant data storage, ensuring isolation between tenant data while optimizing cost.

This model exemplifies modern multi-tenant SaaS architectures, where scalability, security, and cost efficiency are key considerations. It contrasts with traditional single-tenant models, where each customer required a separate application instance, leading to higher costs and maintenance complexity (Liao, 2009).

2.1.2 Scalability

Multi-tenant SaaS applications must deliver seamless user experiences to customers across diverse industries. These applications share a single infrastructure among multiple tenants, making performance optimization critical for scalability, user satisfaction, and business growth (Divami, 2024).

Main components of scalability include:

- Vertical (Scaling Up): Expanding a server's hardware resources, such as increasing RAM, CPU power, or storage capacity, to improve performance.
- Horizontal (Scaling Out): Adding more server instances to distribute the workload, ensuring the system can handle increased traffic dynamically.
- Database scaling: Optimizing the database to accommodate fluctuating user volumes, which may involve sharding, replication, or caching strategies.

Figure 2-3 provides a structured comparison of these three scaling methods, highlighting their implementation, limitations, and cost considerations.

Feature	Vertical Scaling	Horizontal Scaling	Database Scaling
Resources & Implementation			
Primary Method	Adding more power to existing servers	Adding more servers	Optimizing database architecture
Key Resources	RAM, CPU, Storage	Multiple servers, Load balancers	Database servers, Caching systems
Performance & Limitations			
Scalability Limit	Hardware limitations	Virtually unlimited	Storage and query complexity
Downtime Impact	System offline during upgrades	Minimal to none	Varies by strategy
Business Considerations			
Cost Efficiency	Higher costs at scale	More cost-effective for growth	Depends on data volume
Best Use Case	Small to medium workloads	High-traffic applications	Large data operations

Figure 2-3. Comprehensive Comparison of SaaS Infrastructure Scaling Components. (PayPro Global, n.d.).

Each scalability approach has its strengths and trade-offs, as shown in Figure 2-3. Vertical scaling offers a simple upgrade path but is limited by hardware constraints. Horizontal scaling provides greater flexibility, allowing systems to handle increasing traffic dynamically, while database scaling is crucial for managing large volumes of data efficiently.

2.1.3 Load Balancing

In large-scale SaaS applications, load balancing is essential for distributing user traffic across multiple servers, preventing bottlenecks, crashes, and degraded performance. It acts as a traffic manager, ensuring that requests are efficiently allocated to available resources (PayPro Global, 2025). Without effective load

balancing, sudden spikes in traffic can lead to server failures, downtime, and poor user experiences, especially in SaaS systems with high concurrency demands.

Load balancing strategies in cloud computing are broadly categorized into static and dynamic approaches, depending on how workloads are distributed among available resources (Joshi & Kumari, 2016).

1. Static Load balancing:
    - Workloads are assigned before execution, based on predefined rules or fixed allocation strategies.
    - No real-time adjustments are made, meaning it works best in environments with predictable workloads.
    - Example: Round-robin load balancing, where each request is sequentially assigned to the next available server.

2. Dynamic Load balancing:
    - Adapts to real-time traffic conditions, redistributing workloads dynamically as system demands fluctuate.
    - Ensures better resource utilization and avoids overloading any single server.
    - Example: Least connections algorithm, which directs traffic to the server with the fewest active connections.

To efficiently distribute traffic, different load balancing algorithms are implemented, each tailored to specific system needs:

- Round-Robin: Assigns incoming requests sequentially to each server in the pool. Suitable for environments where all servers have similar processing capacity.
- Least Connections: Directs traffic to the server with the fewest active connections at the time of request, ensuring optimal resource usage.
- Weighted Load Balancing: Assigns priority weight to each server based on its processing power, directing more traffic to stronger servers.
- IP Hashing: Maps client requests to a specific server based on their IP address, ensuring session persistence.

- AI-Based Load Balancing: Uses machine learning and predictive analytics to adjust traffic distribution dynamically, optimizing response times and energy efficiency.

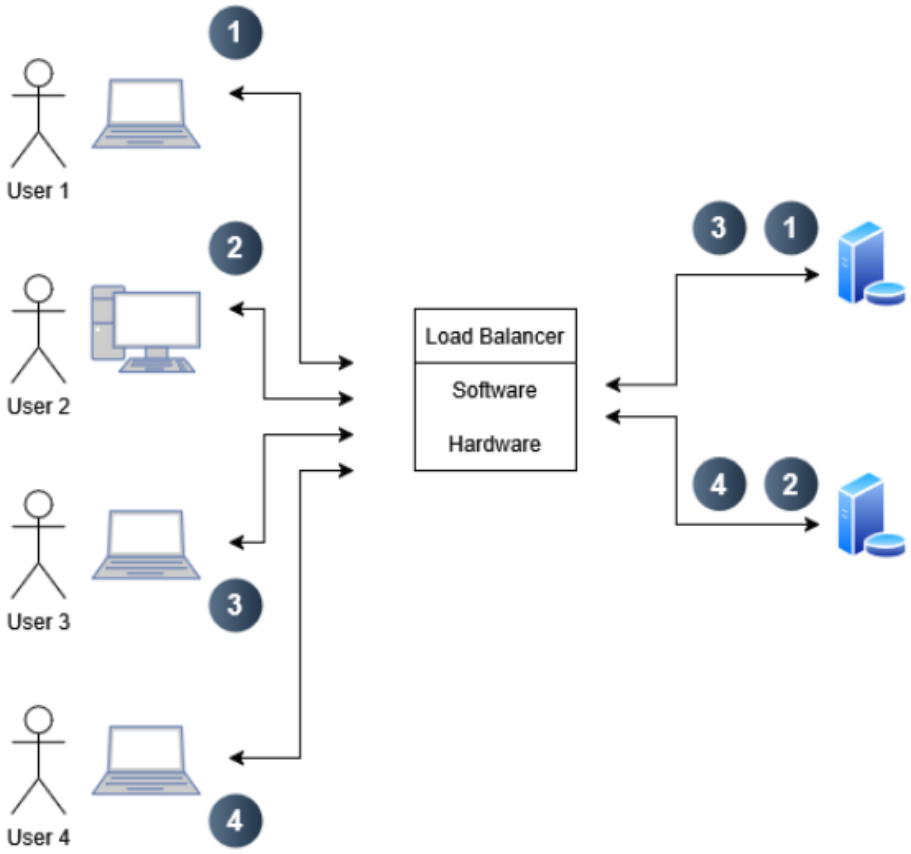

Figure 2-4. Load Balancing in a Multi-Tenant SaaS Applications using the Round Robin Algorithm.

This conceptual diagram illustrates how user requests are distributed through a load balancer, which processes traffic using either software or hardware mechanisms. The requests are then allocated across multiple servers to ensure efficiency and prevent system overload.

2.1.4 Security

Security and data privacy are critical in SaaS applications, regardless of deployment on public, private, or hybrid cloud infrastructure. While many SaaS solutions

operate on third-party managed cloud environments, some enterprises choose self-hosted or private cloud deployments to retain greater control over their data.

Studies indicate that privacy and compliance concerns remain key barriers to full SaaS adoption. SaaS providers must ensure robust security measures, such as encryption, access control, and compliance with industry standards like ISO 27001 (Tiwari & Joshi, 2014) and regulatory frameworks like GDPR (European Commission, n.d.).

SaaS platforms are common targets for Denial-of-Service (DoS) and Distributed Denial-of-Service (DDoS) attacks, where malicious actors flood systems with excessive requests to disrupt availability. These attacks can disrupt business operations, degrade performance, and cause financial losses. To defend against DoS attacks, SaaS providers deploy traffic filtering, rate limiting, and Web Application Firewalls (WAFs), along with cloud-based DDoS mitigation services to detect and block malicious traffic (Netscout, 2024).

2.1.5 Data Privacy

Unauthorized access remains a major security concern in SaaS, as user data is stored on external servers managed by the provider. Attackers may exploit weak authentication mechanisms, misconfigured access controls, or compromised credentials to gain entry. Additionally, insider threats such as employees with excessive privileges or improper offboarding can misuse or exfiltrate sensitive information. Common best practices to reduce these risks include multi-factor authentication (MFA), role-based access control (RBAC), centralized monitoring, and the principle of least privilege (PoLP) (Valence Security, n.d.).

Transparency in how data is collected, processed, and shared is essential to maintaining trust between SaaS providers and users. As SaaS adoption grows, many organizations still overlook the shared responsibility of data protection, leading to blind spots in compliance and transparency (HYCU, 2024). Privacy regulations such as the General Data Protection Regulation (GDPR) mandate that users be informed about what data is collected, how it is used, and with whom it is shared. To meet these standards, SaaS providers must implement consent

mechanisms, publish detailed privacy policies, and enable users to control their data preferences (European Commission, n.d.).

2.2 Core SaaS Architecture and Design Principles

SaaS application architecture is designed to support rapid growth and evolving user demands (CloudZero, 2023). A well-structured SaaS architecture follows several key design principles that enhance performance, security, and modularity. These principles define how SaaS applications are built, deployed, and managed while ensuring long-term efficiency.

Some of the core principles of SaaS architecture include:

- Feature modularity and component-based design
- Service-oriented and microservices architecture
- Statelessness and elasticity for cloud optimization
- Multi-tenancy and dynamic resource allocation
- Security-by-design and compliance integration

Each of these principles influences how SaaS applications handle scaling, security, and cost-efficiency. The following sections explore these in more depth, starting with feature modularity and component-based design.

2.2.1 Feature Modularity and Microservices Architecture

A modular design is essential for SaaS applications as it enables scalability, reusability, and flexibility in software development. Rather than relying on monolithic architecture, modern SaaS platforms adopt component-based approaches, where individual modules or services can be developed, tested, and deployed independently.

One approach to achieving feature modularity is Feature-Oriented Software Development (FOSD), which structures applications as configurable feature sets instead of a rigid, predefined system (Pedreira, Silva-Coira, Saavedra Places, Luaces, & González Folgueira, 2019). This allows for:

- Separation of concerns, where distinct components (e.g., authentication, reporting, API handling) function independently.
- Easier customization, enabling different tenants to activate only the features they need.
- Improved maintainability, as modular services can be updated, replaced, or scaled independently without impacting the overall system.

A well-implemented component-based design improves SaaS efficiency by reducing deployment complexity and allowing for independent scaling of different features based on demand. Many modern SaaS platforms adopt a microservices architecture, where each component (e.g., user management, billing, analytics) operates as a separate service within the system.

Figure 2-5 illustrates how Uber leverages a microservices-based SaaS architecture, structured into independent services such as passenger management, driver management, billing, payments, and notifications. Each service communicates through REST APIs via an API gateway. This modular approach enables scalability, fault isolation, and efficient service management, ensuring high availability and flexibility in handling diverse user interactions.

