WEDNESDAY SCHOOL AND PROFESSIONAL ACTIVITY

Field: 18. Informatics

Microservices implementation architecture and dynamic plugin ecosystem for robust social platform using Kubernetes orchestration

Patrick Stohanzl

Pardubice 2025

WEDNESDAY SCHOOL AND PROFESSIONAL A´ ACTIVITY

MICROSERVIS IMPLEMENTATION ´ ARCHITECTURE AND DYNAMIC ´ PLUGIN ECOSYSTEM EMU PRO ROBUST SOCIAL PLATFORMS USE ÿ´IM ORCHESTRATION KUBERNETICS

IMPLEMENTING MICROSERVICES ARCHITECTURE AND DYNAMIC PLUGIN ECOSYSTEM FOR A ROBUST SOCIAL PLATFORM USING KUBERNETES ORCHESTRATION

AUTHOR Patrik Stohanzl

DELTA SCHOOL - Secondary School of Informatics and Economics

Pardubice Region

SUPERVISOR RNDr. Jan Koupil, PhD.

DEPARTMENT 18. Informatics

Pardubice 2025

Declaration

I declare that my work on the topic of Implementing a Microservice Architecture and a dynamic plugin ecosystem for a robust social platform with using Kubernetes orchestration, I developed it independently under the guidance of RNDr. Jana Koupila, PhD. using professional literature and other information sources, all of which are cited in the work and listed in the reference list.
literature at the end of the work.

I further declare that the printed and electronic versions of the SOC work are identical and that I have no compelling reason against making this work available in accordance with the law. C. 121/2000 Coll., on copyright, on rights related to copyright and amendments to certain laws (Copyright Act), as amended.

In Pardubice on: _____ _____

Patrick Stohanzl

Thanks

I thank my supervisor RNDr. Jan Koupil, PhD. for his dedicated help,
the stimulating comments and endless patience he provided me during my
work.

Abstract

This work deals with the design and implementation of a modern social platform So-cigy, which addresses the identified shortcomings of current social networks through innovative approach to architecture, security and extensibility. The paper analyses the current state of social platforms and identifies their main shortcomings, particularly in the areas of user control, algorithm transparency and
safety.

Based on this analysis, a robust microservice architecture is proposed.
Using Kubernetes for orchestration, HashiCorp Consul for Service Mesh
and PostgreSQL with Patroni to ensure high data availability. The key innovative element is the implementation of a dynamic plugin system based on
on WebAssembly, which enables the safe execution of third-party code in
sandboxed environment.

The work describes in detail the implementation of individual system components, including authentication mechanisms using the FIDO2 standard (Passkeys), microservice ecosystem and client applications for mobile and web platforms. Special attention is paid to the development of our own Rust framework
for creating plugins with support for JSX-like syntax, which greatly simplifies extension development.

Benchmarking results show excellent performance of the Rust implementation plugin system that reaches up to 200,000 fps when processing virtual
DOM. The work also identifies limitations of the current implementation and proposes future development directions, including full integration of plugins into the main application, implementation of multi-cluster solutions and advanced algorithms for personalization

content.

Contents

Chapter 1

Introduction

1.1 Analysis of the need for a new social platform

Social media has become an integral part of everyday life. Looking at the numbers - over 5.24 billion active users worldwide spend an average of more than two hours a day on these platforms. It is clear that this is a significant phenomenon with a far-reaching impact on society. This chapter analyzes the need for the development of a new social platform in the context of current trends, technological possibilities and shortcomings of existing solutions.

1.1.1 Current state of social platforms

The social media ecosystem is undergoing a significant transformation in 2025. According to the latest data (1), the average internet user uses an average of 7 different social platforms per month. This fragmentation points to the growing specialization of platforms and the diversification of user needs in digital pro-century.

When analyzing current trends, several key changes can be identified. Content distribution algorithms are undergoing a recalibration, with platforms no longer judging success solely on the number of views, but focusing more on the quality of interactions and user retention. This shift reflects the growing competition for users' attention(2).

At the same time, the information space is becoming saturated with content generated artificial intelligence, which creates new challenges for maintaining authenticity. Users they increasingly prefer authentic content and transparent communication. Successful creators no longer base their strategy primarily on the volume of content, but on an analytically based approach to creating relevant content for specific targets. groups.

These changes create space for new approaches to platform design, that would better reflect changing user preferences and address shortcomings of current platforms in the areas of authenticity, personalization and control over content.

1.1.2 The issue of a business-oriented approach

When analyzing current social networks, it is possible to identify, at least from the user perspective, aspects, important aspects. Dominant platforms are primarily oriented to commercial interests, not to the needs of users. This is reflected in several key aspects.

Closed algorithms function as non-transparent mechanisms that do not provide sufficient control over the content consumed. They are designed to maximize user engagement and time spent on the platform, which leads to the creation of ""filter bubbles" (3) and supports information wars (IW) (4).

Another problem is the lack of personalization options. Users have minimal control over the appearance and functionality of the interface, which limits the ability to adapt the platform to their own needs. Given the dynamism This is a significant problem for social platforms. When implementing new features users are forced to accept all changes without the ability to retain preferred aspects of previous versions. This lack of control poses a long-term problem for the relationship between the platform and its users.

The economic model of current platforms is often disadvantageous for creators content. Platforms typically take up to 45% of creators' revenue and do not provide dynamic models to support their growth. For example, YouTube keeps 45%

of all direct income to creators, which limits the potential of the creative economy and may lead to a drain of talent to alternative platforms(5).

1.1.3 Technological shortcomings of current solutions

From a technological perspective, current social platforms have several significant shortcomings. They lack quality cross-platform support, which hinders a consistent user experience across devices. Web interfaces are often unoptimized, primarily adapted to mobile devices without adequately utilizing the capabilities of desktop browsers.

In the area of security and authentication, the dominant platforms remain:
Back. Outdated authentication mechanisms based primarily on passwords pose a security risk and degrade the user experience. The lack of support for more modern approaches, such as Passkeys, which offer a higher level of security while simplifying the login process, indicates technological stagnation in this area.

The protection of vulnerable groups, especially children, is inadequately addressed. Im-Implemented mechanisms are often limited to basic time limits without more sophisticated approaches to protecting against inappropriate content. This approach does not respond to the growing emphasis on digital wellbeing and online safety.

In cases where platforms implement basic filtering mechanisms
to protect younger users, as in the case of YouTube Kids, there is fragmentation of the user environment in the form of separate applications without adequate integration with the main platform. Specialized versions have significant limitations in content personalization and lack mechanisms for preserving preferences when switching between environments. The absence of a gradual transition model between platforms for different age categories poses a significant problem, especially for adolescent users who are switching from children's platforms but are not sufficiently protected from potentially inappropriate content in adult environments.

1.1.4 Summary of the analysis performed

Based on the analysis performed, it can be concluded that there is significant room for the development of a new social platform that would address the identified shortcomings of current solutions. The combination of a user-oriented approach, advanced architectural principles, modern authentication mechanisms and personalizable platforms represents a promising direction for the implementation of such a platform.

1.2 Objectives of the work

The main goal of this work is to design and implement a robust social platform with an integrated ecosystem of plugins that addresses the identified shortcomings of current solutions. The work focuses on creating a comprehensive architecture that will reflect current technological trends and at the same time will provide user-oriented access to social interactions in digital space.

1.2.1 Specification of objectives

The primary goal is to develop a modular social platform architecture that enables flexible extensibility through a plugin-based system on WebAssembly technology. This architecture is designed with the following in mind: Scalability, security and maintainability of the code.

- A secondary goal is to implement advanced authentication mechanisms with support for the FIDO2 standard (6) and Passkeys technology.
- The third goal is to create a multiplatform solution with optimized user interface for different devices that will provide a consistent user experience across desktop and mobile platforms.
- The fourth goal is to design and implement transparent content distribution algorithms that give users greater control over their content.

zoomed content and allows personalization of the information flow according to individual preferences.

- The fifth goal is to create a gradual transition model for different age categories of users that will ensure adequate protection of vulnerable groups while maintaining personalization options and user freedom.
- The sixth goal is to implement an economic model that supports content creators through fairer revenue distribution and dynamic tools to support community growth.

The final goal is to evaluate the created solution in terms of user experience, technical efficiency, and potential for long-term sustainability in the dynamically changing social media environment.

Achieving these goals should lead to the creation of a social platform that not only addresses the current shortcomings of existing solutions, but also provides a flexible foundation for future innovations in the field of social interactions in digital space.

1.3 Work structure

This work is structured into eight main chapters that systematically cover the entire process of designing and implementing a robust social platform with a plugin ecosystem.

The introductory chapter presents the issue of social platforms in the contemporary digital environment. It analyzes the need for a new social platform in the context of the identified shortcomings of existing solutions and sets work goals that reflect the ambition to create a user-oriented social platform in collaboration on modularity and extensibility.

The second chapter is devoted to a detailed analysis of the market and existing solutions, while defines the technological prerequisites and the project framework. Special attention is dedicated to the technological stack including Kubernetes and

binding other technologies. The chapter also specifies functional and non-functional requirements that form the basis for the subsequent architecture design.

The third chapter presents a comprehensive design of the system architecture. Getting Started an overall overview of the architecture with emphasis on the client and server parts, continues with a description of the microservices architecture implemented on the platform Kubernetes and focuses on communication layers, security aspects, and It also discusses the limitations of the chosen environment.

The fourth chapter focuses on the detailed implementation of key system components. It describes key management using HashiCorp Vault, implementation of authentication mechanisms and security features, structure of the micro-service ecosystem, and implementation of the frontend application in React Native. Expo and Next.js.

The fifth chapter is dedicated to the ecosystem of user plugins, which presents one of the main innovative elements of the proposed solution. It describes the application layer including API versioning and mechanisms for registering and processing events, plugin layer focused on sandboxing for native devices and implementation of the user interface in conjunction with dynamic rendering of components.

The sixth chapter deals with monitoring and benchmarking of implementation. of the system, describing the tools and methodologies used to monitor the performance and stability of the platform.

