

Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет _____ Комп'ютерних наук _____
(повна назва)

Кафедра _____ Комп'ютерного моделювання та інтелектуальних технологій _____
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти _____ другий (магістерський) _____
Дослідження методів комунікації сервісів у мікросервісній архітектурі /
Research on service communication methods in microservice architecture
(тема)

Виконав:
студент 2 курсу, групи СПРМ-24-1
Голуб Д. К.
(прізвище, ініціали)

Спеціальність ЕЗ Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Системне проектування
(повна назва освітньої програми)

Керівник професор каф. КМІТ Вишняк М. Ю.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____
(підпис)

Гребінник І. В.
(прізвище, ініціали)

2026 р.

Я як студент ХНУРЕ розумію і підтримую політику закладу із академічної доброчесності. Я не надавав і не одержував недозволену допомогу під час підготовки кваліфікаційної роботи. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

11.05.2026



Голуб Дмитро Костянтинович

(дата, підпис, прізвище студента/-ки)

Кваліфікаційна робота не містить відомостей заборонених до відкритого опублікування.

Кваліфікаційна робота виконана у відповідності до стандартів, що діють в Україні.

Попередній захист проведено 05 травня 2026 р.

Керівник кваліфікаційної роботи Вишняк М. Ю.



Харківський національний університет радіоелектроніки

Факультет _____ Комп'ютерних наук
Кафедра _____ Комп'ютерного моделювання та інтелектуальних технологій
Рівень вищої освіти _____ другий (магістерський)
Спеціальність _____ ФЗ Комп'ютерні науки
(код і повна назва)
Тип програми _____ освітньо-наукова
Освітня програма _____ Системне проектування
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри _____

(підпис)
« 18 » 03 2026 р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Студентові _____ Голуб Дмитру Костянтиновичу
(прізвище, ім'я, по батькові)

1. Тема роботи _____ Дослідження методів комунікації сервісів у мікросервісній архітектурі / Research on service communication methods in microservice architecture.

затверджена наказом університету від 06 квітня 2026 р. № 268 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 11.05.26

3. Вихідні дані до роботи: Оформлення пояснювальної записки та захист кваліфікаційної роботи виконується англійською мовою. Аналоги бібліотек для роботи з брокерами повідомлень. Перелік використовуваних програмних засобів: AllFusion ProcessModeler, DiagramLy, JetBrains WebStorm.

4. Перелік питань, що потрібно опрацювати в роботі: провести аналіз предметної області та продуктів конкурентів, визначення сфери застосування системи, формування постановки задачі. Висунути функціональні та нефункціональні вимоги до розроблюваного рішення, реалізувати рекомендоване рішення, інтегрувати рішення у інформаційну систему для перевірки функціональності рішення, провести експериментальні дослідження продуктивності розробленого рішення.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій: 6 слайдів та додаток А з деталізацією окремих рисунків.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Отримання завдання атестаційної роботи	26.01.26	
2	Аналіз завдання, літератури та аналогів з теми атестаційної роботи	15.02.26	
3	Вибір засобів для розробки технічних вимог до програми	27.02.26	
4	Структурне проектування	14.03.26	
5	Вибір середовища розробки програми	24.03.26	
6	Розробка програми	05.04.26	
7	Розробка демонстраційних прикладів	10.04.26	
8	Тестування та оцінка розробленого рішення	17.04.26	
9	Оформлення пояснювальної записки та програмної документації	21.04.26	
10	Оформлення графічної частини та презентаційних матеріалів	28.04.26	
11	Представлення на рецензування	03.05.26	
12	Занесення диплому в електронний архів	04.05.26	
13	Отримання допуску у зав. кафедри	07.05.26	

Дата видачі завдання «18» березня 2026р.

Студент _____  _____ Голуб Д. К.

(підпис)

Керівник роботи _____  _____ Вишняк М. Ю.