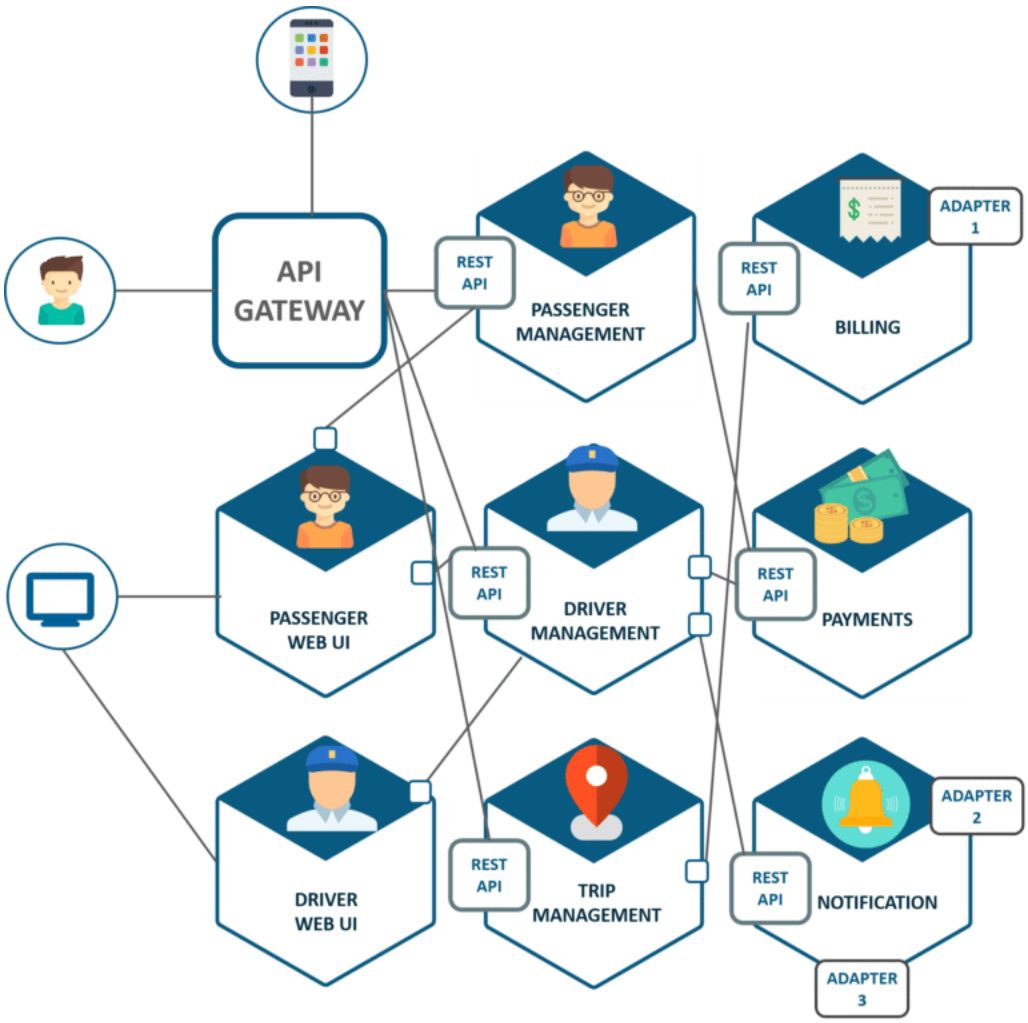

Figure 2-5: Microservices architecture of Uber. (DZone, 2018).

2.2.2 API-First Architecture and Service Communication

Seamless integration with external systems is a key architectural requirement in modern SaaS platforms. This is achieved through Application Programming Interfaces (APIs), which allow software systems to expose and consume functionality in a standardized way. APIs enable SaaS applications to interact with other cloud services, on-premises tools, or legacy systems, supporting automation, data synchronization, and modular workflows (Alumio, 2024).

According to Aleem et al. (2021), more than 70% of companies expect SaaS solutions to integrate with existing systems, whether those are other SaaS platforms or traditional enterprise software. APIs expose functionalities such as data retrieval, metadata management, and service orchestration, which are essential for

automation and extensibility. This aligns with the API-first approach, in which service communication is designed programmatically from the outset, allowing for standardized interactions and smoother third-party integration.

The API-first methodology has become foundational in modern SaaS and Backend-as-a-Service (BaaS) platforms, especially those utilizing microservices. In this approach, development begins by designing and documenting APIs before implementing backend logic. This approach ensures consistent interfaces, promotes service decoupling, and enables parallel development across frontend and backend teams. As noted by Dudjak and Martinović (2020), this design philosophy improves modularity, simplifies system integration, and accelerates time-to-market.

2.2.3 DevOps and CI/CD Integration

In modern software development, DevOps and CI/CD are pivotal concepts that enhance collaboration and streamline the delivery process. DevOps is a cultural and organizational approach that emphasizes collaboration between development and operations teams, aiming to streamline workflows, automate repetitive tasks, and create a more productive environment. It focuses on fostering a culture of collaboration, communication, and continuous improvement to deliver high-quality software faster and more reliably (LaunchDarkly, 2024).

Continuous Integration and Continuous Delivery/Deployment (CI/CD) are key practices within the DevOps framework. Continuous Integration (CI) involves automatically building, testing, and integrating code changes within a shared repository, facilitating early detection of issues and ensuring code quality. Continuous Delivery (CD) extends CI by automatically delivering code changes to production-ready environments for approval, ensuring that software can be released reliably at any time. Continuous Deployment further automates this process by automatically deploying code changes to customers without manual intervention (GitHub, 2024).

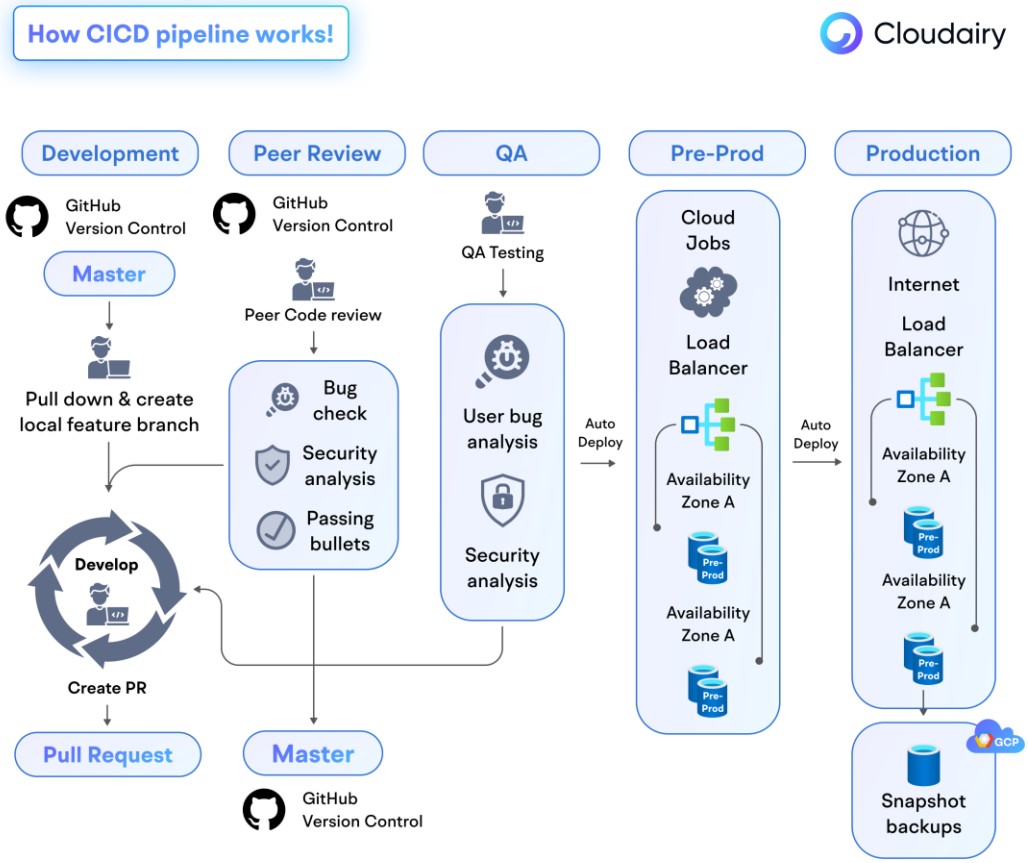

Figure 2-6. Example CI/CD pipeline architecture illustrating development, testing and deployment flow (Cloudairy, 2025).

As illustrated in Figure 2-6, CI/CD pipelines typically move code through multiple stages including development, peer review, testing and production, while enforcing quality and security checks along the way.

By integrating CI/CD practices within a DevOps culture, organizations can achieve faster development cycles, improved deployment frequency, and enhanced software reliability, ultimately delivering greater value to users.

2.3 Challenges in SaaS Development

Developing and designing a SaaS application involves a variety of architectural, technical, and operational considerations. As SaaS platforms must support multiple tenants, scale efficiently, ensure data security, and offer smooth user experience, developers frequently face a range of challenges throughout the development lifecycle, including architectural complexities to operational concerns like

maintaining performance under load, ensuring secure multi-tenancy and managing API access.

This section highlights some of the most critical challenges commonly faced in SaaS development, including multi-tenancy complexity, scalability limitations, and the risk of API abuse.

2.3.1 Multi-Tenancy Complexity

Multi-tenancy is a foundational pillar of SaaS architecture due to its scalability and cost-efficiency but implementing it at scale presents significant architectural and operational challenges.

Data isolation is a critical requirement in multi-tenant SaaS environments to ensure that each tenant's data remains secure and inaccessible to others. While all tenants share the same application instance, their data must remain isolated at both the logical and functional levels, ensuring privacy, integrity, and access control. According to Vashistha and Ahmed (2012), isolation should be considered across both functional and non-functional levels of architecture. In this context, functional isolation ensures that tenants only access their own workflows and features, while non-functional isolation addresses stability, security, and performance, ensuring that one tenant’s actions (e.g., high usage or a security breach) do not impact others.

Another concern is the “noisy neighbor” problem, where one tenant’s excessive resource consumption degrades performance for others. This requires advanced resource throttling, monitoring, and, in some cases, workload isolation using techniques like containerization or function-level scaling.

Additionally, multi-tenancy limits customization flexibility. Offering tenant-specific features or UI changes often leads to tangled codebases or complex configuration logic, increasing maintenance overhead and technical debt.

2.3.2 Scalability Limits

While cloud-based SaaS platforms are theoretically designed to scale indefinitely by adding computing power, storage, or database capacity, in practice, scalability introduces both architectural and operational challenges. Simply increasing resources does not always translate to improved performance or system reliability (Kratzke, 2018). As applications grow, complexities around data consistency, latency, cost-efficiency, and system orchestration often emerge, revealing limits that demand thoughtful architectural design rather than brute-force scaling.

One of the primary limitations of scalability lies in the increasing complexity of distributed systems. As horizontal scaling introduces more instances or microservices, managing coordination across them becomes significantly more challenging (Kratzke, 2018). Tasks such as load balancing, cache consistency, database synchronization, and maintaining session state require more sophisticated orchestration. Without careful architectural planning, these complexities can lead to race conditions, inconsistent system states, and performance bottlenecks that undermine the benefits of scaling.

From a business perspective, cost-effectiveness also acts as a ceiling. Although cloud providers offer "infinite" scaling in theory, resource usage directly correlates with costs. At some point, the marginal cost of scaling may outweigh the business value gained from supporting additional users, especially for smaller SaaS providers.

While SaaS platforms are architected to scale with user growth, the relationship between user base expansion and revenue generation is not always linear. An increasing user base leads to greater infrastructure demands, including servers, storage, database capacity, and support overhead. However, since not every new user contributes equally to revenue, many may remain on free-tier plans or contribute minimal usage activity. This means that the cost of scaling can outpace the actual financial return.

2.3.3 API Abuse

API abuse is a growing concern for businesses and organizations that rely on APIs to provide access to their services and data. API abuse can take many forms, from unauthorized access and data breaches to DDoS attacks and spamming. To prevent API abuse, it is essential to implement robust API security measures (Akamai, n.d.).

One common form of abuse involves high-frequency requests to specific endpoints by automated clients, which can exhaust system resources. Other abuse vectors include exploitation of free tiers, where users script excessive requests to bypass usage limits, and injection attacks, such as SQL or command injection, targeting poorly secured input fields.

These forms of abuse can lead to service downtime, particularly when backend systems become overloaded or unresponsive. In addition, operational costs may rise sharply due to unnecessary compute and bandwidth usage, while repeated abuse incidents can erode user trust, especially when they result in degraded performance or data exposure.

To reduce such risks, SaaS providers often implement rate limiting, which controls how frequently clients can access specific endpoints within a defined time window. Input validation and sanitization are also crucial in preventing injection-based attacks by ensuring only safe and expected data reaches backend services. Additionally, IP filtering and geo-blocking can limit access from known malicious sources, adding another layer of defense.

3 CASE STUDY: REPLYTICS - A STRENGTH TRAINING SAAS PLATFORM

This chapter presents a case study of Replytics, a structured workout tracking SaaS application developed as part of this thesis. The aim is to demonstrate the application of architectural and design principles in the context of a strength-focused fitness platform.

3.1 Overview of Replytics

Replytics is a SaaS-based tracking platform tailored specifically for strength athletes such as bodybuilders and weightlifters. Unlike generalized fitness apps that primarily focus on steps, calories, or cardio-based activities, Replytics offers structured workout logging, progression tracking, and customizable training plans to support long-term performance development.