The seventh chapter brings a discussion and evaluation of the achieved results. It assesses the degree of fulfillment of the set goals, compares the implemented solution with existing alternatives and identifies the limitations of the current implementation along with proposals for future development.

The final chapter summarizes the key findings obtained during the project implementation and formulates recommendations for future research and practice in the field of development. social platforms.

The work is supplemented with appendices containing architecture diagrams, examples code, configuration files, and other relevant documentation that provide a more detailed insight into the technical aspects of the implemented solution.

Chapter 2

Analysis and assumptions

2.1 Market analysis and existing solutions

Unlike the general analysis in the introductory chapter, this section focuses on specific technical implementations of existing social platforms and their architectural approaches. The goal is to identify specific technical aspects that can be improved in the proposed solution.

2.1.1 Technical implementations of dominant platforms

Currently, social platforms use different architectural approaches to solve problems of scalability, security and user experience. Meta (Facebook, Instagram) implemented a large-scale microservices architecture with pro-environment solutions for scaling and load distribution. The company developed its own container management system similar to Kubernetes and specialized performance monitoring tools. However, in terms of authentication, the platform relies primarily on traditional passwords supplemented by two-factor authentication, without full support for modern standards such as FIDO2. (7)

X (formerly Twitter) has undergone a significant architectural transformation, replacing its original monolithic application with a microservices architecture. The platform uses a combination of proprietary and open-source technologies to manage its infrastructure. A notable aspect of X's architecture is its emphasis on real-time

data processing, which creates specific challenges for scaling and consistency. (8)

TikTok represents a modern approach with an emphasis on efficient distribution of video content and advanced algorithms for personalization. The platform uses a cloud-native approach and proprietary solutions for processing multimedia content. From However, from a security perspective, TikTok faces criticism for its lack of transparency in the processing of user data. (9)

2.2 Technological prerequisites and framework for project

Following the identified technological gaps of existing platforms
This chapter presents the technology stack used in the implementation of the proposed solution. The selected technologies were carefully selected with regard to their ability to address identified deficiencies and provide a robust foundation for a modern social platform with a dynamic plugin system.

While dominant platforms like Meta and Twitter have developed proprietary solutions for container and microservices management, the proposed solution is based on standardized open-source technologies that allow for greater flexibility and transparency. Instead of creating closed ecosystems, the focus is on implementing an open architecture that supports extensibility and interoperability.

2.2.1 Use of technology

Kubernetes serves as a platform for orchestrating containerized applications.
It enables efficient scaling, automated management, and deployment of individual microservices. Key benefits include robustness and management flexibility.
cluster. The implementation uses advanced features such as Ingress for traffic routing, network communication rules (Network Policies), secrets (Secrets) and configuration maps (ConfigMaps) for configuration management and certificate manager for automatic management of SSL certificates. Kubernetes option

netes compared to alternatives like Docker Swarm was made primarily due to the need for higher scalability and a more robust ecosystem for managing micro-services, which are key to the architecture of a modern social network.

Consul is a tool designed for managing, registering, and configuring microservices. Thanks to the implementation of Service Mesh, secure communication between individual services can be achieved via mTLS. Within the project, Consul is used in three key roles: as an API Gateway for centralized management of access to the API, as a Service Mesh for secure communication between services using Proxy Defaults, and as a Terminating Gateway for secure communication with external services. The choice of Consul over alternatives such as Istio was made based on its lower hardware resource requirements while maintaining key functionality.

The system uses PostgreSQL as a relational database with the Spillo and Patroni extensions for high availability. This combination enables automatic data replication and failover in the event of a primary node failure.
The advantages are robustness, support for advanced query mechanisms and the ability to use the postgresql operator to ensure high availability. Spillo was chosen for its simplicity in configuring Patroni with PostgreSQL and setting up database connections.

HashiCorp Vault is used to securely manage secrets and encryption keys.
It enables dynamic distribution and secure storage of sensitive data, minimizing the risk of data leakage. Within the project, Vault is planned as a key component for managing encryption keys and securing sensitive data.
It is the only self-hosted key management solution (KMS) that is well integrated with Consul and provides comprehensive data security capabilities in a microservices environment.

For the development of mobile applications, the React Native Expo framework was chosen, which enables rapid development of multiplatform applications. Expo supports easy integration with native device functions and ensures a consistent user experience across different operating systems. For the web part of the application, Next.js is used as a modern React framework providing optimal development

environment and performance optimization. The choice of these technologies was made based on their cross-platform nature and synergy with existing knowledge of the React ecosystem. As part of the implementation, native modules in Kotlin were developed to support WebAssembly and Passkey authentication,
which are not standard parts of React Native Expo.

WebAssembly is a technology that enables the execution of high-performance code in a sandboxed environment. The project uses it to implement a dynamic plugin system that ensures modularity and safe extensions.
functionality. WASM was chosen primarily for its cross-platform nature and
a robust security model that ensures that running third-party plugins
The pages do not compromise the stability and security of the application.

Cloudflare R2 network is used for efficient distribution of static content
CDN, which provides a global network for fast delivery of media content. This solution significantly reduces latency when loading images and videos and
while reducing the load on the backend infrastructure. Cloudflare R2 also offers protection against DDoS attacks and effective caching mechanisms that optimize operating costs.

To ensure comprehensive monitoring and logging, it is implemented
Grafana, Loki and Prometheus stack running in a Kubernetes cluster outside of Consul Service Mesh. This stack provides robust performance monitoring tools,
anomaly detection and log analysis, which is crucial for maintaining stability and performance complex microservice environment.

2.2.2 High Availability (HA)

High availability is a key aspect of architectural design, ensuring
uninterrupted system operation even in the event of failure of individual components. In In the context of the implemented solution, HA includes several levels of redundancy. At the microservice level, high availability is ensured through the Consul Service Mesh, which enables the implementation of a Mesh Gateway for communication between multiple clusters. This approach ensures efficient scaling of microservices within the Service Mesh. At the database level, the

high availability using Spillo with Patroni and PostgreSQL, which provides automatic data replication and a voting mechanism for determining the master node in in the event of a failure. At the infrastructure level, Kubernetes is used for automatic recovery after a failure and load redistribution between available nodes. This multi-layered approach to high availability is essential to ensure the reliable operation of a globally accessible social network with an expected high by the number of currently connected users.

The chosen technology stack represents a balanced compromise between robustness, flexibility and security. Each of the technologies mentioned was carefully selected with regard to the contribution to achieving the project objectives, i.e. creating modern, scalable and secure social networks. Although the implementation is of a complex technology stack represents a significant challenge in terms of learning curve and integration of individual components, the resulting solution provides a solid foundation for the implementation of an innovative social platform with an emphasis on security, scalability and extensibility.

2.3 Functional and non-functional requirements

Based on the market analysis and identified technological gaps, the following functional and non-functional requirements for the proposed social platform.

2.3.1 Functional requirements

Implementing a robust plugin system is a key feature platform design requirement. The system must allow functionality to be extended via WebAssembly, ensure safe code execution in sandboxed environment and provide standardized APIs for interaction with the platform. The plugin system should support dynamic user interfaces, event registration and processing, and API versioning to ensure long-term compatibility.

Authentication mechanisms must include support for modern standards,

In particular, Passkeys using the FIDO2 protocol eliminate the need for traditional passwords. Implementing QR codes for login is another requirement that will simplify the authentication process across devices. These mechanisms must be implemented with an emphasis on security and user-friendliness.

Content management requires the implementation of a system for sharing and storing multimedia content, including images and text contributions. The system must support various content formats and ensure efficient distribution via CDNs. Transparent algorithms for personalizing displayed content with user control are another key requirement.

Social interactions must be implemented through a user-defined system. circles, which enable the creation of social connections between users. This system must support different types of relationships, including friendships, followings, and group interactions, and provide granular privacy controls for different types of social connections. Users must be able to add other users to specific circles and customize the visibility of content for individual circles.

2.3.2 Non-functional Requirements

Scalability is a key non-functional requirement for the system. The architecture must be able to scale horizontally to support growing numbers of users and data volumes. Implementation on the Kubernetes platform must enable efficient load distribution and performance optimization during peak times.

Security requires the implementation of comprehensive measures, including mTLS for secure communication between microservices, encryption of data at rest and in transit, and protection against common attack types. Authentication mechanisms must meet the latest security standards and provide robust protection for user accounts.

The observability of the system must be ensured through the implementation monitoring and logging tools. These tools must provide a comprehensive overview of system performance and status, enable anomaly detection and effective problem diagnosis.

The user experience must be consistent and optimized across different

devices and platforms. Implementing responsive design and optimizing for different screen sizes and interaction types are key requirements in this area. The system must provide an intuitive interface for managing user circles and personalizing content.

System availability must reach high levels (99.9% and above) through redundancy, automatic recovery after an outage and geographical distribution of services. Implementing a strategy to minimize the impact of planned maintenance work is another requirement in this area.

Chapter 3

System architecture design

This chapter presents a comprehensive proposal for an architecture for a social platform system that implements the identified requirements and addresses the shortcomings of current solutions. The architecture is designed with an emphasis on modularity, scalability, and security, while leveraging modern technological approaches described in the previous chapter. The platform has been named Socigy.

3.1 Overall architecture overview

The proposed architecture of the Socigy social platform represents a complex eco-system of interconnected components, which together form a robust foundation for the operation of a modern social network, as shown in Figure 3.1. The architecture is conceived as a distributed system consisting of three main parts: client applications, cloud infrastructure, and external storage services.
and content distribution.

From a topology perspective, the system is designed as a multi-layer architecture, where individual layers have clearly defined responsibilities and interfaces. This approach
It enables independent development, testing, and deployment of individual components, which increases the agility of the development process and facilitates system maintenance.