(підпис)

РЕФЕРАТ

Пояснювальна записка містить 85 аркушів, 32 рисунки, 5 таблиць, 2 додатки, 25 джерел, графічний матеріал

МІКРОСЕРВІСНА АРХІТЕКТУРА, МОНОЛІТНА АРХІТЕКТУРА, ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ, ОБМІН ПОВІДОМЛЕННЯМИ, МАСШТАБОВАНІСТЬ, СЕРВЕРНА ЧАСТИНА, АРХІТЕКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ІНФОРМАЦІЙНА.

Об'єктом дослідження кваліфікаційної роботи є процес розробки технології для уніфікації обміну повідомленнями між сервісами у мікросервісній архітектурі.

Предметом дослідження є інформаційні технології та програмні методи створення інформаційних систем на базі мікросервісної та монолітної архітектури, сучасні протоколи, патерни та технологічні рішення, що застосовуються для організації взаємодії в мікросервісних екосистемах.

Метою кваліфікаційної роботи є дослідження методів взаємодії сервісів у мікросервісній архітектурі, дослідження архітектурних варіації серверної частини інформаційної системи.

Методи досліджень кваліфікаційної роботи включають системний підхід, методи структурного аналізу, методи об'єктно-орієнтованого програмування, методи функціонального програмування, методи мікросервісного проектування (декомпозиція застосунку на сервіси, визначення функціональних меж сервісів, комунікація між сервісами).

Результатом цієї кваліфікаційної роботи є бібліотека, що дозволяє уніфіковано будувати мікросервісну архітектуру, абстрагуючись від брокера повідомлень, що використовується на проєкті та дозволяє швидко та зручно розгортати мікросервісну архітектуру.

Область застосування цієї кваліфікаційної роботи – розробка та оптимізація програмного забезпечення для здійснення зв'язку між мікросервісами у межах мікросервісної архітектури.

ABSTRACT

The explanatory note contains 85 pages, 32 figures, 5 tables, 2 appendices, 25 sources, graphic material

MICROSERVICE ARCHITECTURE, MONOLITHIC ARCHITECTURE, OBJECT-ORIENTED PROGRAMMING, MESSAGING, SCALABILITY, SERVER PART, SOFTWARE ARCHITECTURE, INFORMATION.

The object of the qualification work is the process of developing technology for unifying message exchange between services in microservice architecture.

The subject of research is information technologies and software methods for creating information systems based on microservice and monolithic architecture, modern protocols, patterns, and technological solutions used to organize interaction in microservice ecosystems.

The purpose of the qualification work is to study methods of interaction between services in microservice architecture and architectural variations of the server part of an information system.

The research methods used in the thesis include a systematic approach, structural analysis methods, object-oriented programming methods, functional programming methods, and microservice design methods (decomposition of an application into services, definition of functional boundaries of services, communication between services).

The result of this qualification work is a library that allows developers to build a microservice architecture in a unified way, abstracting from the message broker used in the project and allowing developers to quickly and conveniently deploy a microservice architecture.

The scope of this qualification work is the development and optimization of software for communication between microservices within a microservice architecture.

CONTENT

Introduction	8
1 Analysis of subject area.....	11
1.1 Overview of backend architectures	11
1.1.1 Architectural features of monolithic architecture	12
1.1.2 Architectural features of microservice architecture	13
1.1.3 Features of messaging in microservice architecture	16
1.1.4 Comparison of monolithic and microservice architecture	17
1.2 Modern approaches to application development in microservice architecture	19
1.2.1 Modern message brokers and their comparative analysis.....	19
1.2.2 Analyzing products with similar functionality	22
1.3 Problem statement	23
2 Development of unified messageing environment technology.....	24
2.1 Developing requirements of messaging environment.....	24
2.2 Justification of technology stack selection.....	25
2.3 Defining the architectural features of the library under development	26
2.4 Defining algorithms for working with message brokers.....	31
2.5 Development of library components and features of integration into the project	38
3 Design and development of an experimental information system.....	47
3.1 Description of the subject area	47
3.2 Development of functional requirements for the web-application	48
3.3 Design and development of information system components	63
4 Conducting experiments and comparing the results obtained	71
4.1 Performance evaluation of the developed library in the context of different message brokers	71
4.2 Analysis of the potential use of the developed software solution in practice	78
4.3 Exploring the advantages of using microservice architecture.....	80
Conclusions	82
List of referenced sources	84
APPENDIX A Graphical materials.....	86
APPENDIX B Program text.....	93