Replytics is intended for users who follow structured strength training methodologies such as progressive overload and periodized training cycles. It addresses the limitations of traditional tools like spreadsheets and notes-taking applications, which, while flexible, lack built-in logic for performance tracking, data validation, and automated progression monitoring. Replytics functions as a superset of these manual approaches by introducing additional capabilities such as exercise templating, historical analytics, and gamification features. These include a leveling system and achievement tracking, which are designed to enhance user engagement and support long-term training adherence.

3.2 Replytics' Goals and Unique Selling Points

Replytics is designed to provide structured, data-centric tools for strength athletes seeking greater insight into their training progress. The platform supports full control over training history, allowing workouts to be logged, edited, or deleted with ease. Users can create reusable workout templates and structured plans, improving efficiency and consistency in workout logging.

A key design objective is to move beyond basic tracking by introducing performance analysis features. Interactive graphs display strength progression, volume trends, and personal records, enabling users to identify patterns and adapt their programs accordingly. The integrated calendar reinforces adherence by visually organizing training activity, while subtle UI elements such as icons and other graphical cues support engagement without distracting from core functionality.

To further differentiate Replytics, gamification features such as leveling and achievement tracking are implemented. These elements are not simply for motivation but are embedded into the progression logic, reflecting real user effort over time. The goal of this platform is to maintain the simplicity of traditional tools while adding meaningful structure, analysis, and engagement mechanisms suited for long-term use.

4 Selecting the Right Tech Stack for SaaS Development

Choosing the right technology stack is a fundamental decision in the development of any SaaS application. The selected tools, frameworks, and languages not only shape how the application is built and maintained but also how it scales, performs, and remains viable long-term. For SaaS platforms, where reliability and efficiency under high user load are critical, the tech stack must align with the product's functional requirements as well as its long-term architectural goals. This chapter explores common tech stacks used in SaaS development and outlines the reasoning behind the technologies chosen for building Replytics.

4.1 Common SaaS Tech Stacks

While the web development ecosystem continues to grow and evolve rapidly, a few technology stacks have consistently remained popular choices for building SaaS applications. These stacks offer a balanced combination of performance, community support, and scalability, making them reliable foundations for modern SaaS platforms.

4.1.1 MERN Stack

The MERN stack is one of the most widely adopted tech stacks for SaaS development. MERN stands for **M**ongoDB, **E**xpress, **R**eact, and **N**ode, after the four key technologies that make up the stack.

- MongoDB — document database
- Express(.js) — Node.js web framework
- React(.js) — a client-side JavaScript library
- Node(.js) — the most widely used JavaScript runtime environment (MongoDB, n.d.)

The MERN stack enables rapid development with a consistent language across both the backend and frontend, reducing the learning curve for developers. As a group, these components make the MERN stack a preferred choice for developers seeking an efficient full-stack JavaScript development stack. However, it may present limitations for highly relational data models or CPU-intensive operations,

where alternatives like SQL-based or compiled backend languages might perform better (Oracle, 2024).

4.1.2 LAMP Stack

A LAMP stack is a bundle of four open-source technologies commonly used to build dynamic websites and web applications. The acronym stands for:

- Linux: The operating system layer that provides the environment.
- Apache: A widely used HTTP web server that handles client requests.
- MySQL: A relational database management system responsible for storing structured application data.
- PHP: A server-side scripting language used to generate dynamic server-side content (Amazon, n.d.).

Historically, LAMP has been one of the most foundational tech stacks in web development, powering countless early web applications and content management systems like WordPress and Drupal. While newer stacks like MERN and serverless architectures have gained popularity, LAMP remains relevant, particularly in enterprise environments or legacy systems.

4.1.3 T3 Stack

The T3 Stack is a modern full-stack web development stack built around Next.js, TypeScript, and Tailwind CSS. At its core, it emphasizes type safety, modularity, and developer speed. While the stack is not rigidly defined, it often includes tools such as tRPC (for typesafe API communication), Prisma for database access, and NextAuth.js for authentication.

Unlike traditional stacks with fixed components, the T3 Stack is more of a philosophy than a strict template. As Browne (n.d.), the creator of the T3 Stack, notes: “This is NOT an all-inclusive template. We expect you to bring your own libraries that solve the needs of YOUR application.”

4.2 Replytics' Chosen Stack: React, Bun, Hono, MongoDB

The selection of Replytics' tech stack was shaped in part by the strengths and trade-offs identified in common SaaS stacks explored previously. While solutions like MERN and LAMP offer stability and broad community adoption, the goal for Replytics was to prioritize modern performance, developer experience, and minimal overhead.

Replytics' technology stack was selected to align with modern web development principles, emphasizing performance, scalability, and developer productivity. The core stack consists of React for the frontend, Bun as the JavaScript runtime, Hono as the lightweight backend framework, and MongoDB as the database. Bun and Hono were selected over Node.js and Express for their improved speed and lower resource usage, as demonstrated in performance benchmarks conducted in Section 5.4.

Together, these tools provide a fast, flexible, and scalable foundation for building a responsive SaaS platform tailored to structured strength training.

4.2.1 Frontend: React

React was selected as the frontend library for Replytics due to its proven reliability, extensive ecosystem, and flexible, component-based architecture. While React itself focuses primarily on rendering UI components, its rich ecosystem allows developers to integrate additional tools and libraries (e.g., routing, state management) to build a comprehensive frontend tailored to Replytics' requirements.

As web apps grow in complexity, efficiently updating the UI becomes increasingly challenging. React addresses this issue through the Virtual DOM (Document Object Model), a lightweight, in-memory representation of the actual DOM. React leverages the Virtual DOM to minimize direct DOM manipulation, improving performance by reducing unnecessary re-renders. According to FreeCodeCamp (2024), understanding how the Virtual DOM works is essential for optimizing performance in React applications.

4.2.2 Backend: Bun and Hono

Bun is a modern JavaScript runtime designed to serve as a drop-in replacement for Node.js. It is implemented in Zig and leverages JavaScriptCore, resulting in faster startup times, reduced memory usage, and improved execution speed compared to traditional runtimes (Bun, n.d.).

Hono is a minimalistic web framework built for speed and simplicity, particularly suited for edge runtimes. It offers built-in middleware support and an intuitive routing system while maintaining a small footprint (Hono, n.d.). Hono's light-weight, performance-oriented design aligns well with Bun's fast runtime, resulting in a cohesive backend stack optimized for speed and low resource consumption.

This combination was chosen for Replytics' backend to assess whether emerging technologies like Bun and Hono can offer efficient, maintainable alternatives to conventional stacks in SaaS development.

4.2.3 Database: MongoDB

MongoDB is a NoSQL document database designed for flexibility, scalability, and performance. It stores data in BSON format (a binary form of JSON), which allows developers to represent complex hierarchical relationships and unstructured data without requiring a rigid schema. This flexibility is particularly advantageous for SaaS applications like Replytics, where user data such as workouts, templates, and performance logs can vary significantly between individuals.

One of MongoDB's core advantages lies in its ability to adapt data structures dynamically without downtime or manual migrations. This makes it well-suited for agile, iterative development workflows commonly used in modern SaaS projects. MongoDB provides built-in support for horizontal scaling through sharding, high availability through replica sets, and flexible deployment architectures. These features make it a strong choice for handling large volumes of user-generated content and analytical workloads (MongoDB, n.d.).

In addition, MongoDB’s native developer tools and language support, particularly for JavaScript and TypeScript, make it highly accessible and intuitive for full-stack teams. Its document-based model aligns naturally with the data requirements of modern applications and helps reduce development time compared to traditional relational databases (MongoDB, n.d.).

5 EVALUATION OF BUN-HONO AND NODE-EXPRESS FOR SAAS

This chapter compares two JavaScript-based backend stacks: Bun with Hono and Node.js with Express. The objective is to evaluate their suitability for SaaS development by benchmarking key performance and resource usage metrics under consistent and controlled conditions.

While Node.js and Express have long served as the standard JavaScript backend stack, newer alternatives like Bun and Hono aim to deliver better startup speed, lower memory usage, and improved runtime performance.

5.1 Benchmarking Metrics Overview

Before diving into the implementation of the benchmark suite, it is useful to define and explain the core performance and resource metrics used in this evaluation. These metrics span both system-level and application-level perspectives and help assess the responsiveness, scalability, and efficiency of backend stacks in a SaaS context.

5.1.1 Cold Start Time

Cold start refers to the time it takes for a server to become responsive after being launched from a stopped state. This is especially relevant for serverless environments, containerized deployments, and auto-scaling infrastructure, where rapid initialization is crucial.

In real-world scenarios, fast cold start times directly impact responsiveness and operational efficiency. Serverless platforms (e.g., AWS Lambda) and container orchestrators (e.g., Kubernetes) often spin up instances on demand. A backend stack with low startup latency can handle traffic spikes more gracefully, reduce response delays for the first user, and improve developer experience during local testing or automated CI/CD pipelines.

5.1.2 Request Throughput and Latency

These metrics were collected under simulated load using Autocannon, a high-performance HTTP benchmarking tool (Autocannon, 2024):

- Requests per Second (RPS): How many requests the server can process per second.
- Latency (ms): Time taken to respond to individual requests. Lower latency reflects better responsiveness under pressure.

5.1.3 Application-Level Memory Metrics

Retrieved using the process.memoryUsage() API and exposed through a /metrics endpoint (Node.js, n.d.):

- heapUsed: The actively used portion of the JavaScript heap.
- heapTotal: The total heap allocated, which can expand dynamically.
- external: Memory used by external objects managed by the V8 engine (e.g., Buffers, TypedArrays).
- rss (Resident Set Size): Total memory allocated to the process, including heap, stack, and native code.
- heapUsedPercentage: Ratio of heap used to total heap, useful for observing memory pressure at runtime.

5.1.4 System-Level Resource Metrics

Collected using Linux system utilities:

- Memory (MB): Captured via *ps*, reflecting actual RAM usage.
- CPU (%): Processor utilization during high-load operation.
- Child Processes: Number of subprocesses spawned, if any, under load.

5.2 Test Bench Design

To ensure consistent and reproducible testing across both stacks, the following tools and methodologies were employed:

- **Autocannon**: Simulates concurrent HTTP clients and captures metrics such as request throughput, response latency, and network throughput.
- **Custom Bash Scripts**: Automate test execution, handling server lifecycle, timing cold start, and logging result.
- **System and App-Level Monitoring**: Combines native Linux tools (ps, lsof, pgrep) and custom /metrics endpoints to track CPU usage, memory consumption, and garbage collection behavior.
- **Git Branch Isolation**: Each test case (cold start, JSON response, and resource profiling) was maintained in its own Git branch to ensure versioned reproducibility and easy separation of test scenarios.

5.3 Benchmark Implementation

5.3.1 Cold Start Test - Implementation Details

The cold start test focused on measuring the time required for each server to become responsive after being launched. A Bash script (Appendix 1) was developed to automate five iterations per stack (Bun + Hono and Node.js + Express), ensuring consistent and repeatable results.

For each iteration, the script executed the following steps:

- **Process Cleanup**: Any existing process occupying the test port (3000 or 3001) was terminated using lsof and kill.
- **Timed Launch**: The server was launched via bun run index.ts or npm start, and its process ID (PID) was tracked.
- **Response Polling**: A loop using curl continuously attempted to reach the root endpoint (/) until a successful HTTP 200 response was returned.
- **Timing Calculation**: The duration from execution to first response was measured using nanosecond precision timestamps (date +%s%N).

Both servers exposed a minimal GET / route returning plain text to ensure identical behavior. This minimal implementation ensured a controlled environment focused solely on measuring startup latency, without the influence of processing or payload complexity.

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.send('Hello World!');
});

const port = 3001;
app.listen(port, () => {
  console.log(`Express server running on port ${port}`);
});

```

Figure 5-1. Minimal Express server used for cold start benchmarking.