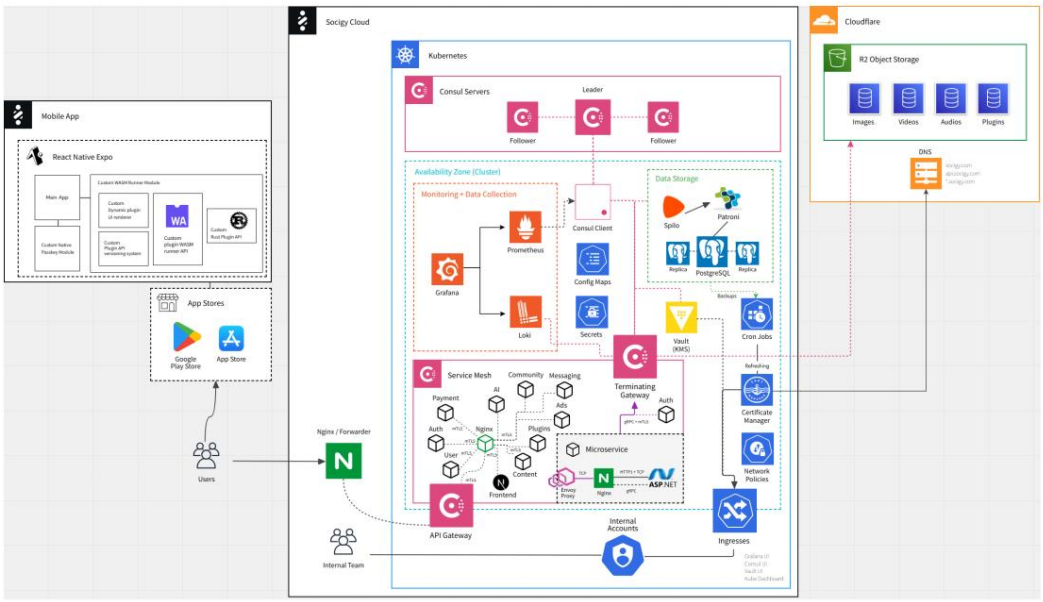

Figure 3.1: Overview of the overall architecture

3.1.1 Client-side overview

The client part of the architecture, shown in Figure 3.2, is represented by two main components: a mobile application developed using the React Native Expo framework and a web application implemented using
Next.js.

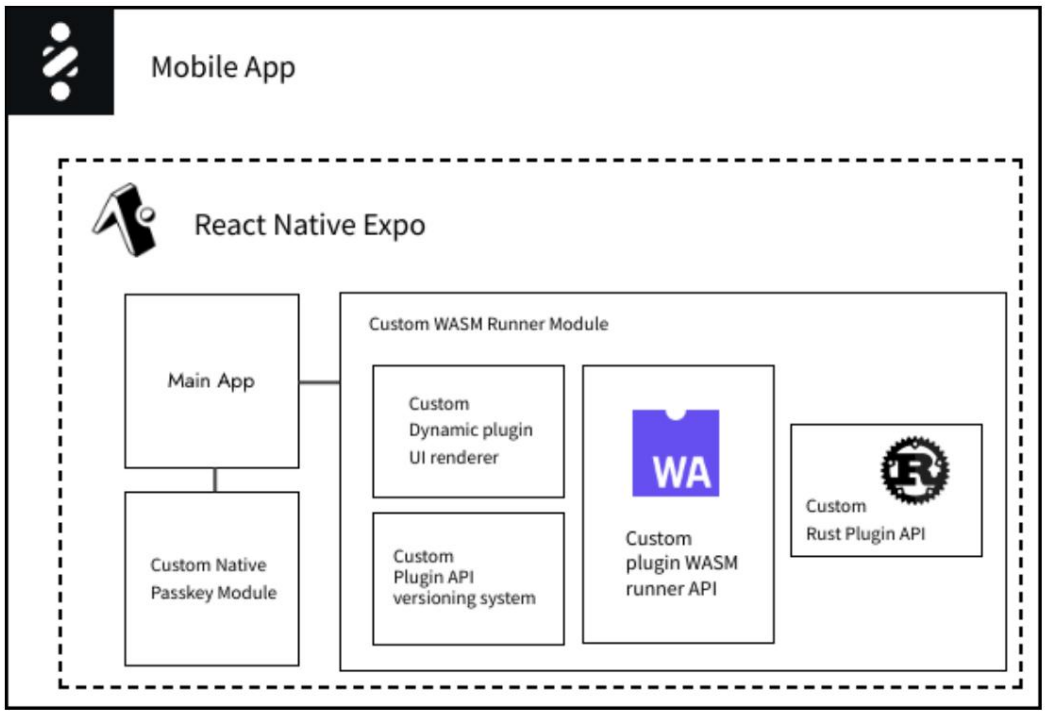

Figure 3.2: Client architecture overview

3.1.2 Overview of the server part

The server part of the architecture, labeled “Socigy Cloud” and shown in Figure 3.3, represents the core of the entire system and is implemented on a platform form of Kubernetes.

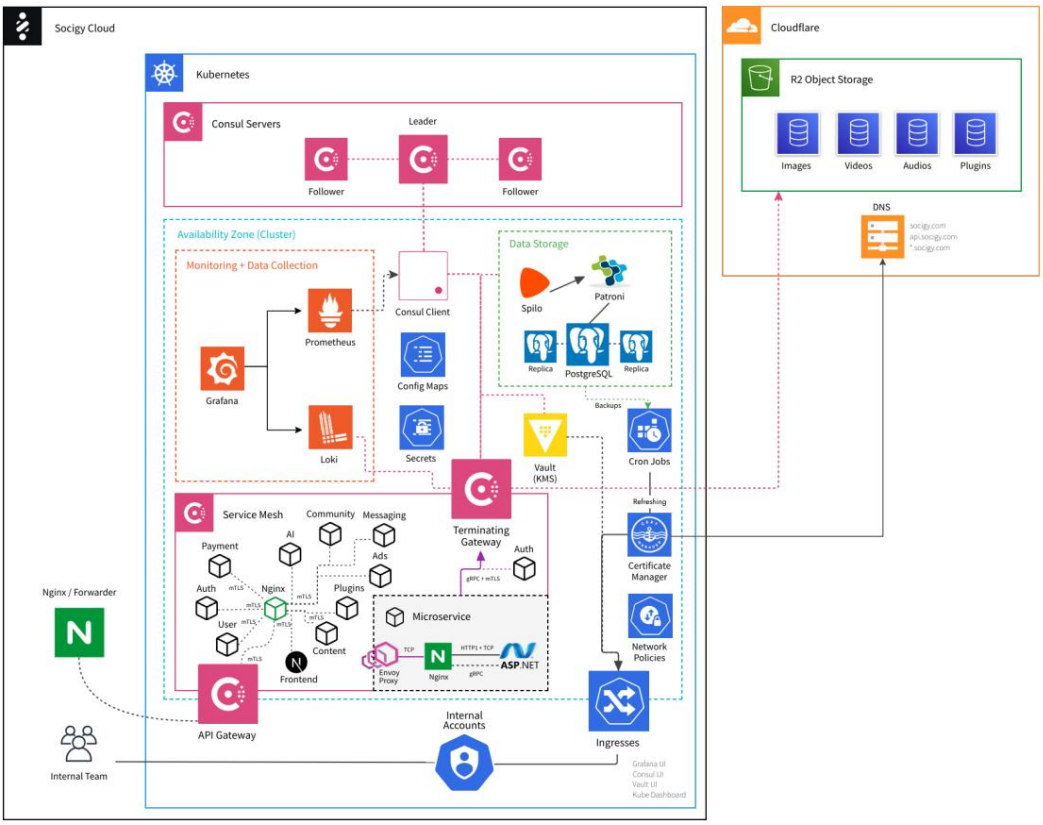

Figure 3.3: Server architecture overview

3.2 Microservice architecture on the platform Kubernetes

Microservices architecture divides a complex system into smaller, independently deployable services communicating via clearly defined interfaces. The chosen approach enables independent development, testing and deployment of individual components, which significantly increases the agility of the development process and facilitates system maintenance. Several key domains represented by separate microservices were identified during the implementation:

- Authentication
- User management

- Content management
- Plugins
- Artificial intelligence
- Messaging
- Communities
- Advertisements
- Payments

Kubernetes provides a comprehensive set of tools for orchestrating microservices. The desired state of the system is defined using declarative manifests specifying the number of replicas, requirements for computing resources, network policies, and other configuration parameters. The chosen method allows versioning of the infrastructure as code and provides a transparent mechanism for management.
change.

Ku- are used to ensure isolation and manage computing resources.
bernetes namespaces providing logical separation of individual parts of the system.
The implementation enables effective management of access permissions and network policy settings.

Microservices configuration is implemented through ConfigMaps and Secrets, which manage configuration parameters and sensitive information. ConfigMaps are used for common configuration parameters, while Secrets store sensitive information such as database credentials or API keys.
The implemented approach separates configuration from code and enables dynamic updates without the need to recompile and deploy new versions of applications.

High availability and fault tolerance are ensured by the distribution strategy
load across multiple nodes within an Availability Zone. Kubernetes automatically distributes microservice instances across available nodes and, if one node fails, moves the affected instances to functional nodes, minimizing the impact on service availability.

The management of the deployment of new versions of microservices is implemented by the strategy role ling update, which gradually replaces running instances with new versions without service outage. The implementation minimizes the risk associated with deploying new version and allows for quick recovery in case of problems.

Readiness and liveness probes are implemented to monitor the health and performance of microservices, which Kubernetes uses to detect dead instances and automatically restart them. Readiness probes determine the readiness
Newly deployed instances receive traffic, while liveness probes detect instances in an inconsistent state requiring a restart.

3.3 Communication layer

Effective communication between system components is a key aspect architecture design. As shown in Figure 3.1, the system uses a specialized communication layer to ensure secure, reliable, and efficient
data exchange.

For communication between microservices in Service Mesh, it is primarily used gRPC protocol that provides high performance, low latency and support streaming data. This protocol, based on HTTP/2, enables efficient serialization of structured data using Protocol Buffers and supports bi-directional streaming, which is essential for implementing real-time functions social platforms. The advantage of gRPC is also the generation of client and server interfaces from definition files, which significantly simplifies the development and API maintenance.

Consul Service Mesh is implemented through an Envoy proxy deployed as sidecars next to each microservice instance. This approach, referred to as the sidecar pattern, allows for transparent implementation of network functions without the need to modify the microservices code. Communication between micro-services pass through the Envoy proxy, which provides routing, load balancing, circuit breaking, and other advanced network functions.