INTRODUCTION

Modern backend development is characterized by rapidly increasing business requirements and scaling of information systems. Monolithic architectures [1], which dominated at the dawn of web development, often fail to cope with the loads imposed on highly loaded and rapidly evolving applications. In response to these challenges, the software development industry has embraced the microservice architecture [2] (MSA) paradigm, which proposes decomposing a complex system into a set of small, loosely coupled, and independently deployable services.

The transition to microservice architecture promises significant benefits. It allows different development teams to work on individual services autonomously, choosing the most appropriate technology stacks and development methodologies for specific tasks. This increases the speed of new feature release (time-to-market) and the overall flexibility of the development process. In addition, MSA provides better fault tolerance: the failure of one service does not necessarily lead to the collapse of the entire system. Another important advantage is the ability to independently scale individual components depending on their load, which leads to more optimal use of resources.

However, these advantages come at a price of increased complexity. The key challenge in the design and operation of distributed systems is to organize reliable, efficient and secure interaction between multiple isolated services. Problems that in a monolith were solved by simple method calls are transformed into networking tasks in a microservice ecosystem. This gives rise to a whole range of new complexities: the need for resilience to network failures, data consistency in a distributed transaction, service discovery, configuration management, distributed trace monitoring, and centralized logging.

Thus, efficient service interaction is the cornerstone of successful implementation of a microservice architecture. The choice of methods and technologies for this interaction (such as synchronous HTTP/REST APIs [3],

asynchronous messages via brokers, etc.) directly affects the performance, reliability and maintainability of the whole system.

One of the problems in implementing asynchronous communication in microservice systems today is the rigid binding of business logic (logic that practically implements the needs of the subject area) to a specific implementation of a message broker. In fact, each new integration with the queue developers have to implement practically from scratch, creating custom adapters for a specific project and technology. This results in significant development and support overhead, code duplication, and reduced architecture flexibility. Teams are constantly “reinventing the wheel,” spending time writing and debugging one-size-fits-all code to connect to brokers instead of focusing on solving substantive business problems. This results in a situation when even within one organization different projects use completely different, incompatible approaches to working with queues, which makes it difficult to migrate between solutions, maintain code and rotate developers between projects.

The relevance of the topic is due to the wide spreading of microservice approaches in building enterprise solutions and highly loaded platforms [4]. However, as the complexity of systems grows, the strong dependence of application business logic on specific implementations of message brokers becomes an acute problem. In practice, this leads to a situation when each new integration with the queue or transition to another technology (for example, from RabbitMQ [5] to Kafka [6] or NATS [7]) developers have to implement from scratch, creating custom adapters for a particular project. This generates significant development and support overhead, reduces architecture flexibility, and makes migration between solutions difficult. As a result, teams face vendor lock-in at the infrastructure level, which limits scalability and optimization costs. In this context, researching inter-service abstraction techniques and developing a unified solution to handle different message brokers seems to be a critical challenge to create truly flexible, maintainable and efficient distributed systems.

The purpose of the qualification work is to study the methods of interaction of services in microservice architecture, the study of architectural variations of the server part of the information system.

The object of the study is the process of designing and developing technology to unify and standardise the exchange of messages between services in microservice architecture.

The subject of the study is information technologies and software methods for creating information systems based on microservice and monolithic architecture, modern protocols, patterns and technological solutions used to organize interaction in microservice ecosystems.

In order to achieve this goal, the following tasks will be solved: to analyze the existing methods and protocols of interaction, to highlight their advantages, disadvantages and areas of application; to design and develop a unified solution in the form of a library [8], which will abstract the actual message broker and allow quick and convenient integration into the project; to design an experimental information system on which the integration of the developed solution will be tested; to compare the efficiency of the developed solution.