```
import { Hono } from 'hono'

const app = new Hono()

app.get('/', (c) => c.text('Hello World!'))

export default {
  port: 3000,
  fetch: app.fetch,
}

```

Figure 5-2. Minimal Hono server used for cold start benchmarking.

Figures 5-1 and 5-2 illustrate the minimal server implementations used for benchmarking cold start times. The Express server (Figure 5-1) and the Hono server (Figure 5-2) both expose a simple GET endpoint at the root path, ensuring consistent functionality between stacks to enable fair comparison.

5.3.2 JSON Response Throughput - Implementation Details

This benchmark measured each stack's ability to handle a high volume of JSON responses under concurrent load, simulating realistic SaaS traffic patterns. Both implementations exposed a single GET / route that returned a timestamped message, as shown in Figure 5-1 and Figure 5-2.

The test was conducted using Autocannon, a high-performance HTTP benchmarking tool capable of generating thousands of requests per second with configurable concurrency and pipelining.

The implementation included the following steps:

- Each server exposed a single GET / route that returned a small JSON object containing:
    - A greeting message
    - A Timestamp
- Autocannon Configuration:
    - Connections: 100 concurrent clients
    - Duration: 30 seconds
    - Pipelining factor: 10 (to simulate multiple in-flight requests per connection)
- Target endpoints:
    - http://localhost:3000 for Bun + Hono
    - http://localhost:3001 for Node.js + Express
- Script Automation (Appendix 2) used to:
    - Start the server in the background
    - Wait for initialization
    - Run autocannon with the specified parameters
    - Cleanly terminate the server process after each test

To avoid resource contention and ensure consistency, the test was run separately for each stack. Both server implementations were functionally identical, ensuring a fair and unbiased comparison.

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.json({
    message: "Hello, World",
    timestamp: new Date().toISOString(),
  });
});

const port = 3001;
app.listen(port, () => {
  console.log(`Express server running on port ${port}`);
});

```

Figure 5-3. Minimal Express server used for JSON response benchmarking.

```
import { Hono } from 'hono'

const app = new Hono()

app.get('/', (c) => c.json({
    message: "Hello World",
    timestamp: new Date().toISOString(),
}))

export default {
    port: 3000,
    fetch: app.fetch,
}

```

Figure 5-4. Minimal Hono server used for JSON response benchmarking.

Figures 5-3 and 5-4 show the minimal server implementations used for JSON response benchmarking. Both the Express server (Figure 5-3) and the Hono server (Figure 5-4) expose a single GET endpoint that returns a lightweight JSON object containing a greeting message and a timestamp. These served as the

baseline API endpoints for measuring throughput and latency under concurrent load.

A custom bash script was used to simulate 100 concurrent clients sending pipelined HTTP requests for 30 seconds to either the Bun + Hono or Node.js + Express server. It automates the entire test lifecycle, including server startup, execution, and cleanup, ensuring consistent and repeatable benchmarking across runs.

5.3.3 Resource Usage Profiling - Implementation Details

The third benchmark focused on profiling the runtime resource usage of each stack under sustained load. While request throughput and latency offer valuable performance indicators, understanding memory behavior, CPU consumption, and process management is just as important when evaluating backend efficiency in SaaS environments.

This test was designed to capture both system-level and application-level metrics by combining native Linux utilities with internal monitoring endpoints exposed by each server.

Monitoring Implementation

Each server implementation (see Figure 5-5 and Figure 5-6) included a /metrics route that returned detailed memory statistics using the process.memoryUsage() API. The metrics collected included:

- **heapUsed**: Memory used by active JavaScript objects
- **heapTotal**: Total heap allocated, which can dynamically grow
- **external**: Memory used by external resources (e.g., Buffers, TypedArrays)
- **rss** (Resident Set Size): Total memory allocated to the process, including heap, stack, and native code
- **heapUsedPercentage**: The ratio of heapUsed to heapTotal, indicating memory utilization intensity

A bash script (Appendix 3) automated the benchmarking and monitoring process:

- Server Launch: Each stack was started using “bun run” or “npm start”
- Load Simulation: A 30-second Autocannon test simulated 100 concurrent clients with pipelining enabled
- Resource Sampling: Every second, the script queried:
    - ps for CPU and RSS

    - curl for application memory metrics
    - pgrep for child process count
- Logging: All values were stored in a CSV for later analysis
- Post-Test Aggregation: Averages were computed for each metric to summarize behavior under load

Application Endpoints

Each server exposed two routes:

- /: Returned a small JSON message and timestamp
- /metrics: Returned structured JSON containing memory usage details (see Figures X and X)

These endpoints ensured consistent load generation while allowing introspection into the runtime memory behavior of each implementation.

```
const express = require('express');
const app = express();

function getMemoryMetrics() {
    const heapUsed = process.memoryUsage().heapUsed;
    const heapTotal = process.memoryUsage().heapTotal;
    const external = process.memoryUsage().external;
    const rss = process.memoryUsage().rss;

    return {
        message: "Hello, World",
        timestamp: new Date().toISOString(),
        metrics: {
            memory: {
                heapUsed: `${Math.round(heapUsed / 1024 / 1024)} MB`,
                heapTotal: `${Math.round(heapTotal / 1024 / 1024)} MB`,
                external: `${Math.round(external / 1024 / 1024)} MB`,
                rss: `${Math.round(rss / 1024 / 1024)} MB`,
                heapUsedPercentage: `${Math.round((heapUsed / heapTotal) * 100)}%`
            }
        }
    };
}

app.get('/', (req, res) => {
    res.json({
        message: "Hello, World",
        timestamp: new Date().toISOString(),
    })
})
app.get('/metrics', (req, res) => {
    res.json(getMemoryMetrics());
});

const port = 3001;
app.listen(port, () => {
    console.log(`Express server running on port ${port}`);
});
```

Figure 5-5. Express-based /metrics implementation.

```
import { Hono } from 'hono'

const app = new Hono()

function getMemoryMetrics() {
    const memory = process.memoryUsage();
    const heapUsed = memory.heapUsed;
    const heapTotal = memory.heapTotal;
    const external = memory.external;
    const rss = memory.rss;

    return {
        message: "Hello, World",
        timestamp: new Date().toISOString(),
        metrics: {
            memory: {
                heapUsed: `${Math.round(heapUsed / 1024 / 1024)} MB`,
                heapTotal: `${Math.round(heapTotal / 1024 / 1024)} MB`,
                external: `${Math.round(external / 1024 / 1024)} MB`,
                rss: `${Math.round(rss / 1024 / 1024)} MB`,
                heapUsedPercentage: `${Math.round((heapUsed / heapTotal) * 100)}%`
            }
        }
    };
}

app.get('/', (c) => c.json({
    message: "Hello, World",
    timestamp: new Date().toISOString(),
}))

app.get('/metrics', (c) => c.json(getMemoryMetrics()))

export default {
    port: 3000,
    fetch: app.fetch,
}
```

Figure 5-6. Hono-based /metrics implementation

5.4 Benchmark Results

5.4.1 Cold Start Time

The cold start benchmark measured how quickly each backend stack became responsive after being launched from a stopped state. Five iterations were executed per stack, and the time to the first HTTP 200 OK response was recorded using nanosecond-resolution timestamps.

Table 5-1. Cold start comparison between the two stacks.

Stack	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5	Average
Bun + Hono	22 ms	22 ms	23 ms	21 ms	21 ms	21.8 ms
Node + Express	143 ms	144 ms	140 ms	139 ms	146 ms	142.2 ms

Bun consistently demonstrated sub-25 ms startup times, making it a strong candidate for cold-start-sensitive environments such as serverless platforms, auto-scaling containers, or short-lived CI/CD builds. Node.js with Express had a longer startup latency, averaging over 140 ms per launch.

5.4.2 JSON Response Throughput

This benchmark evaluated each stack's ability to handle concurrent HTTP traffic returning lightweight JSON payloads. The test simulated 100 clients, each using HTTP pipelining with 10 in-flight requests per connection, over a 30-second window.

Bun + Hono significantly outperformed **Node.js + Express** in both throughput and latency:

1. Bun + Hono
    - Average Latency: 4.51 ms
    - Average Throughput: 196791 requests/s
    - Total Requests: ~5.9 million
    - Total Data Transferred: 1.12 GB
2. Node.js + Express
    - Average Latency: 71.7 ms
    - Average Throughput: 13,840 requests/sec
    - Total Requests: ~416,000
    - Total Data Transferred 132 MB

These results highlight Bun's high-performance runtime and efficient request handling via Hono. The low latency and high throughput make it particularly suitable for SaaS applications with high user interaction or API-heavy operations.

5.4.3 Resource Usage Profiling

The third benchmark profiled how each stack behaved under sustained high load by collecting both system-level and application-level resource metrics. Each server was subjected to a 30-second autocannon test with 100 concurrent clients using pipelined requests.

The results below represent averaged metrics, sampled once per second throughout the test duration.

Bun + Hono

- **Average Memory (ps):** 2.20 MB
- **Heap Used (avg):** 31.39 MB
- **Heap Total:** 16.04 MB
- **External Memory:** 29.39 MB
- **RSS:** 117.54 MB
- **Heap Utilization:** 191.18%
- **Child Processes:** 1
- **CPU Usage:** 0%
- **Total Requests:** ~6.1 million

Node.js + Express

- **Average Memory (ps):** 2.06 MB
- **Heap Used (avg):** 33.26 MB
- **Heap Total:** 67.17 MB
- **External Memory:** 2 MB
- **RSS:** 112.78 MB
- **Heap Utilization:** 49.17%
- **Child Processes:** 1
- **CPU Usage:** 0%
- **Total Requests:** ~410,000

These results show that while both stacks remained stable under load, Bun + Hono demonstrated more efficient heap usage despite working with a smaller

total heap size. Its high heap utilization percentage suggests aggressive memory recycling, ideal for lightweight, dynamic workloads.

Node.js + Express, on the other hand, reserved significantly more heap space but used less of it, reflecting a more conservative memory management model. In high-load scenarios, Bun's tighter memory footprint and higher throughput indicate better scalability and efficiency.

5.5 Assessment of the Bun + Hono Stack

In conclusion, while both the Bun-Hono and Node.js-Express stacks can support modern SaaS workloads, the Bun-Hono stack demonstrated superior performance in cold start time, request throughput, and resource efficiency. These advantages make it a compelling choice for high-traffic, API-intensive applications. However, the Node.js-Express stack remains a mature and dependable option, particularly for more general-purpose use cases. Depending on the specific requirements of a SaaS application, either stack can be considered a viable backend solution.

6 REPLYTICS' TECHNICAL IMPLEMENTATION

This chapter provides a detailed overview of the technical implementation of Replytics, a full-stack SaaS platform developed to support users in tracking and analyzing strength training data.

The following subsections outline the system architecture, RESTful API design, authentication strategy, database schema, and deployment pipeline. Together, these components form the core of the Replytics platform, supporting its goal of turning gym performance data into actionable insights.

6.1 System Architecture and Component Breakdown

The Replytics platform follows a modular full-stack architecture based on the client-server model. The application is structured to ensure scalability, maintainability, and a clear separation of concerns between its frontend, backend, and data layers. This section outlines the major architectural components and how they interact to support the platform's functionality.

6.1.1 Architectural Overview

Replytics is built using a decoupled architecture in which the frontend and backend communicate via a RESTful API. A simplified overview of the system is shown in Figure 6-1, which illustrates the core components and their interactions.