API Gateway, implemented using Consul API Gateway, serves as

entry point for external requests and routes them to the appropriate microservices.
This component supports various protocols including HTTP/1.1, HTTP/2, and gRPC, enabling efficient communication between various types of clients.

For secure communication with external services outside the Service Mesh, a Terminating Gateway is implemented. It provides mTLS termination and translation between secure communication within the Service Mesh and potentially
insecure communication with external systems. This solution allows microservices to communicate securely with external APIs, databases, or legacy systems.

To support real-time communication between clients and the server, it is used SignalR technology, which provides high-level abstractions for Web-Sockets, Server-Sent Events, and Long Polling. This technology enables efficient implementation of notifications, chat, and other real-time features with minimal development effort.

Routing network traffic inside a Kubernetes cluster outside of Service
The mesh is implemented using Kubernetes DNS, which provides a service discovery mechanism. Each service registered in Kubernetes is accessible via a DNS record in the format <service-name>.<namespace>.svc.cluster.local, which allows transparent communication between services without knowledge of the physical
location.

3.4 Security of the deployed ecosystem

Security is a critical aspect of the proposed architecture, especially for a social platform processing sensitive user data. The implemented security model includes several layers of protection.

Communication security is implemented using mutual TLS (mTLS),
when communicating parties verify each other's identity using certificates. Con-sul Service Mesh automatically ensures the issuance, distribution and rotation of certificates for services. This mechanism effectively prevents eavesdropping and man-in-the-middle attacks, which is essential for protecting sensitive

user data.

Access Control Lists (ACLs) are implemented at the Consul level for controlling access to services and their APIs. This mechanism allows for granular definition of permissions and ensures that each service has access only to necessary resources.

Intentions in Service Mesh define allowed communication paths between services. The principle of least privilege is implemented here, where communication between services is only allowed upon explicit definition.

HashiCorp Vault is used to manage secrets and encryption keys (KMS), which provides a centralized repository for sensitive information with access control and auditing mechanisms. This component is key to implementing encryption of data at rest and in transit and secure management of authentication credentials. I give.

Certificate Manager provides automatic management of SSL certificates in a Kubernetes cluster. This tool eliminates the risks associated with manual management certificates and ensures encryption of all external communication using valid certificate.

Network Policies define permitted communication paths at the network level layers, thus providing an additional level of isolation and protection against lateral movement in the event of compromise of any system component.

3.5 Environmental limitations

The proposed architecture of the Socigy social platform, despite its robustness, exhibits several limitations that will need to be addressed in the context of long-term system development and scaling.

The main technological challenge of the current implementation is the missing native support for multi-port configuration in stable versions of Consul Service Mesh. This limitation is particularly evident when implementing complex communication patterns, where microservices need to simultaneously support various communication protocols (gRPC, HTTP/1.1, SignalR) without direct implementation.

HTTPS at the application container level.

To overcome this barrier, it was necessary to integrate the Nginx proxy server as an additional component in each microservice container. Nginx exposes one port visible to the Envoy proxy and internally handles routing requests between different ports designated for specific communication protocols. This solution effectively circumvents the basic limitation of Service Mesh, but at the same time, it increases the complexity of deployment and creates additional computational overhead.

Another significant limitation of the current implementation is the absence of a Mesh Gateway to facilitate communication between multiple Kubernetes clusters. The current architectural configuration limits the deployment of the system to a single cluster (Availability Zone), which significantly limits the possibilities of geographical distribution and presents risk in terms of high availability in the event of a catastrophic failure of the data center

Tuesday

Implementing Consul Service Mesh naturally generates some computational overhead. overhead resulting from the need to run an Envoy proxy in parallel with each microservice instance. This overhead can be significant, especially in environments with limited computing resources or when deploying a large number of microservices with minimal requirements for computing power.

Integration with external systems via Terminating Gateway is potential performance bottleneck when increasing the volume of communication with external services. The current implementation does not allow automatic scaling Terminating Gateway based on dynamic load, which requires continuous monitoring and manual configuration adjustments.

The architecture was primarily optimized for cloud deployment environment. Migration to on-premise infrastructure or a hybrid solution would could require significant adjustments, especially in the areas of automatic scaling, load balancing and service discovery.

3.6 Database solutions

Data persistence in the Socigy social platform is ensured by a robust database PostgreSQL-based solution using Spillo and Patroni for implementation high availability. As shown in Figure 3.3, the database architecture is designed with reliability, performance, and fault tolerance in mind.

PostgreSQL was chosen as the primary database system due to its advanced features, including support for complex data types, indexing using GIN and GiST indexes, which are essential for effective social media search data, and a robust transaction model ensuring data integrity. Schema The database was designed with the specific requirements of a social platform in mind, including entities such as users, circles, relationships between users and messages, which allows for the effective representation of social ties and interactions.

To ensure high availability, a Master-Slave architecture is implemented using streaming replication, which ensures continuous replication. data from the primary node to the replication nodes. The configuration includes one primary node (Master) and two replication nodes (Replica), which provides data redundancy and the possibility of automatic failover in the event of a primary node failure.

A key component for architecture management is Patrons, specialized a tool for orchestrating PostgreSQL clusters. Patroni implements mechanisms for monitoring the state of database nodes, detecting failures, and automatically selecting a new primary node from available replicas. Failover process minimizes downtime in the event of a primary node failure and ensures continuous services.

Spillo, a PostgreSQL operator for Kubernetes, provides an integration layer between the database cluster and the Kubernetes ecosystem. The component provides automated management of database clusters, including provisioning, backups, restoration and scaling.

To ensure data consistency in a distributed environment, transaction mechanisms and conflict resolution strategies were implemented. Databases The schema uses advanced PostgreSQL features such as the uuid-ossp extensions. for generating unique identifiers and pgtrgm for efficient full-text

search.

Regular backups and database maintenance are provided by Kubernetes Cron Jobs. Database configuration is managed using Kubernetes Config Maps, which allows centralized management of configuration parameters and their dynamic update without the need to restart the database. instances.

Chapter 4

Key implementation detail component

4.1 Authentication and security

Implementation of authentication mechanisms and security measures is key part of the design of the social platform Socigy. The system uses modern authentication methods with an emphasis on ensuring the security of user data when while maintaining an intuitive user interface.

4.1.1 Passkey implementation

The primary authentication mechanism is Passkeys based on the standard FIDO2, which provides a high level of security while simplifying the login process. On the web platform, the functionality is implemented through the WebAuthn API, which allows direct integration with biometric device sensors and security keys. (10)

For a mobile application developed in React Native Expo, it was necessary to implement your own native module for the Android platform. Server part The authentication system uses a specialized library developed by FIDO Alliance for validation of authentication data and management of registered devices.

4.1.2 QR Code Sign-in

An alternative authentication mechanism is QR Code sign-in, which is currently in the implementation phase. This approach allows users to log in to a web application by scanning a QR code with a mobile device on which they are already authenticated.

4.1.3 Multi-factor authentication

A significant security feature of the implemented system is mandatory multi-factor authentication (MFA), which is activated automatically when a new user registers. The primary factor is email MFA, where the system requires verification of the email address before completing the registration process.
Furthermore, support for Time-based One-Time Password (TOTP) (11) is implemented as an alternative MFA method.

4.1.4 Device-based authentication

The authentication system is device-oriented, which brings several key benefits. Users can manage their authenticated devices through a dedicated interface in account settings, including the ability to immediately revoke access rights to a specific device.

This approach allows for the implementation of advanced security policies at the device level and the use of authenticated devices for secondary purposes, such as authorizing sensitive operations or implementing QR code login.

4.1.5 Security mechanisms at the API level

Comprehensive security measures are implemented at the API and data transfer levels. Authentication tokens are stored in Secure, HttpOnly cookies, which eliminates the risk of their theft via JavaScript code. Anti-

Forgery tokens that ensure API requests come from legitimate sources.

A significant security feature is the implementation of Cross-Origin Re-source Sharing (CORS), which is implemented individually at the level of each microservice. This decentralized approach was chosen due to the different requirements for data accessibility within individual services. CORS policies are set to allow access only from relevant and verified domains, thereby minimizing the risk of unauthorized access to the API. Individual implementations of CORS also allow for flexible access to future functionality extensions, such as the ability to embed posts on external websites, where specific endpoints may have different

restrictive CORS settings.

Communication between the client and the server is secured via TLS/SSL using modern cryptographic algorithms. To increase the security of the web application, Content Security Policy (CSP) and other security headers are implemented, which provide protection against various types of attacks, including Cross-Site Scripting (XSS), clickjacking and data injection. CSP defines allowed resources for loading scripts, styles, images and other types of content, thereby minimizing the risk of malicious code execution.

4.2 Microservice ecosystem

The implementation of a microservice ecosystem is the core of the proposed social platform. Microservices are developed using ASP.NET AOT 8, which provides performance optimizations through Ahead-of-Time compilation.
The architecture of microservices is based on the principle of Dependency Injection, which enables flexible management of dependencies and facilitates testing of individual com-speaker.

The microservices communication infrastructure is implemented through a combination of several protocols - gRPC for high-performance communication between services, SignalR for real-time communication, and standard HTTP/1.1 for

REST API. The core of the communication layer is the Kestrel server, which provides high performance and low latency. To address the limitations of Consul Service Mesh in the area of multi-port configuration, a custom Dockerfile was implemented with integrated Nginx, which acts as a reverse proxy. HTTP/1.1 and SignalR communication is mapped to port 5000, HTTP/2 gRPC to port 5001, while Nginx listens on port 8080, which is then used by the Envoy proxy within the Service Mesh.

4.2.1 Databases ORM mapper

A significant aspect of implementing a microservice ecosystem is the development of a custom PostgreSQL ORM mapper, which was necessary for compatibility with AOT compilation. Entity Framework, the standard ORM framework for .NET applications, does not provide full support for AOT compilation, which led to the need to implement a custom solution for object-relational mapping.

The developed mapper implements basic CRUD operations (Create, Read, Update, Delete) and provides a type-safe interface for working with database entities. The mapper architecture is based on generic classes that allow defining mappings between database tables and domain objects. The mapper uses Npgsql as a low-level provider for communication with the PostgreSQL database and implements its own mechanisms for managing connections, transactions, and query result mapping.