1 ANALYSIS OF SUBJECT AREA

1.1 Overview of backend architectures

Modern software development is characterized by continuous complication of functional requirements and scaling of information systems. The evolution of software products naturally leads to the growth of their architectural complexity, which in turn creates an objective need to apply modular design principles. Currently, the industry has formed three main approaches to the architectural organization of applications: monolithic, microservice and hybrid.

Monolithic architecture, historically the most common approach, involves the development of a single logically indivisible component containing all of the application's business logic. While easy to develop and deploy in the initial stages, this approach demonstrates significant limitations in scaling and evolving complex systems, exhibiting low fault tolerance and creating barriers to the adoption of new technologies.

Microservice architecture emerged as an answer to the challenges associated with the development of complex distributed systems. This approach is based on the decomposition of an application into a set of loosely coupled services, each of which encapsulates a specific business opportunity and can be developed, deployed and scaled independently. The key advantages of the microservice paradigm are: increased system fault tolerance, the ability to independently scale components, technological heterogeneity and accelerated development processes by organizing the work of small autonomous teams.

Hybrid architecture is a compromise solution combining elements of monolithic and microservice approaches. This model is often used in scenarios of gradual migration from monolith to microservices or in cases when it is inexpedient to separate certain system components into separate services. Within the framework of this study, such an architecture will be equated to a microservice, since the difference is largely only in the scope that can be covered by a single service.

1.1.1 Architectural features of monolithic architecture

Monolithic architecture is a classic approach to software system design where all components of an application are integrated into a single logical executable unit. This architectural style is characterized by a unified code base where all business logic, and data layer are tightly coupled and function within a common process.

The main feature of the monolithic architecture is the ease of initial development and deployment. All modules of the system (often excluding front-end) are compiled and deployed as a single unit, which minimizes the complexities of coordination between different parts of the application in the initial stages of the project. Interaction between components is performed through fast in-process calls, which ensures high performance and low latency in processing operations.

The architecture assumes the use of common resources and dependencies by all modules of the system. This makes it easy to organize transactional data and ensure consistency of the system state through a single database. Debugging and monitoring of the application is simplified because all components are executed in a single process and use common logging mechanisms.

Monolithic systems demonstrate ease of management and administration through centralized configuration and a single deployment process. Security is also simplified with a single point of entry and a common authentication and authorization mechanism.

The technology stack of a monolithic system typically remains unified throughout the application lifecycle, which creates limitations for using specialized technologies for different tasks. Development and deployment become slower as the code base grows, as even small changes require a complete rebuild and testing of the entire system.

1.1.2 Architectural features of microservice architecture

Microservice architecture is a modern approach to designing software systems based on decomposition of an application into a set of loosely coupled autonomous services. Each service encapsulates a certain business opportunity and functions as an independent process, which provides a high degree of modularity and flexibility of the system.

A key feature of this architecture is the independence of deployment and scaling of individual services. Each microservice is developed, deployed [9] and scaled autonomously to optimize resource utilization and ensure system elasticity according to the load on specific functional components. This allows teams to work independently on different services, choosing the most appropriate technologies and stacks for specific tasks.

The architecture assumes organization of interaction between services through lightweight mechanisms, typically message brokers. Services have their own subject area and are responsible for their own data and its storage, which facilitates a clear division of responsibility in the system. Each service can use the most suitable database for its tasks, realizing the principle of polyglot persistence.

Fault tolerance of the system is provided by fault isolation - problems in one service do not lead to complete failure of the whole system. Microservice architecture naturally supports distributed computing and allows to efficiently distribute the load between different nodes of the system.

Development within the framework of microservice architecture implies adherence to the principle of bounded context, where each service has clearly defined boundaries of responsibility and interacts with other services through well-defined interfaces. This promotes evolutionary development of the system and the possibility of gradual introduction of changes without the need to modify the entire system.

In distributed systems, business process coordination is realized through two fundamentally different approaches: orchestration and choreography. Orchestration involves a central controlling component that explicitly dictates the sequence of

operations by invoking the appropriate services and controlling the state of the process. In contrast, choreography is a decentralized model where services coordinate their work through the exchange of events without centralized control. Each service reacts independently to relevant events in the system, which creates a more flexible, loosely coupled and fault-tolerant architecture. It is choreography that most closely matches the philosophy of microservice systems, providing maximum autonomy of services and evolutionary flexibility of the architecture. Examples of different types of message coordinations are shown in the figure 1.1.