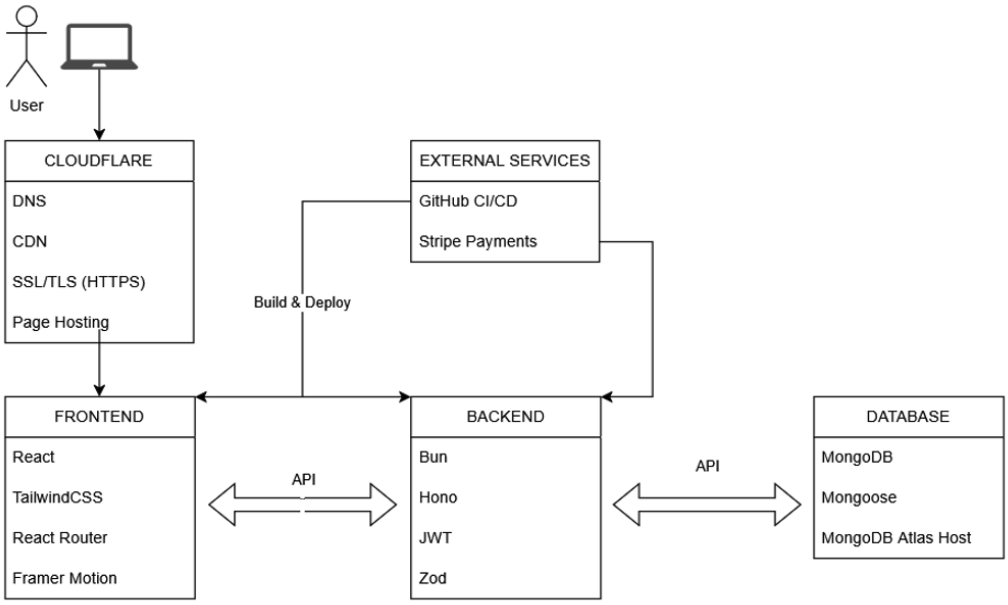

Figure 6-1. High-level architecture of Replytics, illustrating core components and their interactions within the SaaS system.

As shown in Figure 6-1, Replytics is designed using a decoupled architecture where the frontend and backend communicate via a RESTful API. Users access the application through a browser, with traffic first routed through Cloudflare, which manages DNS resolution, content delivery (CDN), HTTPS termination, and static page hosting via Cloudflare Pages.

The frontend is built using React, styled with TailwindCSS, and enhanced by React Router and Framer Motion to provide smooth navigation and transitions. API requests from the frontend are sent to the backend, which is implemented with Hono, a lightweight framework running on the Bun runtime. The backend handles business logic, input validation using Zod, and secure authentication using JSON Web Token (JWT).

Application data is stored in MongoDB, accessed through Mongoose, with collections such as users, workouts, and templates. The backend interacts with external services like Stripe for handling payments and GitHub CI/CD for automating deployment workflows. This modular architecture improves scalability, maintainability, and developer productivity.

6.2 API Design and Restful Implementation

Designing a robust and scalable API is fundamental for any SaaS platform, as the API often serves as the primary interface between clients and backend logic. A well-designed API should offer predictable and versioned endpoints, use appropriate HTTP methods and return consistent response formats.

The API design in Replytics aligns with widely recognized best practices for RESTful services, focusing on simplicity, scalability, and long-term maintainability (Microsoft, 2025).

Key goals of the API design include:

- Consistency: All endpoints follow uniform naming conventions and HTTP method usage.
- Versioning: All routes are scoped under a version prefix (e.g., /api/v1/) to allow for backward-compatible iterations.
- Security: API access is authenticated using JWT tokens and governed by role-based access control (RBAC).
- Future-readiness: Designed to support future extensions such as mobile clients, wearable integrations, or partner APIs.

As shown in Listing 6-1, the Replytics API defines a modular route structure organized under a versioned base path. Each resource (e.g., exercises, users) is defined in its own route file and mounted independently, allowing for scalable growth and easy maintenance.

```
// routes/index.ts
import type { Hono } from "hono";
import exerciseRoutes from "./exerciseRoutes";

const routes = (app: Hono) => {
  const api = app.basePath("/api");

  const v1 = api.basePath("/v1");
  v1.route("/exercises", exerciseRoutes);
};

export default routes;
```

Listing 6-1. Sample versioned API route definition in routes/index.ts

6.2.1 Endpoint Design and Organization

The Replytics API is structured according to RESTful principles, with a strong emphasis on clarity, consistency, and scalability. All endpoints follow a versioned structure (/api/v1/), allowing for future enhancements without breaking compatibility for existing clients.

Key design characteristics:

- Resource-Based Routing: The API exposes logically grouped resources such as users, workouts, and templates, with routes that reflect their hierarchical relationships.
- HTTP Method Semantics: Endpoints make appropriate use of standard HTTP methods:
  - **GET** for retrieving resources
  - **POST** for creating resources
  - **PUT** for updating resources
  - **DELETE** for removing resources
- Uniform Naming Conventions: Each route follows consistent naming patterns, which improves maintainability and developer onboarding.

```
1. GET    /api/v1/users/:id       → Retrieve user profile
2. POST   /api/v1/workouts        → Create new workout log
3. PUT    /api/v1/templates/:id   → Update a workout template
4. DELETE /api/v1/workouts/:id    → Delete a specific workout entry
```

Listing 6-2. Example of versioned endpoint structure for key Replytics resources.

Scalability-Oriented Features:

- Pagination and Filtering: List endpoints support pagination and optional query parameters to reduce response size and improve client performance.
- Versioning: All routes are scoped under a version prefix (/api/v1/), enabling non-breaking changes in the future.
- Structured Responses: All responses follow a uniform JSON structure, simplifying error handling and data parsing on the client side.

The API design emphasizes long-term maintainability and adaptability across different platforms, including web and mobile clients. By focusing on resource clarity

and predictable patterns, the Replytics API supports iterative feature growth without compromising usability.

6.2.2 Authentication, Authorization and Abuse Prevention

To protect user data and prevent abuse, the Replytics API incorporates multiple layers of security across both backend and frontend. The system is designed to ensure only authorized access, enforce rate limits, and validate request integrity, which is particularly important for a SaaS platform handling personal data.

Authentication and Authorization:
All protected routes require a valid JWT, passed in the Authorization header. Each token securely encodes the user ID and is signed using the HMAC algorithm to prevent tampering. If a token is missing, invalid, or expired, the server returns a 401 Unauthorized response. For more sensitive routes, middleware enforces role-based access control (RBAC), allowing only admin users to proceed. Unauthorized access attempts are met with a 403 Forbidden status.

Rate Limiting:
To defend against request flooding and abuse, a backend rate limiter tracks incoming requests per IP address. If a client exceeds the predefined threshold within a time window, subsequent requests are denied with a 429 Too Many Requests response.

Token Lifecycle Management:
Tokens are issued with a 24-hour expiration by default. Users who select the "remember me" option receive tokens valid for 30 days. This approach follows common best practices for session management, where longer expiration or refresh strategies are used to improve usability while maintaining security (Auth0, n.d.). It strikes a balance between usability and security, allowing persistent sessions without risking long-term access without revalidation.

Planned Cloudflare-Backed Hosting:
Replytics is designed to be deployed behind Cloudflare, leveraging DNS management, DDoS protection, and performance optimizations such as caching and

SSL termination. This infrastructure choice will add reliability and global scalability to the API without exposing the origin server directly to the public internet.

Frontend Security Measures:
On the client side, tokens are securely stored in localStorage and automatically attached to all API calls via a custom useApi hook. Protected pages use a ProtectedRoute component to guard against unauthorized access. If a token expires or becomes invalid, the user is automatically logged out and redirected to the login screen.

6.3 Frontend Architecture and Integration

The frontend of Replytics is built with React and TypeScript, following a modular, feature-oriented structure that emphasizes maintainability, performance, and good developer experience. The architecture cleanly separates views, logic, and utility functions, enabling scalable development and maintaining a clear separation of concerns.

6.3.1 Application Structure and Component Design

The Replytics frontend is organized within the src/ directory using a modular structure that enhances maintainability and code reuse. Key directories include:

- pages/: Page-level components that define core application views (e.g., dashboard, login, registration)
- layouts/: Reusable layout wrappers that structure shared UI regions like headers, sidebars, and containers
- hooks/: Custom React hooks for encapsulating logic such as API communication and authentication handling
- context/: React Context providers for managing global states like authentication
- utils/: Utility functions and shared logic for tasks like formatting, validation, or token decoding

This structure allows each part of the application to evolve independently, enhancing readability and minimizing tight coupling between components.

Replytics uses TailwindCSS for utility-first styling, `react-icons` for iconography, and `framer-motion` for animations and transitions. Together, these tools create a clean and consistent user interface and support smooth interactions across pages and components. Listing 6-3 shows an example of a React counter component that uses these three technologies.

```
import { useState } from 'react';
import { motion } from 'framer-motion';
import { FaPlus } from 'react-icons/fa';

const CounterCard = () => {
    const [count, setCount] = useState<number>(0);

    const increment = () => {
        setCount(prev => prev + 1);
    };

    return (
        <motion.div
            initial={{ opacity: 0, y: 10 }}
            animate={{ opacity: 1, y: 0 }}
            whileHover={{ scale: 1.02 }}
            transition={{ duration: 0.2 }}
            className="w-64 p-4 bg-gray-800 rounded-2xl shadow-md"
        >
            <h2 className="text-lg font-bold text-white mb-2">Counter</h2>
            <p className=" text-white font-semibold mb-4">{count}</p>
            <button
                onClick={increment}
                className="text-white cursor-pointer"
            >
                <FaPlus />
            </button>
        </motion.div>
    );
};

export default CounterCard;
```

Listing 6-3. Sample React component using TailwindCSS, Framer Motion and React Icons.

6.3.2 State Management

Replytics manages application state using React's built-in tools, separating global authentication logic from component-specific state. This separation ensures a responsive user interface and reliable session behavior across the application.

Authentication state is handled through a custom AuthContext powered by React's Context API, providing:

- Global access to login status, user data, and JWT tokens

- Token persistence using localStorage
- Automatic logout upon token expiration
- Full state reset on logout

Local UI state, such as form inputs and loading indicators, is handled using useState hooks. This ensures components remain isolated and easy to maintain.

6.3.3 Routing Strategy

Replytics organizes its routing configuration in a centralized file, separating routes into public and private categories. The routing system supports:

- Authentication-based route protection via a ProtectedRoute wrapper
- Dynamic route rendering based on the user's login state
- Consistent UI structure through layout components
- Fallback routing for unknown paths (404 handling)

The authentication state directly determines which routes are accessible, ensuring that users are redirected appropriately based on session validity. This close integration between state and routing results in a secure, intuitive navigation experience across the app.

6.3.4 API Communication and Type Safety

Replytics ensures consistent and secure interaction with the backend. This is achieved by combining a centralized API communication layer with a robust type system powered by TypeScript.

API Integration

The application uses a custom useApi hook, built on top of the Fetch API, to manage HTTP requests. This hook handles:

- Automatic inclusion of JWT tokens in headers
- Request state tracking (loading, error, success)
- Unified error handling and logging
- Support for CRUD operations (create, read, update, delete)

This centralized strategy minimizes boilerplate code and enforces consistency across components.

Shared Type System

All major data models (e.g., users, exercises, workouts, templates) are defined in shared TypeScript interfaces, stored in the shared/types/ directory. These types are used across:

- API response validation
- Form data modeling
- Component props
- State management
- Route parameter typing

This type-safe approach ensures that frontend and backend remain aligned, reducing bugs caused by mismatched data structures and improving developer productivity through IDE autocomplete and inline documentation.

The useApi hook manages all backend communication in a centralized and reusable way. It handles request state, error messages, and token-based authorization automatically. By abstracting repetitive logic such as header configuration and endpoint construction, it helps keep the codebase consistent and easier to maintain. Listing 6-4 shows a simplified example of the fetchData method inside the useApi hook.

```
const fetchData = async (path: string = "") => {
  try {
    setIsLoading(true);
    const response = await fetch(`${API_URL}${endpoint}${path}`, {
      headers: getHeaders(),
    });
    const result: ApiResponse<T> = await response.json();
    if (!result.success) throw new Error(result.message);
    setData(result.data);
    return result.data;
  } catch (err) {
    setError(err.message);
    throw err;
  } finally {
    setIsLoading(false);
  }
};
```

Listing 6-4. Simplified fetchData method inside the useApi hook. It manages state, error handling and authorization headers automatically.

6.4 Database Schema and Data Storage Strategies

Replytics uses MongoDB as its primary database due to its flexibility with document-based data structures and its ability to scale horizontally as the user base grows. The schema is organized around core domain entities such as users, workouts, templates, and exercises.

6.4.1 Data Modeling Approach

Replytics models its data using MongoDB's nested document structure rather than traditional relational tables. Core entities like workouts, users, and templates are stored in separate collections, while exercises and sets are embedded within workout documents to reflect the natural hierarchy of strength training data.