A key feature of the mapper is its support for compiling queries at build time, which is consistent with the principles of AOT compilation. This feature eliminates the overhead associated with dynamically compiling queries at runtime and contributes to the overall performance of the application. The mapper also implements optimizations for batch operations and supports asynchronous database access, which is critical for the scalability of microservices.

From a security perspective, the mapper implements parameterized queries, which effectively prevents SQL injection attacks. The mapper also provides mechanisms for logging and monitoring database operations, which facilitates diagnostics and performance optimization.

4.2.2 Authentication middleware

A specialized authentication middleware has been implemented within the microservice ecosystem, which ensures user identity verification and provides an authentication context for processing requests. The middleware works on the principle of an interceptor, which intercepts incoming requests and performs authentication before passing them on to the target handlers.

On the Auth microservice, the middleware works locally by calling a custom-created internal interface ITokenService, which provides methods for validating and managing authentication tokens. This solution minimizes network communication and optimizes the performance of the authentication process.

On other microservices, authentication is implemented via gRPC communication directly with the Auth microservice. When a request arrives, the middleware extracts the authentication token, sends it via gRPC calls to the Auth microservice for validation, and then applies the retrieved user data to the HttpContext. This approach ensures centralized authentication management and consistent application of security policies across the entire ecosystem.

4.2.3 Middleware for validation of internal requirements

To ensure communication between microservices, custom middleware was implemented, which ensures authentication and authorization of internal requests.
The original draft considered the use of mTLS certificates provided by Envoy Proxy, which would enable the identity of the calling service to be verified based on the certificate.

Due to the need to implement multi-port configurations and support various
However, due to the communication protocols, it was necessary to switch Envoy Proxy to TCP mode, which does not provide the ability to transfer certificates to requestors. For this reason, the middleware design was modified and an alternative approach based on OAuth clients was implemented.

Newer solutions for creating an OAuth client in the Auth database for each microservice. During internal communication, the middleware verifies the validity of the client and its secret key via a gRPC request sent to

Microservice Auth. This approach ensures secure communication between microservices even in environments with limited TCP data using Envoy Proxy mode.

4.2.4 Middleware for user data extraction

To improve user experience and increase security, it was implemented middleware that analyzes information about the client device and browser. Middleware extracts the device fingerprint and UserAgent string from the request based on this information provides contextual data to Http-Context.

The information obtained is used for various purposes, including the detection of potentially suspicious logins, optimizing the user interface for specific devices and the collection of analytical data about used devices and browsers.

4.2.5 Microservice authentication

The authentication microservice provides centralized management of authentication, authorization, and auditing across the entire system. The core of the mechanism is the implementation of of the FIDO2 standard, providing secure user authentication without traditional passwords. For managing logins and maintaining authentication status, they are used JWT (JSON Web Tokens).

Data persistence is ensured through a dedicated auth_db database, logically separated from other microservice databases. This solution supports the principle of separation of responsibilities and enables independent management of authentication of the data.

Implementing Passkeys was a significant challenge, requiring careful integration with the FIDO2 standard and optimization of the user experience across different devices and browsers. Although OAuth functionality is not currently available fully implemented, the database schema already includes preparation for future integration of this technology.

For a detailed look at the database structure and specific implementation The complete database schema is given in detail in Chapter 9. Authentication

microservices.

4.2.6 User microservice

The user microservice manages user profiles and relationships between them.
users. It includes storing and updating basic information about users,
such as first name, last name, email, and date of birth. The microservice implements a sophisticated system for managing various types of relationships between users, including friends, followers, or blocks.

Data is stored in a dedicated user_db database, whose schema is optimized for efficient querying of complex relationships between users.
The key functionality is the management of user circles (circles), enabling
organizing contacts into logical groups with different permission levels.

The system supports the import and management of personal contacts, including mechanisms for matching imported contacts with existing platform users. Security and privacy are ensured by a granular authorization system,
which allows users to precisely control access to their personal information. It is complemented by a robust system for detecting and preventing unwanted
behavior, including blocking users and reporting inappropriate content.

4.2.7 Content microservices

The content microservice ensures comprehensive content management and user interaction on the platform. It implements a robust system for uploading, storing and distributing multimedia content, including images, videos and text posts.
Although the streaming functionality is not currently fully implemented
implemented, the architecture is ready for future integration of this function.

A notable aspect is the content categorization and user management system.
interests, enabling efficient organization and search for content. Microservice
includes a basic implementation of content AI profiles that adjust recommended categories and interests, providing a foundation for future implementation of more advanced personalization algorithms.

Data is stored in a dedicated content_db database. Management system
User interaction with content includes features for rating, commenting
and sharing the contribution, including the performance when processing a large amount of current interactions.

The architecture is designed with future expansion in mind to include advanced content moderation methods using artificial intelligence models to automatically detect and filter potentially harmful content. This planned integration will significantly increase the efficiency of the moderation process and strengthen security.
user environment.

4.2.8 Plugins and microservices

The plugin microservice provides management of plugins, their lifecycle, and platform extensibility. The core is a system for managing the entire process from build through publish to distribute plugins. The architecture implements sophisticated versioning that allows developers to publish updates while maintaining compatibility with existing installations. The system supports
various publication states including beta versions, preparation, review and final publication forged condition.

A significant component is the localization system, which allows for customization of plugins for different languages and regions. The implementation supports storing localized texts in JSON format, which provides flexibility in defining complex localization structures with validation of regional codes.

Data is stored in a dedicated plugin_db with a complex table structure for plugin metadata, versions, installations, localizations, assets, and user
data. The database schema uses sophisticated indexing to optimize
execution of queries and ensuring referential integrity.

The installation management system tracks plugins at the user and device level, which allows installation on various devices with status monitoring mechanisms,
usage statistics and update management. Security aspects include validation
and verification of plugins, including integrity checks and detection of potentially harmful code with different levels of verification status.

For storing and distributing binaries and assets, integration with Cloudflare R2 Object Storage is implemented, ensuring efficient management of WebAssembly modules, icons, and other related files. There is also a rating and review system that allows users to provide feedback to developers with mechanisms for preventing abuse.

4.3 Mobile applications

The mobile application of the Socigy social platform is implemented using React Native Expo, which enables the development of cross-platform applications with native performance. The application includes key social network functions, including post display, profile management, user circle management, relationship management, and plugin commerce.

Advanced techniques have been implemented to optimize performance and user experience. One of them is the use of the FlashList component, which is an optimized version of the standard FlatList component. Flash-List offers higher performance and lower memory requirements when rendering long lists, which is crucial for smooth browsing of contributions and other datasets. Although FlashList is not used throughout the application, its deployment in critical parts significantly contributes to the overall performance.

Another significant optimization is the implementation of image caching using ExpoImage. This component uses the powerful native libraries SDWebI-mage for iOS and Glide for Android, which ensure efficient loading and caching of images. This significantly reduces the number of network requests and speeds up the display of previously loaded images.

An approach based on React Con-texts was chosen for managing the application state. This solution allows for efficient data sharing between components without the need to explicitly pass props through the entire component hierarchy.
The implementation includes several key contexts, such as AuthContext for managing authentication, ThemeContext for controlling the appearance of the application, and other specialized contexts for managing user data and settings.

The plugin installation process in the mobile application is designed with simplicity and security in mind. The user can browse the available plugins in the integrated store, select the desired version and initiate the installation. The “installation” itself consists of registering the user and device on the server as a user of the given plugin, without the need to download and run the code directly in the application.
This solution allows for central management of plugins and their authorization.

4.4 Web application

The web version of Socigy is developed using the Next.js framework, which provides powerful tools for server-side rendering and performance optimization. The web application functionality includes the same key elements as the mobile version, including post viewing, profile management, user circle and relationship management, and a plugin store.

The Zustand state manager was chosen for status management in the web application. Unlike React Contexts used in mobile applications, Zustand offers a simpler API and better performance when working with complex states. Zustand allows the creation of isolated state repositories, which makes it easier to test and maintain the code. This choice reflects the specific needs of a web application and a different approach to front-end architecture compared to the mobile version.

The web application also includes a dedicated section for plugin developers. This section provides tools for plugin management, including metadata editing, version management, log tracking, localization and asset management. Developers also have access to analytics data about their plugins, including user ratings and reviews. There is also an interface for managing the plugin database with an overview of usage limits.

To ensure responsive design and a consistent look across different
The best way to do this is to use the TailwindCSS framework. This utility-first CSS framework enables rapid development of responsive interfaces and easy customization. design.

Although the current implementation of plugins does not include their direct integration,

into the main application, this is planned for a future iteration. This decision was motivated by the need to thoroughly test the concept and optimize the architecture before full integration. An older version of the implementation, which included direct plugin integration, is available in the appendices of the work for comparison and analysis. (9, /client/native/app-old-iteration)

Both versions of the application, mobile and web, share common logic for key functions such as viewing posts, managing profiles and user relationships.
This approach ensures a consistent user experience across platforms and simplifies maintenance and the development of new features.

Chapter 5

An ecosystem of user plugins

This chapter describes the design and implementation of a user-friendly plugin ecosystem, which is one of the key innovative elements of the proposed social platform. The system enables extensibility of the platform through third-party modules, while ensuring security, stability, and consistency.
user experience.

5.1 Application layer

The application layer of the plugin system provides an interface between the platform core and user plugins. This layer implements mechanisms for managing the plugin lifecycle, communicating between plugins and the platform, and ensuring the safe execution of third-party code.

5.1.1 API versioning

The API versioning system is a key aspect of the plugin ecosystem, ensuring long-term compatibility and maintainability. The implemented approach allows plugins with different versions to run simultaneously.

system API, which eliminates the need for regular plugin updates when changes to the underlying platform.