Each workout entry contains embedded subdocuments for multiple exercises and sets, enabling related data to be stored and retrieved in a single query. This structure reduces the need for complex joins and improves performance in read-heavy environments such as dashboards or mobile clients (MongoDB, n.d.).

Figure 6-2 shows a simplified example of a MongoDB document using field-value pairs. This format is well-suited for managing nested and user-generated training data.

Figure 6-2. Example of a MongoDB document structure (MongoDB, n.d.).

The schema design follows several key principles to ensure both performance and flexibility when working with training-related data:

- Embedding vs. Referencing: Closely related data (e.g., sets within exercises) is embedded to support atomic updates and fast retrievals. Relationships to users or external templates use references.
- Hierarchical Structure: Each workout entry is self-contained, storing all related sets and exercises, which improves performance for full-document retrieval.
- Use of Timestamps and Auditing Fields: Most collections include timestamps like createdAt and updatedAt, which support sorting, filtering, and historical tracking of changes.

6.4.2 Data Access Patterns

Replytics is optimized for read-heavy operations, such as dashboards, progress tracking, and workout history views. Data access in Replytics generally follow two main patterns:

- Write-heavy inserts during logging sessions, where users submit a workout with embedded exercises and sets.
- Frequent reads for displaying the recent workouts, templates, or user statistics.

Common access patterns include:

- Fetching recent workouts for a user, sorted by createdAt
- Updating individual sets or notes within a workout (handled as atomic updates to embedded documents)
- Querying personal records (PRs) or calculating training volume trends from exercise data

To support these operations:

- Workouts are stored with embedded exercise data to reduce join complexity
- Indexes are applied to fields like userId, createdAt, and exerciseId (where applicable) to improve query speed and filter performance across collections.

- Aggregation pipelines allow Replytics to compute metrics like weekly volume and personal bests without fetching and processing raw documents client-side.

This structure ensures low-latency queries for common use cases while keeping write operations efficient and allowing for flexible schema evolution.

Listing 6-5 illustrates a typical read operation in the backend, where workout templates for the authenticated user are retrieved from MongoDB and sorted alphabetically. The query uses find() and sort() while handling authentication context and returning a standardized JSON response.

```
// routes/templateController.ts

getUserTemplates: async (c: Context) => {
  try {
    const userId = c.get("userId");
    if (!userId) {
      return c.json({ success: false, message: "User ID not found" }, 401);
    }

    // Query templates belonging to the user, sorted alphabetically
    const templates = await Template.find({ owner: userId }).sort({ name: 1 });

    return c.json({ success: true, data: templates });
  } catch (error) {
    console.error("Error fetching templates:", error);
    return c.json({
      success: false,
      message: error instanceof Error
        ? error.message
        : "Failed to fetch templates",
    }, 500);
  }
};

```

Listing 6-5. Simplified backend function for retrieving authenticated user's workout templates, sorted alphabetically.

These patterns demonstrate how Replytics leverages MongoDB's strengths in handling dynamic, user-centric data, balancing flexible schema design with optimized querying and update efficiency.

6.5 CI/CD Pipeline with GitHub Actions

Replytics utilizes GitHub Actions to automate its continuous integration (CI) workflow. Each push to a development branch triggers automated testing across both

the backend and frontend, ensuring code quality and minimizing the risk of regressions prior to deployment.

The CI pipeline is structured into two parallel jobs:

- Backend Testing: Executes on an Ubuntu runner, installs dependencies using Bun, and runs unit tests with bun test.
- Frontend Testing: Uses Node.js (v20) to install frontend dependencies with npm ci, then runs automated tests using npm test.

Although full continuous deployment (CD) is not yet implemented, Replytics is designed with CD-readiness in mind. The GitHub Actions workflow is structured to easily extend into automated deployment once a production environment is added.

This future deployment flow will likely include:

- Post-merge automation: Automatically deploy new builds when code is merged into the main branch.
- Backend Deployment: The Hono + Bun backend can be containerized with Docker and deployed via cloud services.
- Frontend Deployment: The React frontend can be built and deployed to platforms like Cloudflare Pages.

Automating these steps will reduce manual effort, enable faster release cycles, and help maintain production stability as the platform scales.

7 CONCLUSIONS AND REFLECTIONS

This thesis explored the full-stack development of Replytics, a niche Software as a Service (SaaS) application designed to support strength training through structured workout logging and performance tracking. The project focused on building a modern, scalable architecture using React, TailwindCSS, Bun, Hono, and MongoDB. Core web functionalities such as user authentication, workout creation and editing, and workout history viewing were successfully implemented and tested, resulting in a working prototype that demonstrates the platform's key goals.

The original objectives outlined in Section 1 were achieved within the defined scope. A scalable SaaS prototype was implemented using a lightweight full-stack architecture, and the performance of the Bun runtime and Hono framework was benchmarked against the more established Node.js and Express stack. These tests, which evaluated cold start latency, concurrent JSON handling, and memory usage, confirmed that the modern stack offered clear performance advantages. On the frontend, a modular structure was used alongside centralized API communication through a custom hook and context-based state management, supporting maintainability and clean component design.

Several parts of the development process were completed smoothly, including the streamlined backend architecture and minimal DevOps pipeline using GitHub Actions. While additional features and extensions remain possible, the current implementation establishes a solid foundation for a niche SaaS product with a performance-oriented backend and a responsive, developer-friendly frontend.

Looking ahead, the platform can be extended with features such as adaptive workout planning, advanced user analytics, or expanded gamification elements like leveling systems and achievements. Deployment-ready architecture supports containerization and cloud hosting, with MongoDB suitable for migration to managed solutions such as Atlas. These possibilities highlight the flexibility and future scalability of the current design.

REFERENCES

Akamai. n.d. What is API Abuse? Retrieved 25.03.2025. https://www.akamai.com/glossary/what-is-api-abuse

Aleem, S., Ahmed, F., Batool, R., & Khattak, A. 2021. Empirical investigation of key factors for SaaS architecture. Retrieved 23.03.2025. https://ieeexplore.ieee.org/abstract/document/8669948

Allous, F. 2024. Optimizing User Engagement in Fitness Apps: A Comparative Analysis of Features and User Models. Retrieved 26.4.2025. https://essay.utwente.nl/100800/1/Allous_BA_EEMCS.pdf

Alumio, n.d. What is iPaaS? Retrieved 27.4.2025. https://www.alumio.com/the-ipaas-guide-all-about-ipaas

Amazon. n.d. What is a LAMP stack? Retrieved 26.03.2025. https://aws.amazon.com/what-is/lamp-stack/

Amazon Web Services. n.d. Tenant isolation - SaaS architecture fundamentals. Retrieved 15.4.2025. https://docs.aws.amazon.com/whitepapers/latest/saas-architecture-fundamentals/tenant-isolation.html

Autocannon. 2024. Autocannon (Version 5.0.0) [NPM package]. Retrieved 17.4.2025. https://www.npmjs.com/package/autocannon/v/5.0.0

Auth0. n.d. Refresh Tokens. Retrieved 26.4.2025. https://auth0.com/docs/secure/tokens/refresh-tokens

Browne, T. n.d. Create T3 App. Retrieved 26.03.2025. https://create.t3.gg/en/introduction

Bun. n.d. What is Bun? Retrieved 27.03.2025. https://bun.sh/docs

CloudZero. 2023. What Is SaaS Architecture? 10 Best Practices in 2024. Retrieved 15.4.2025. https://www.cloudzero.com/blog/saas-architecture/

Divami. 2024. Building a Scalable SaaS Architecture: Best Practices. Retrieved 15.4.2025. https://divami.com/news/building-a-scalable-saas-architecture-best-practices/

Dudjak, M., Martinović, G. 2020. An API-first Methodology for Designing a Microservice-based Backend as a Service Platform. Retrieved 23.03.2025. https://doi.org/10.5755/j01.itc.49.2.23757

European Commission. n.d. Data protection in the EU. Retrieved 15.4.2025. https://commission.europa.eu/law/law-topic/data-protection_en

freeCodeCamp. 2024. What is the Virtual DOM in React? Retrieved 27.03.2025. https://www.freecodecamp.org/news/what-is-the-virtual-dom-in-react/

GitHub. 2024. What is CI/CD? Retrieved 24.03.2025. https://github.com/resources/articles/devops/ci-cd

Hono. n.d. Hono. Retrieved 27.03.2025. https://hono.dev/docs/

HYCU. 2024. The Rise of SaaS and the Alarming Gap in Data Protection. Retrieved 15.4.2025. https://www.hycu.com/blog/rise-of-saas-and-the-alarming-gap-in-data-protection

Joshi, S., & Kumari, U. 2016. Load Balancing in Cloud Computing: Challenges & Issues. Retrieved 08.03.2025. https://ieeexplore.ieee.org/abstract/document/7917945

Kratzke, N. 2018. A brief history of cloud application architectures. Retrieved 17.4.2025. https://www.mdpi.com/2076-3417/8/8/1368

LaunchDarkly. 2024. DevOps vs. CI/CD: Complete Guide to Better Software Delivery. Retrieved 24.03.2025. https://launchdarkly.com/blog/devops-vs-cicd/

Liao, H. 2009. Design of SaaS-Based Software Architecture. Retrieved 05.03.2025. https://ieeexplore.ieee.org/abstract/document/5260685/authors#authors

Microsoft. 2025. RESTful web API design. Retrieved 18.4.2025. https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design

MongoDB. n.d. Advantages of MongoDB. Retrieved 27.03.2025. https://www.mongodb.com/resources/compare/advantages-of-mongodb

MongoDB. n.d. Database Scaling. Retrieved 27.4.2025. https://www.mongodb.com/resources/basics/scaling

MongoDB. n.d. MERN Stack Explained. Retrieved 26.03.2025. https://www.mongodb.com/resources/languages/mern-stack

MongoDB. n.d. Why Use MongoDB and When to Use It? Retrieved 19.4.2025. https://www.mongodb.com/resources/products/fundamentals/why-use-mongodb

Netguru. 2023. What is SaaS? Retrieved 27.2.2025. (note: if you cited them earlier for SaaS background, otherwise remove this if not used)

Node.js. n.d. Process.memoryUsage(). Retrieved 17.4.2025. https://nodejs.org/api/process.html#processmemoryusage

PayPro Global. 2025. What is Load Balancing in Cloud Computing? Retrieved 09.03.2025. https://payproglobal.com/answers/what-is-load-balancing-in-cloud-computing/

PayPro Global. 2025. What is SaaS Infrastructure? Retrieved 08.03.2025. https://payproglobal.com/answers/what-is-saas-infrastructure/

Pedreira, O., Silva-Coira, F., Saavedra Places, Á., Luaces, M. R., & González Folgueira, L. 2019. Applying Feature-Oriented Software Development in SaaS Systems: Real Experience, Measurements, and Findings. Retrieved 18.03.2025. https://ieeexplore.ieee.org/abstract/document/10251854

Tiwari, K., & Joshi, S. 2014. A review of data security and privacy issues over SaaS. Retrieved 18.03.2025. https://ieeexplore.ieee.org/abstract/document/7238432

Vashistha, A., & Ahmed, P. 2012. SaaS Multi-Tenancy Isolation Testing-Challenges and Issues. Retrieved 24.03.2025.

https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=44571e8644e093442c152b54742a1e1fb4f37f4e

Yasrab, R. 2018. Platform-as-a-Service (PaaS): The Next Hype of Cloud Computing. Retrieved 27.2.2025. https://arxiv.org/abs/1804.10811

ATTACHMENTS

APPENDICES

Appendix 1. Bash Script - coldStart.sh

```bash
#!/bin/bash

# Color definitions for terminal output
GREEN='\033[0;32m'
RED='\033[0;31m'
NC='\033[0m' # No Color

# Kill any process occupying the specified port
kill_port() {
    lsof -ti:$1 | xargs kill -9 2>/dev/null || true
}

# Measure cold start time for a server
measure_cold_start() {
    local name=$1
    local port=$2
    local start_cmd=$3
    local iterations=$4

    echo -e "\n${GREEN}Testing $name cold start (${iterations} iterations)${NC}"
    echo "----------------------------------------"

    for i in $(seq 1 $iterations); do
        kill_port $port

        eval "$start_cmd" &
        SERVER_PID=$!

        START_TIME=$(date +%s%N)

        # Poll until server responds successfully
        while ! curl -s "http://localhost:$port" > /dev/null; do
            sleep 0.001
        done

        END_TIME=$(date +%s%N)
        DURATION=$(( ($END_TIME - $START_TIME) / 1000000 ))

        echo "Iteration $i: ${DURATION}ms"

        kill $SERVER_PID 2>/dev/null
        sleep 1
    done
}

# Run cold start test for both stacks
ITERATIONS=5
measure_cold_start "Bun + Hono" 3000 "cd bun-hono && bun run index.ts" $ITERATIONS
measure_cold_start "Node + Express" 3001 "cd node-express && npm start" $ITERATIONS

echo -e "\n${GREEN}Benchmark completed!${NC}"