Versioning is implemented through semantic versioning, where plugins specify the required API version in their in-app developer interface

The system automatically detects compatible API versions and provides plugins. gin corresponding interface. This mechanism is implemented in the class PluginCacheManager, which analyzes the requested API version and selects the most suitable implementation:

```
1 private fun resolveBestVersion(requestedVersion: String,
      availableVersions: List<String>): String? {
2         val baseVersion = if (requestedVersion.startsWith("^")) {
3                 requestedVersion.substring(1)
4         } else {
5                 requestedVersion
6         }
7
8         val requestedSemver = Semver(baseVersion,
            Semver.SemverType.NPM)
9         val rangeStart = requestedSemver
10        val rangeEnd = Semver("${requestedSemver.major + 1}.0.0",
            Semver.SemverType.NPM)
11
12        return availableVersions
13                .map { Semver(it, Semver.SemverType.NPM) }
14                .filter { it.compareTo(rangeStart) >= 0 &&
                   it.compareTo(rangeEnd) < 0 }
15                .maxByOrNull { it }
16                ?.toString()
17 }
```

This approach ensures backward compatibility and allows for gradual evolution. API without disrupting the functionality of existing plugins.

5.1.2 Registration and release of events

The event system allows plugins to respond to user actions and changes in the application. The implementation includes mechanisms for registering event listeners, distributing events, and processing them within plugins.

To facilitate work with events, a specialized part of the developed Rust framework is provided, which provides high-level abstractions for registration and reporting. events:

```
thread_local! {
    pub(crate) static REGISTERED_EVENTS: Rc<RefCell<HashMap<String,
    ÿ   Box<dyn FnMut(&UIEvent)>>>> =
    ÿ   Rc::new(RefCell::new(HashMap::new()));
}

#[wasm_bindgen]
pub fn invoke_ui_event(id: String, event: String) {
    let json_res = match
     ÿ serde_json::from_str::<UIEvent>(event.as_str()) {
        Ok(res) => res,
        Err(e) => {
            logging::adv_error(
                format!("Failed to deserialize UIEvent -> {}",
                 ÿ e).as_str(),
                None,
                false,
            );
            return;
        }
    };

    REGISTERED_EVENTS.with(move |events| {
        if let Some(listener) = events.borrow_mut().get_mut(&id) {
            listener(&json_res);
        }
```

```
        });
}
```

This code allows event listeners to be registered and triggered when the user interacts with the plugin components. The events are serialized into JSON format and passed between native code and WebAssembly modules hem.

On the client side, a system for distributing events between plugins and application address:

```
    const eventNamesArray: string[] = [
        "onPointerEnter",
        "onPointerEnterCapture",
      // Other event types
    ];
    // ... renderVDOM()
    const { type, events, children, key } = element;
    if (events) {
        Object.keys(events).forEach((event) => {
            if (eventNamesArray.includes(event)) {
                const eventIds = events[event];

                // Internal event callback
                const callback = (e: any) => {
                    const eventData = JSON.stringify({
                        type: event,
                        ...e.nativeEvent,
                    });
                    eventIds.forEach((eventId) => {
                        uiRegistry.invokeUiEvent(pluginId, eventId, eventData);
                    });
                };

                // Props passed to the rendered component
```

```
25                 if (props) {
26                     props[event] = callback;
27                 } else {
28                     props = { [event]: callback };
29                 }
30             }
31         });
32     }
```

This approach allows plugins to respond to user interface events such as clicks, resizes, or other interactions, providing rich possibilities for implementing interactive components.

5.1.3 Dynamic user interfaces

Dynamic user interfaces are a key feature of the plugin system, allowing plugins to define and render their own components within the application. The implementation is based on the concept of virtual DOM (VDOM), which allows for efficient user interface updates.

Developed Rust framework provides JSX-like syntax for defining user interface:

```
1 #[ui_component]
2 struct Page {
3         render_string: bool,
4         content: Option<PropStr>,
5         image_url: Option<PropStr>,
6 }
7
8 impl UIComponent for Page {
9         fn render(&mut self) -> Option<UIElement> {
10                ui! {
11                    <View class_name="flex-1 flex b" on_layout={|e| {
12                        info!("OnLayout event was fired on Page. Event:
                          ÿ {:?}", e);
```

```
13                      }}>
14                          <Text class_name="text-2xl font-inter-bold
                             ÿ text-foreground">Your watchlist</Text>
15                          <Counter render_string={self.render_string}
                             ÿ     image_url={self.image_url.clone().unwrap()}
                             ÿ     content={self.content.clone().unwrap()} />
16                      </View>
17                  }
18              }
19 }
```

This code defines a component with properties and a method for rendering, which uses a custom-developed macro ui! for defining the structure of the user interface. The framework automatically converts this definition into a virtual DOM representation, which is then serialized and passed to the native application.

On the client side, a component rendering system is implemented defined plugins:

```
1 export default function Dynamic({
2     id,
3     defaultElement,
4     props,
5     uiRegistry,
6 }: DynamicProps) {
7     const [vdom, dispatchChange] = useReducer(dynamicVdomReducer,
       ÿ undefined);
8
9     useEffect(() => {
10        const registeredPlugin = uiRegistry.getComponentPlugin(id);
11        if (!registeredPlugin) {
12            return;
13        }
14
15        const changeSubscription = SocigyWasm.addListener(
```

```
16              "onComponentChange",
17              (data) => {
18                  if (!data.changes) return;
19                  const changes: VDOMChange[] = JSON.parse(data.changes);
20                  dispatchChange(changes);
21              }
22          );
23
24          if (
25              !SocigyWasm.renderComponent(registeredPlugin, id,
                ÿ JSON.stringify(props))
26          ) {
27              console.error(`Failed to render dynamic component ${id}`);
28          }
29          return () => {
30              changeSubscription.remove();
31          };
32      }, [id]);
33
34      if (!vdom) {
35          return defaultElement;
36      }
37      return renderVDOM(vdom);
38  }
```

This code renders the component defined by the plugin and updates it when changes occur. The system uses a virtual DOM to optimize performance.
and minimizing the number of updates to the real DOM.

5.2 Plugin layer

The plugin layer ensures the safe execution of third-party code and provides interface for interaction with the application core. This layer is implemented with using WebAssembly, which enables efficient and safe code execution

in a sandboxed environment.

5.2.1 Sandoxing on native devices

Implementing sandboxing on native devices is a significant technical challenge, especially on mobile platforms. To solve this problem A specialized native module for Android has been developed that allows for secure running WebAssembly code in an isolated environment.

The SocigyWasm module implements the interface between native code and WebAssembly modules:

```
1 class SocigyWasmModule : Module() {
2     private final var PluginCacheManager: PluginCacheManager? = null;
3
4     override fun definition() = ModuleDefinition {
5         Name("SocigyWasm")
6
7         OnCreate() {
8             SocigyWasmExceptions.setSendEvent(::sendEventWrapper);
9             try {
10                PluginCacheManager = PluginCacheManager(getContext(),
                      ÿ ::sendEventWrapper);
11            } catch (e: Exception) {
12                sendEvent("onFatal", bundleOf(
13                    "message" to "FATAL_ERR -          " + e.toString(),
14                    "uiDelay" to 0
15                ));
16            }
17        }
18
19        // API definition for communication with React Native
20        // ...
21    }
22 }
```

This module provides an interface for loading, initializing, and running WebAssembly modules within a React Native application. The implementation uses WebView as the runtime for WebAssembly, ensuring compatibility with most
Android device.

A key aspect of the implementation is the isolation of plugin code, which is provided by the WebAssembly sandbox. WebAssembly provides a security model based on linear memory and restricted access to the host environment, which effectively prevents unauthorized access to system resources.

Access to platform functionality is controlled through explicitly exported functions that are available to plugins:

```
1 WebView!!.addJavascriptInterface(ISocigyLogging(sendEvent),
    ÿ "SocigyLogging");
2 WebView!!.addJavascriptInterface(ISocigyInternal(sendEvent),
    ÿ "SocigyInternal");
3 WebView!!.addJavascriptInterface(ISocigyUtils(sendEvent),
    ÿ "SocigyUtils"); 4
WebView!!.addJavascriptInterface(ISocigyPermissions(sendEvent),
    ÿ "SocigyPermissions");
5 WebView!!.addJavascriptInterface(ISocigyUI(sendEvent), "SocigyUI");
```

This approach enables precise control over which functions
what plugins have access to, and implementing a permissions system that allows users to control what actions plugins can perform.

5.3 UI

The plugin system user interface provides mechanisms for defining, registering, and rendering components defined by plugins. This part of the system ensures the integration of plugins into the application user interface and provides a consistent user experience.

5.3.1 Component definitions and registration

The component definition and registration system allows plugins to create own user interface that is integrated into the application. Implementation It includes mechanisms for defining components, registering them within the application, and managing their lifecycle.

The definition of components within plugins is implemented through the developed Rust framework, which provides high-level abstractions for creating a user interface:

```
#[ui_component]
struct Counter {
    render_string: bool,
    content: PropStr,
    image_url: PropStr,
}

impl UIComponent for Counter {
    fn render(&mut self) -> Option<UIElement> {
        ui! {
            <View class_name="flex-1 flex-row items-center
             ÿ justify-center">
                <Text>{self.content.to_string()}</Text>
                <Image source={{ uri: self.image_url.to_string() }}
                 ÿ style={{ width: 100, height: 100 }} />
            </View>
        }
    }
}
```

This code defines a component with properties and a method for rendering, which uses the ui! macro to define the structure of the user interface. The component is then registered within the application:

```
year component_id = "2368bb7a-1021-49d1-85f3-7049fb15abed";
register_component::<MyComponent>(&component_id);
```

On the client side, a system for registering and managing com-
speaker:

```
1 private internal_registerComponent(data: ComponentBasicEventData) {
2     this.components.set(data.componentId, data.pluginId);
3     let registered = this.plugins.get(data.pluginId);
4     if (!registered) this.plugins.set(data.pluginId,
        ÿ [data.componentId]);
5     else {
6         registered.push(data.componentId);
7         this.plugins.set(data.pluginId, registered);
8     }
9 }
```

This code ensures the registration of the component within the application and its association with the respective plugin. The registered components are then available for rendered within the application.

5.3.2 Rendering Components

The component rendering system ensures efficient rendering of user interface defined by plugins. The implementation is based on the concept of virtual DOM, which allows for efficient updating of the user interface.