```

Appendix 2. Bash Script - json-benchmark.sh

```bash
#!/bin/bash

# Define colors for terminal output
GREEN='\033[0;32m'
RED='\033[0;31m'
NC='\033[0m' # No Color

```

```
# Kill any process using the specified port
kill_port() {
    lsof -ti:$1 | xargs kill -9 2>/dev/null || true
}

# Run JSON response benchmark for a given stack
run_benchmark() {
    local name=$1
    local port=$2
    local start_cmd=$3

    echo -e "\n${GREEN}Testing $name JSON Response Performance${NC}"
    echo "----------------------------------------"

    kill_port $port

    # Start the server
    eval "$start_cmd" &
    SERVER_PID=$!

    # Give the server time to initialize
    sleep 2

    # Run load test with autocannon
    autocannon -c 100 -d 30 -p 10 http://localhost:$port

    # Stop the server
    kill $SERVER_PID
    sleep 1
}

# Run benchmarks for both stacks
run_benchmark "Bun + Hono" 3000 "cd bun-hono && bun run index.ts"
run_benchmark "Node + Express" 3001 "cd node-express && npm start"

echo -e "\n${GREEN}Benchmark completed!${NC}"

```

Appendix 3. Bash Script - resource-benchmark.sh

```
#!/bin/bash

# Colors for output
GREEN='\033[0;32m'
YELLOW='\033[1;33m'
NC='\033[0m'

# Function to kill processes on specific ports
kill_port() {
    lsof -ti:$1 | xargs kill -9 2>/dev/null || true
}

# Function to get memory usage in MB
get_memory_usage() {
    local pid=$1
    ps -o rss= -p $pid 2>/dev/null | awk '{print $1/1024 " MB"}'
}

# Function to get CPU usage percentage
get_cpu_usage() {
    local pid=$1
    ps -o pcpu= -p $pid 2>/dev/null
}

# Function to get child processes
get_child_processes() {
    local pid=$1
    pgrep -P $pid | wc -l
}

```

```
# Function to get detailed metrics from the application
get_app_metrics() {
    local port=$1
    curl -s http://localhost:$port/metrics
}

# Function to run benchmark with resource monitoring
run_benchmark() {
    local name=$1
    local port=$2
    local start_cmd=$3

    echo -e "\n${GREEN}Testing $name Resource Usage${NC}"
    echo "----------------------------------------"

    # Kill any existing process on the port
    kill_port $port

    # Start the server in background
    eval "$start_cmd" &
    SERVER_PID=$!

    # Wait for server to start
    sleep 2

    # Create results directory if it doesn't exist
    mkdir -p results

    # Start resource monitoring in background
    {
        echo "Timestamp,Memory(MB),CPU(%),ChildProcesses,HeapUsed(MB),HeapTo-
tal(MB),External(MB),RSS(MB),HeapUsedPercentage" > "results/${name}-resources.csv"
        while kill -0 $SERVER_PID 2>/dev/null; do
            memory=$(get_memory_usage $SERVER_PID)
            cpu=$(get_cpu_usage $SERVER_PID)
            children=$(get_child_processes $SERVER_PID)

            # Get application metrics
            app_metrics=$(get_app_metrics $port)
            heap_used=$(echo "$app_metrics" | jq -r '.metrics.memory.heapUsed' | sed
's/ MB//')
            heap_total=$(echo "$app_metrics" | jq -r '.metrics.memory.heapTotal' |
sed 's/ MB//')
            external=$(echo "$app_metrics" | jq -r '.metrics.memory.external' | sed
's/ MB//')
            rss=$(echo "$app_metrics" | jq -r '.metrics.memory.rss' | sed 's/ MB//')
            heap_percentage=$(echo "$app_metrics" | jq -r '.met-
rics.memory.heapUsedPercentage' | sed 's/%//')

            echo "$(date +%s),${memory// MB/},$cpu,$children,$heap_used,$heap_to-
tal,$external,$rss,$heap_percentage" >> "results/${name}-resources.csv"
            sleep 1
        done
    } &
    MONITOR_PID=$!

    # Run autocannon
    echo -e "${YELLOW}Running performance test...${NC}"
    autocannon \
        -c 100 \
        -d 30 \
        -p 10 \
        --headers "Accept: application/json" \
        --headers "Content-Type: application/json" \
        --json-report \
        --output-json "results/${name}-performance.json" \
        http://localhost:$port

    # Kill the server and monitoring
    kill $SERVER_PID $MONITOR_PID 2>/dev/null
    sleep 1

    # Calculate and display average resource usage
```

```
    echo -e "\n${YELLOW}Average Resource Usage:${NC}"
    awk -F',' 'NR>1 {
        sum_mem+=$2;
        sum_cpu+=$3;
        sum_children+=$4;
        sum_heap_used+=$5;
        sum_heap_total+=$6;
        sum_external+=$7;
        sum_rss+=$8;
        sum_heap_percentage+=$9;
        count++
    }
    END {
        print "Memory: " sum_mem/count " MB"
        print "CPU: " sum_cpu/count "%"
        print "Child Processes: " sum_children/count
        print "Heap Used: " sum_heap_used/count " MB"
        print "Heap Total: " sum_heap_total/count " MB"
        print "External: " sum_external/count " MB"
        print "RSS: " sum_rss/count " MB"
        print "Heap Used %: " sum_heap_percentage/count "%"
    }' "results/${name}-resources.csv"
}

echo -e "${GREEN}Starting Resource Usage Benchmark${NC}"
echo "This test will measure:"
echo "- Memory usage over time"
echo "- CPU usage over time"
echo "- Child processes"
echo "- Heap usage and garbage collection"
echo "- External memory"
echo "- RSS (Resident Set Size)"
echo "- Performance metrics"
echo "----------------------------------------"

# Test Bun+Hono
run_benchmark "bun-hono" 3000 "cd bun-hono && bun run index.ts"

# Test Node+Express
run_benchmark "node-express" 3001 "cd node-express && npm start"

echo -e "\n${GREEN}Benchmark completed! Results saved in results/ directory${NC}"

```

Appendix 4. Survey: Evaluating User Satisfaction with Current Strength Training Apps

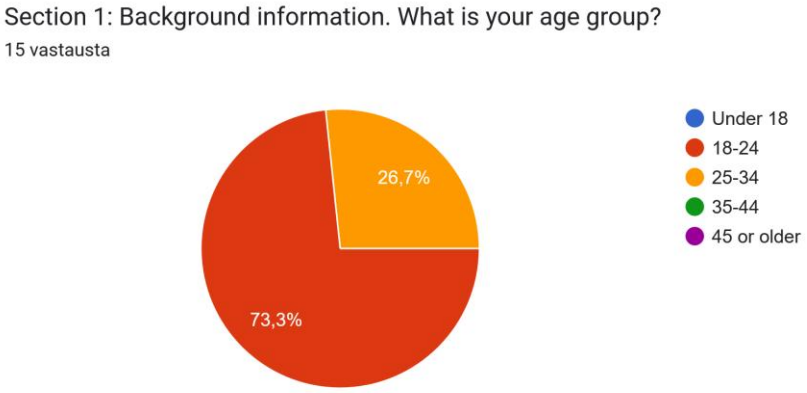

Figure A4.1. Age group distribution of respondents.

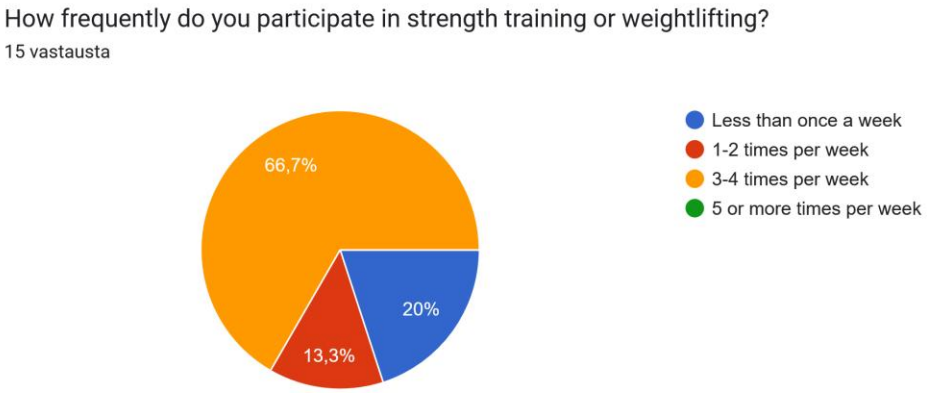

Figure A4.2. Frequency of strength training or weightlifting.

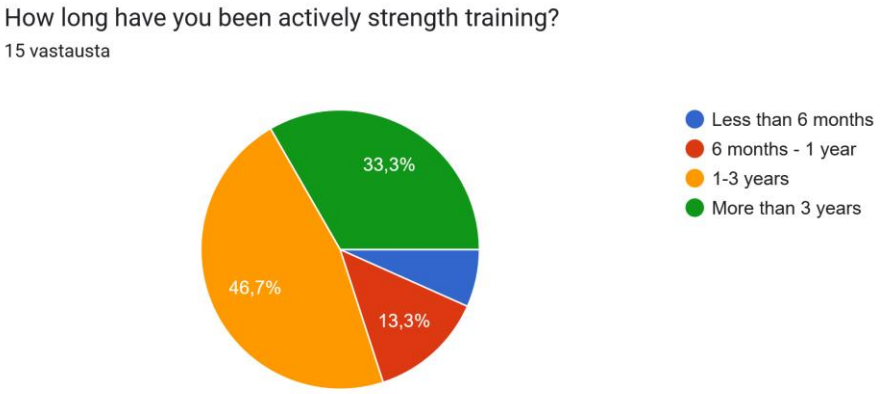

Figure A4.3. Duration of active strength training experience.

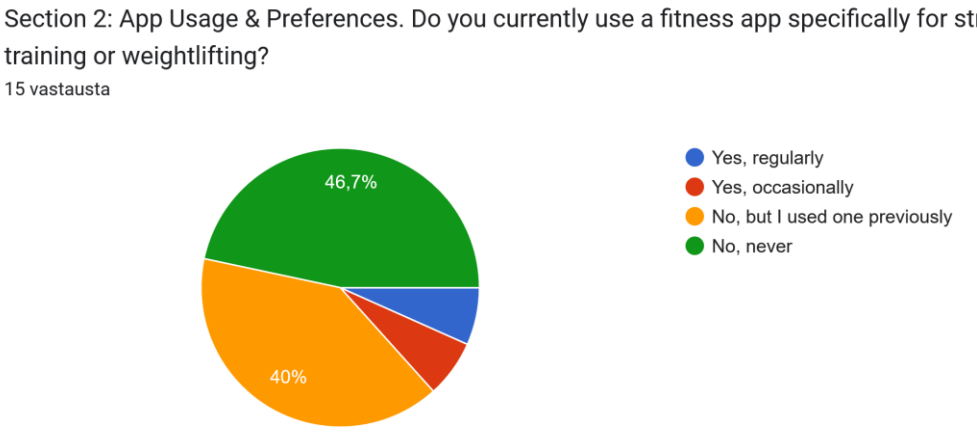

Figure A4.4. Use of fitness apps for strength training.