Component rendering is implemented using a specialized renderers that convert the virtual DOM representation into native components:

```
1 function renderVDOM(
2     element: UIElement | undefined,
3     uiRegistry: UIRegistry,
4     pluginId: string
5 ): React.JSX.Element | undefined {
6     if (!element) return undefined;
7     else if (typeof element === "string") return
        ÿ <Text>{element}</Text>;
8
```

```
    const { type, events, children, key } = element;
    let props = { ...element.props };

    const Component = getElementByType(type);
    const renderedChildren = children?.map((x) =>
        renderVDOM(x, uiRegistry, pluginId)
    );

    if (type == "Fragment") {
        return <Component children={renderedChildren} />;
    }

    if (events) {
        // Events...
    }

    if (type == "External") {
        return (
            <Component
                key={element.key}
                {...props}
                children={renderedChildren}
                uiRegistry={uiRegistry}
            />
        );
    }

    return (
        <Component key={key} {...props} children={renderedChildren} />
    );
}
```

This code converts the virtual DOM representation into native React components, which are then rendered within the application. The system supports

hierarchical components, properties and events, which enables the creation of a complex user interface.

Optimization

To ensure optimal performance when rendering components defined
plugins have been implemented using various optimization techniques. The key optimization is the use of virtual DOM, which allows minimizing the number of
updating the real DOM.

```
function dynamicVdomReducer(
    state: UIElement | undefined,
    actions: VDOMChange[]
): UIElement | undefined {
    return produce(state, (draft) => {
        if (typeof draft === "string") return;
        else if (!draft) return actions[0].element;

        actions.forEach((action) => {
            switch (action.type) {
                case "addElement": {
                    // ...
                }
                case "replaceElement": {
                    // ...
                }
                // Other types of changes
            }
        });

        return draft;
    });
}
```

This code implements a reducer for applying changes to the virtual DOM. System supports various types of changes, such as adding, removing, or updating elements, which allows for efficient updating of the user interface without the need to redraw the entire tree.

Another optimization is to use WebAssembly for efficient data processing and logic on the plugin side. WebAssembly provides near-native performance code, which allows for the effective implementation of complex algorithms and data processing within plugins.

Chapter 6

Monitoring and benchmarking

6.1 Monitoring and logging

Effective monitoring and logging is a key aspect of managing the complex microservice architecture of the Socigy social platform. To ensure robust system oversight, a specialized monitoring infrastructure based on modern open-source tools has been implemented.

The core of the monitoring system is a stack composed of Grafana and Promethea and Loki, which runs in a dedicated namespace within a Kubernetes cluster, but outside the Consul Service Mesh. This architectural decision ensures independence of monitoring components on the monitored environment, which eliminates potential cascading failures in the case of a Service Mesh problem.

Prometheus serves as the primary tool for collecting and storing metrics data of various system components. In the implemented Prometheus solution dynamically discovers monitored targets via the Kubernetes API, which allows new microservice instances to be automatically added to the monitoring system without the need for manual configuration. For each microservice standardized metrics are exposed including CPU usage, memory, request latency and the number of processed transactions.

Loki complements the monitoring infrastructure as a scalable system for log aggregation and analysis. Unlike traditional log management solutions, Loki

does not index the content of the logs, but only the metadata, which significantly reduces storage and computing resource requirements. The implementation includes standardized log formats across all microservices, which facilitates their analysis and correlation. The logs are structured in JSON format and contain contextual information such as request ID, user ID, and other relevant metadata, which allows for efficient tracing of requests across a distributed system.

Grafana serves as a central visualization platform that integrates data from Prometheus and Loki into clear dashboards. This approach enables a comprehensive view of the performance and health of the entire microservice ecosystem.

Specialized ex-porters are used to monitor a Kubernetes cluster, providing a detailed view of the status of individual nodes, pods, and other Kubernetes objects. These metrics are integrated into a central monitoring system, allowing for the correlation of application-level issues with potential infrastructure-level issues.

Consul Service Mesh provides an additional layer of monitoring data through integrated metrics about communication between services. These metrics include latency, throughput, and error rates of individual calls between microservices, which allows the identification of bottlenecks and problematic services within a distributed system.

6.2 Benchmarking

Benchmarking is a critical part of the development process of the Socigy social platform, especially in the context of a plugin system that must efficiently handle dynamic user interfaces and interactions. Extensive performance tests were conducted as part of the development process to optimize key system components.

A significant aspect of benchmarking was the evaluation of various implementation access to the plugin system. The original AssemblyScript-based implementation achieved a performance of approximately 16,000 frames per second (fps) in stress tests. This metric represents the speed at which the system is able to

generate updates to the virtual DOM and prepare JSON representations for rendering the user interface, without counting the time needed for actual rendering to the screen.

The subsequent reimplementation of the plugin system in Rust resulted in dramatic performance improvement, reaching up to 200,000 fps under the same testing conditions. This more than twelve-fold increase in performance can be attributed to several factors:

- More efficient memory management in Rust compared to AssemblyScript
- Optimized implementation of virtual DOM with minimal overhead costs
- More efficient serialization and deserialization of JSON structures
- Improved algorithm for detecting changes and minimizing updates

Benchmarking also included measuring latency when processing user requests. interactions such as clicks, size changes, or other events. The Rust implementation achieved an average latency of under 5 ms, which is well below the 100 ms threshold that is considered the limit for perceiving instant user response.
interface.

In addition to the performance tests of the plugin system, load tests of the microservice architecture as a whole were also performed. These tests simulated high loads with thousands of concurrent users and measured the system's ability to scale and maintain consistent performance. The results showed that the implemented architecture is able to scale horizontally effectively and maintain stable latency even under high loads.

Benchmarking revealed several potential bottlenecks in the system, mainly in the area of database operations and inter-service communication. Based on these findings, a number of optimizations were implemented. A key improvement was the introduction of advanced connection pooling for database operations, which significantly reduced latency and increased system throughput. Another planned optimization is the implementation of distributed caching of frequently accessed data using

Redis. This solution has the potential to significantly reduce the load on the primary database and speed up system response. Although Redis caching has not yet been fully implemented due to time constraints, preliminary analyses suggest that it could deliver up to a 30% improvement in response speed for the most frequently requested queries. data.

The benchmarking results confirm that the chosen technology stack and im-breeding approaches provide a solid foundation for a powerful and scalable social platform. In particular, the transition to Rust to implement critical components of the plugin system proved to be a key decision for achieving excellent performance when processing a dynamic user interface.

Chapter 7

Discussion and evaluation of results

In this chapter, I will focus on evaluating the achieved results, comparing the implemented solution with existing alternatives, and identifying limitations of the current implementation along with suggestions for future development.

7.1 Evaluation of achieved results

The implementation of the Socigy social platform represents a comprehensive technological solution that successfully addresses a number of identified shortcomings of existing social networks. When evaluating the achieved results, several aspects need to be taken into account.

From the perspective of the frontend implementation, a functional user interface was achieved. interface that demonstrates the basic concept of the platform. Using React Native Expo for the mobile application and Next.js for the web application made it possible to create a cross-platform solution with a consistent user experience. Although the current UI/UX implementation does not achieve all of the originally intended goals, such as supporting docking tabs and full multitasking on the web platform, it provides a solid foundation for further development.

Microservices architecture implemented on the Kubernetes platform has proven

its effectiveness in solving scalability and modularity problems. The implementation includes four key microservices (Auth, User, Content, Plugin), which together they provide a robust foundation for social platform functionality. Currently implementation is limited to one Kubernetes cluster, which limits the possibilities geographical distribution, however the architecture is designed with future expansion into multi-cluster environments in mind.

A significant achievement is the implementation of a plugin-based system on WebAssembly, which enables the safe execution of third-party code in sandboxed environment. Although the integration of plugins into the main application was not completed due to time constraints, a complete infrastructure was created for development, distribution and management of plugins, including its own Rust framework for plugin development with support for JSX-like syntax.

From a security perspective, significant progress has been made in implementing modern authentication mechanisms, including support for Passkeys. (FIDO2) and multi-factor authentication. Implementation of mTLS within Service Mesh provides secure communication between microservices. The planned integration of HashiCorp Vault for secret and encryption key management was not fully implemented due to time constraints, although the infrastructure for its deployment was ready.

Monitoring and observability of the system were ensured by implementing a stack Grafana, Loki and Prometheus, which provide comprehensive tools for performance monitoring and anomaly detection. This infrastructure is key to ensuring reliable operation and proactive identification of potential problems.

7.2 Comparison with existing solutions

When comparing the implemented solution with existing social platforms Several significant differences and innovations are evident.

From a technological point of view, Socigy differs from existing platforms by implementing modern authentication mechanisms. While platforms such as However, Instagram and Facebook still rely primarily on traditional passwords supplemented by

two-factor authentication, Socigy implements Passkeys based support
based on the FIDO2 standard, which provides a higher level of security while
simplification of the login process.

A significant innovation compared to existing solutions is the implementation of devices
oriented authentication, which gives users greater control over access
to their accounts. This access allows users to manage their authenticated devices and, if
necessary, immediately revoke access rights
specific device without having to change the login details for all others
device.

In terms of user interface, the current Sociga implementation lags behind
behind the highly optimized interfaces of dominant platforms, which have invested significant
resources in UI/UX development and testing. However, the concept
support for docking panels and multitasking on the web platform is
a potential advantage over existing solutions that typically do not support these features.

7.3 Implementation limitations and suggestions for building teaching development

The current implementation of the Socigy social platform has several limitations, which
they represent opportunities for future development.

One of the main limitations is the lack of full integration of plugins into the main
application. Although a complete infrastructure for the development and distribution of plugins
was created, their integration into the user interface was not timely.
reasons completed. Future development should focus on completing this integration, which
would allow users to fully exploit the potential of the plugin
system.

From a security perspective, incomplete implementation represents a significant limitation.
HashiCorp Vault for secret and encryption key management. However
the infrastructure for deploying Vault was prepared, its full integration with
other components of the system has not been completed. Future development would

should focus on completing this integration, which would provide a robust solution for management of sensitive information.

To optimize performance, it would be appropriate to implement distributed caching using Redis, which could significantly reduce latency and increase system throughput. This optimization would be especially beneficial when scaling.
platform for the largest number of users.