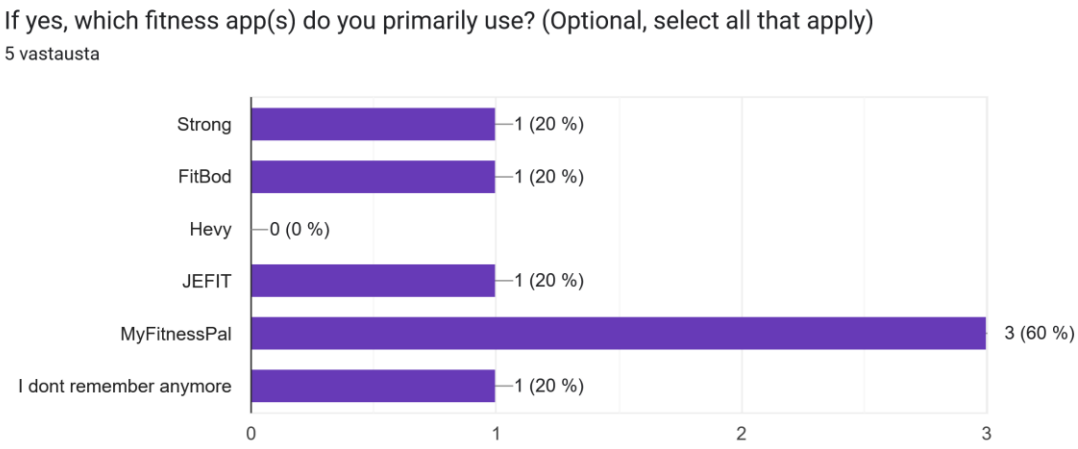

Figure A4.5. Fitness apps currently used (multiple selections allowed).

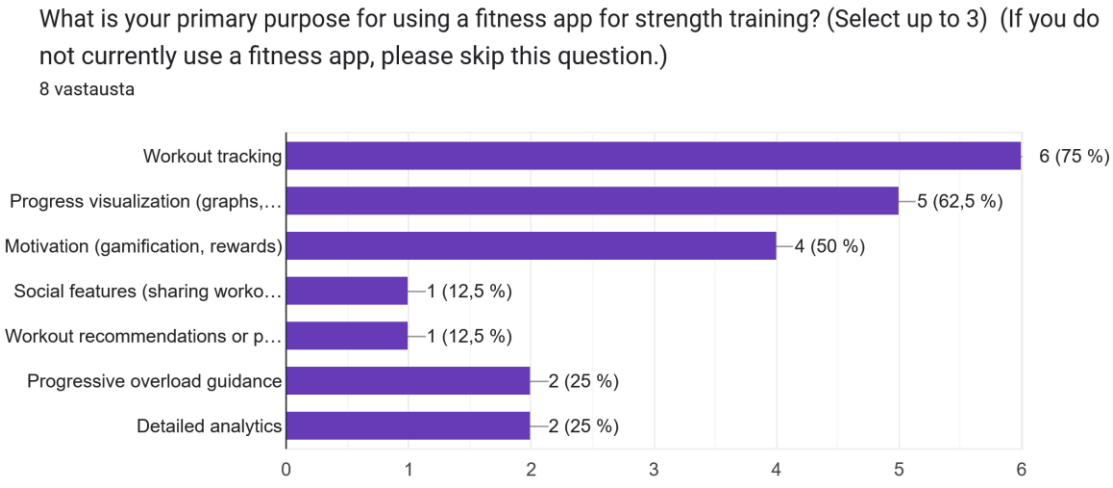

Figure A4.6. Primary purposes for using a fitness app (multiple selections allowed).

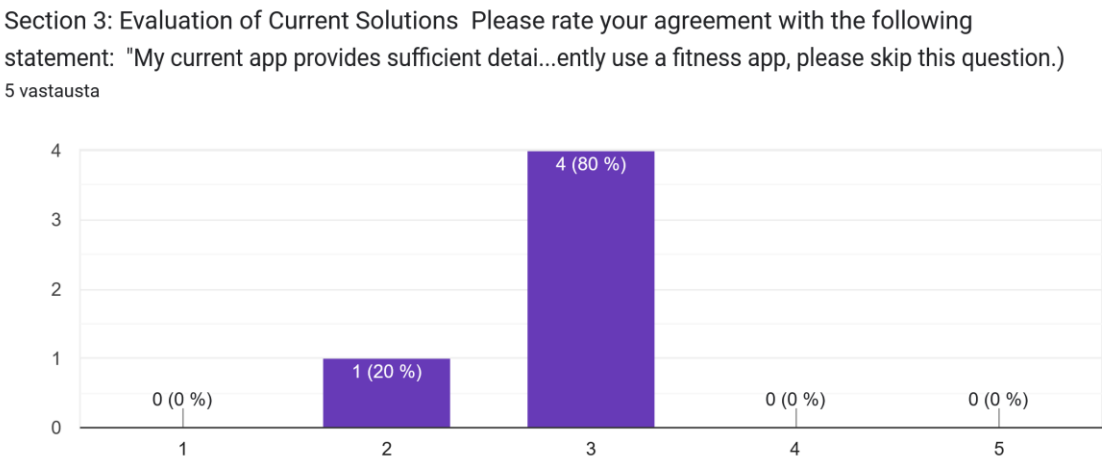

Figure A4.7. Overall evaluation of existing strength training app solutions.

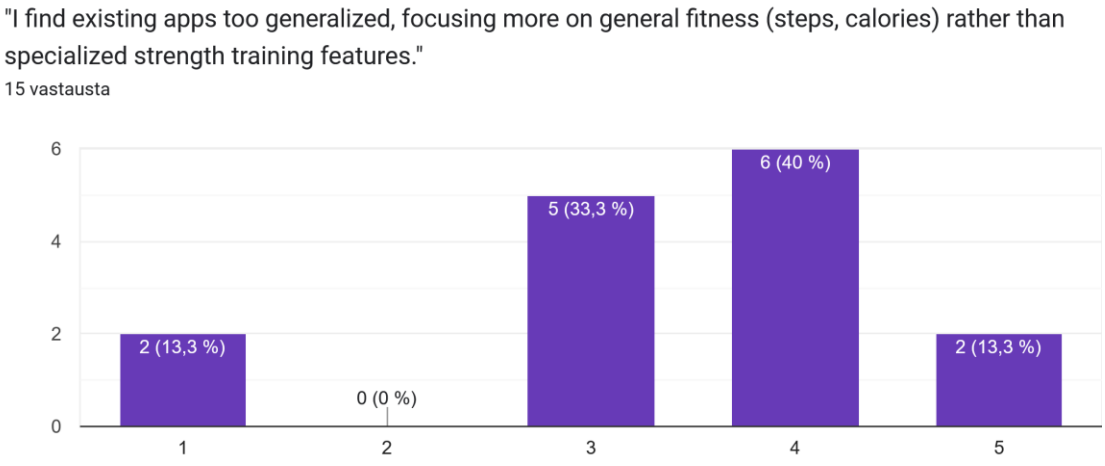

Figure A4.8. Agreement with the statement: *"I find existing apps too generalized, focusing more on general fitness rather than specialized strength training features."*

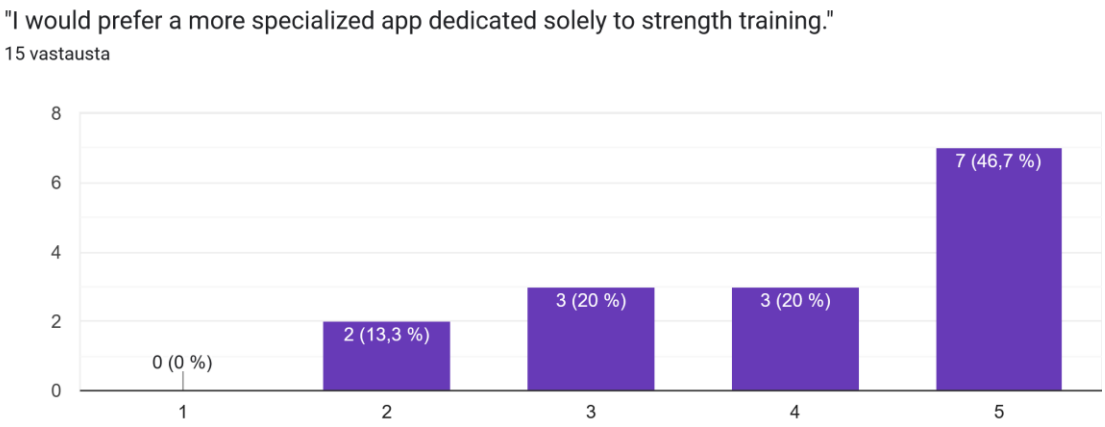

Figure A4.9. Agreement with the statement: *"I would prefer a more specialized app dedicated solely to strength training."*

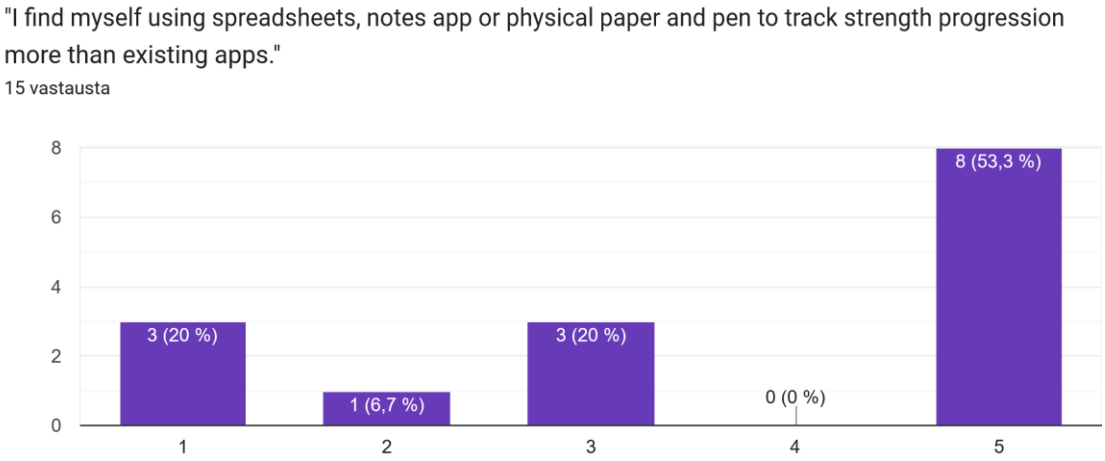

Figure A4.10. Agreement with the statement: *"I find myself using spreadsheets, notes apps, or physical paper and pen to track strength progression more than existing apps."*

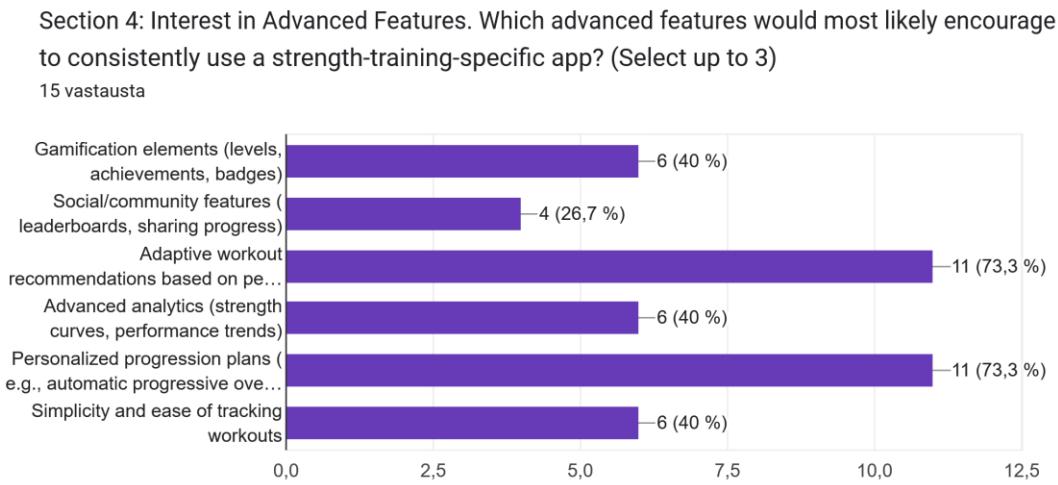

Figure A4.11. Most desired advanced features that would encourage consistent use of a strength-training-specific app (multiple selections allowed).