Another direction for future development could be the implementation of advanced content personalization algorithms that would provide users with more relevant content while maintaining transparency and control. This could
include the implementation of a model-based content moderation system artificial intelligence, which would increase the security of the platform and the quality of the user experience environment.

From the perspective of the extensibility of the plugin system, it would be appropriate to implement more advanced mechanisms for authorization management and performance monitoring. plugins. This could include implementing a system for dynamic
allocating resources to plugins based on their usage and implementing advanced metrics for monitoring plugin performance and stability.

Implementing these improvements would significantly increase competitiveness Socigy platform and would provide users with an innovative and secure environment
for social interactions in digital space.

Chapter 8

Conclusion

8.1 Summary of key findings

This work presented the design and implementation of a modern social platform So-cigy, which addresses the identified shortcomings of existing solutions through an innovative approach to architecture, security, and extensibility. The achieved results demonstrate the potential of the selected technological solutions for creating a robust and user-oriented social network.

The implementation of microservice architecture on the Kubernetes platform has proven as a suitable choice for ensuring scalability and modularity of the system. Kubernetes provided a robust foundation for orchestrating containerized applications, which enabled efficient management, scaling, and deployment of individual microservices.
Using Service Mesh with HashiCorp Consul has brought significant benefits in the area of securing communication between services and centralized management of network policies.

On the frontend, significant progress was made by implementing a cross-platform solution using React Native Expo for mobile applications and Next.js for the web application. This approach ensured a consistent user experience across devices and platforms, while enabling efficient code sharing between implementations.

The most significant contribution of the work is the implementation of a plugin system

based on WebAssembly, which allows third-party code to be safely run in a sandboxed environment. The developed Rust framework for creating plugins with support for JSX-like syntax significantly simplifies extension development and provides developers with intuitive tools for creating user interfaces.
Benchmarking has demonstrated the excellent performance of the Rust implementation, reaching up to 200,000 fps when processing the virtual DOM, which represents more than a twelve-fold improvement over the original AssemblyScript implementation.

From a security perspective, significant progress has been made in implementing modern authentication mechanisms, including support for Passkeys (FIDO2) and multi-factor authentication. This approach eliminates the risks associated with traditional passwords and provides users with a higher level of security while simplifying the login process.

8.2 Recommendations for future research and practice

Based on the experience gained during the development of the Socigy platform, several recommendations can be formulated for future research and practice in the field of modern social networks.

The priority for further development should be full integration of the plugin system. into the main application, which would allow users to fully exploit the extensibility potential of the platform. At the same time, it would be appropriate to implement more advanced mechanisms for managing permissions and monitoring the performance of plugins, including a system for dynamically allocating resources based on their usage.

A significant area for future research is the implementation of a multi-cluster solution using a Mesh Gateway, which would enable transparent communication between services deployed in different clusters and regions. This approach would significantly increase the availability and resilience of the system to failures of individual data centers.

From a user interface perspective, there is room for implementation of advanced features such as docking tabs and multitasking on a web platform, which would provide a unique user experience different from existing ones

solution.

To optimize performance, it would be appropriate to implement distributed caching using Redis, which could significantly reduce latency and increase system throughput. This optimization would be especially beneficial when scaling.
platform for the largest number of users.

Another direction of research should focus on innovative approaches to social interactions, especially the concept of user circles (circles), which provides more flexible and granular control over content sharing and privacy.
This approach could be further developed by implementing algorithms for automatically recommending relevant circles based on interaction patterns and
interest of users.

In the area of security, it would be appropriate to complete the HashiCorp Vault integration for managing secrets and encryption keys, which would provide a robust solution for management of sensitive information. At the same time, a more advanced system should be developed for detection and prevention of unauthorized access, including the implementation of viral analysis to identify potentially suspicious activities.

The implementation of these recommendations would significantly increase the competitiveness of ness of the Socigy platform and would provide users with an innovative and secure environment for social interactions in digital space.

Chapter 9

Attachments

All attachments and additional materials can be found in the following GitHub repository in the /schemas folder

Literature

1. KEPIOS. Global Social Media Statistics [https://datareportal.com/social-media-users]. DataReportal, 2025. Date cited: 21. Unora 2025.

2. POWELL, Nicole. Social Media Algorithms: How to Crack the Code in 2025 [https : / / www . halconmarketing. com / post / cracking - social-media-algorithms-in-2025]. Halcon, 2024. Date cited: January 15, 2025.

3. NGUYEN, Tien T.; HUI, Pik-Mai; HARPER, F. Maxwell; TERVEEN, Lorraine; KONSTAN, Joseph A. Exploring the Filter Bubble: The Ef-fect of Using Recommender Systems on Content Diversity [https:// archives. iw3c2. org / www2014 / proceedings / proceedings / p677 . pdf]. ACM Press, 2014. Citation date: January 15, 2025.

4. TADDEO, Mariarosaria. Information Warfare: A Philosophical Perspective [https://www.researchgate.net/publication/234627039_ Information_Warfare_A_Philosophical_Perspective]. University of Oxford, 2021. Date of citation: January 15, 2025.

5. MARTIN, Maddie. How Much Money Do You Get Per View on You-Tube? (2025 Stats) [https://www.thinkific.com/blog/youtube-money-per-view]. Thinkific, 2024. Date of citation: January 15, 2025.

6. ALLIANCE, FIDO. What is FIDO2? [https://fidoalliance.org/ fido2/]. FIDO Alliance, [br]. Date of citation: January 15, 2025.

7. JAIN, Ayushi. Decoding Instagram System Design & Architecture (And How Reels Recommendation Works?) [https://www.techaheadcorp.com / blog / decoding - instagram - system - design - architecture - and-how-reels-recommendation-works/]. Tech Ahead, 2024. Date citation: January 15, 2025.

8. BRYANT, Daniel. The Infrastructure Behind Twitter: Scaling Networking, Storage and Provisioning [https://www.techaheadcorp.com/ blog / decoding - tiktok - system - design - architecture/]. Info Q, 2017. Citation date: January 15, 2025.

9. SINHA, Deepak. How TikTok Works: Decoding System Design & Architecture with Recommendation System [https://www.techaheadcorp.com/blog/decoding-tiktok-system-design-architecture/]. Tech Ahead, 2024. Date of citation: January 15, 2025.

10. MDN. Web Authentication API [https://developer.mozilla.org/ en-US/docs/Web/API/Web_Authentication_API]. MDN Web Docs, 2025. Date of citation: January 15, 2025.

11. ALI, Peshawa Jammal Muhammad. Two-Factor Authentication 2FA: An Overview of HOTP and TOTP [https://www.researchgate.net/ profile / Peshawa - Muhammad - Ali / publication / 375867152 _ Two - Factor _ Authentication _ 2FA _ An _ Overview _ of _ HOTP _ and _ TOTP / links/65604e2fce88b8703107f7ac/Two-Factor-Authentication-2FA-An-Overview-of-HOTP-and-TOTP.pdf]. Koya University, 2023. Citation date: January 15, 2025.

Graduation project assignment in computer science subjects

Name and surname: *Patrik Stohanzl*

For the school year: *2024/2025*

Class: *4. A*

Field: *Information Technology 18-20-M/ 01*

Thesis topic: *Design and implementation of a modern social network*

Supervisor: *RNDr. Jan Koupil, Ph.D.*

Method of processing, objectives of the work, instructions on the content and scope of the work:

The goal of this project is to design and develop a modern and innovative social interaction platform that will differentiate itself from existing solutions by integrating plugins and opening up possibilities for the community. The platform will be robust, scalable and focused on the needs of modern users, providing a unique and enriching experience.

Project specifications:

1. **Analysis and design:**
   - o An analysis of existing social networks and their functions will be carried out.
   - o System requirements will be defined and a user interface design will be created interface.
   - o A database structure will be designed for storing user data, contributions, and plugins.
2. **Basic functions:**
   - o **Login/Registration:**
     - ÿ A secure and convenient method for registration will be implemented and user login.
     - ÿ Authentication will be provided using Passkeys for secure and click-free login.
     - ÿ QR code login will be enabled for easy login on other devices.
   - o **Uploading content:**
     - ÿ Users will be allowed to upload photos and texts and share them with other users.
   - o **Viewing shared content:**
     - ÿ The ability to view posts and content from others will be provided users.
   - o **Adding friends:**
     - ÿ Users will be allowed to add their friends and loved ones as friends.
3. **Integration of community plugins (optional point):** o A sandbox environment for safe code execution will be implemented client-side plugins.

- Tools will be provided for community plugin development and integration.

4. **Research and design of the technology stack:**
   - Research will be conducted into available technologies and tools suitable for development of this type of application.
   - A technology stack will be designed based on the research.
   - During implementation, the technology stack will be verified and optimized.
5. **Notifications and messaging:**
   - A notification system will be implemented to inform users about new posts, activities, and plugin updates.
6. **Testing and documentation:**
   - Thorough testing of the application will be performed, including functional tests, user interface tests, and performance tests.
   - User and developer documentation will be created.
7. **Presentation and defense:**
   - A project presentation will be prepared, which will include demonstrations video showing the main features of the application.
   - A project defense will be prepared, including technical details and experience gained.

Required outputs:

- Functional web application for communication and content sharing
- Secure user login and registration methods.
- System for uploading, managing and viewing content.
- Ability to add friends and manage user connections.
- (Optional) Sandbox environment for plugin development and integration.
- Notification and messaging system.
- User and developer documentation.
- Presentation and demonstration video.

Rating:

The project will be evaluated based on the following criteria:

- Quality and functionality of the user interface.
- The ability of the application to meet all set objectives.
- Innovation and originality of the solution.
- System flexibility and adaptability.
- Level of project documentation and presentation.
- Creativity and effectiveness of solutions.

Brief timeline (with dates and specific tasks): • **September:** Problem analysis, creation of UML usecase diagrams, application design
• **October-November:** Backend development (registration, authentication, file management, communication processes and API, deployment issues)
• **December:** Frontend development, preparation for plugins
• **January:** Project finalization
• **February - March:** Documentation work and bug fixing