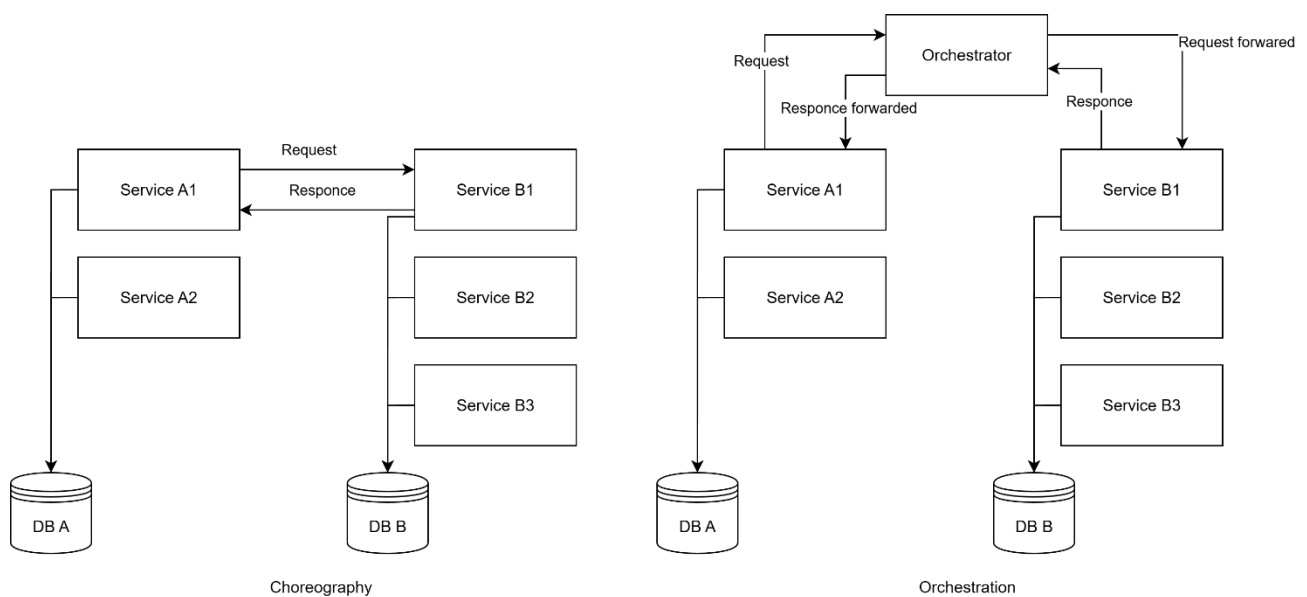


Figure 1.1 – Example of orchestration and choreography messaging approaches

Various communication patterns are used to exchange messages between services. Despite the development of asynchronous approaches, synchronous request-response remains the most common communication pattern in microservice systems. This pattern dominates in scenarios that require immediate confirmation of an operation or data retrieval to proceed, such as user authentication, payment processing, or obtaining up-to-date information from different services. These challenges have led to the emergence of asynchronous implementation of the request-response pattern via message brokers. This approach combines the benefits of the familiar request-response pattern with the benefits of asynchronous communication,

such as increased resilience, better handling of peak loads, and reduced connectivity between services.

A message broker is a software component that mediates data exchange between different parts of a distributed system. Its primary function is to receive, store, and then deliver messages from senders to receivers, separating data generation from processing. This approach improves system reliability and scalability by enabling asynchronous interaction between services and reducing interdependencies. A message broker can also perform routing, load balancing, and guaranteed delivery, making it a key element of architectures focused on event-based and streaming data processing. Most often, in service-oriented architecture, a message broker is used to create a high-performance, reliable, secure, and externally inaccessible environment within which messages are passed between parts of a distributed system. A schematic example of the role of a message broker in a microservice architecture is shown in Figure 1.2.

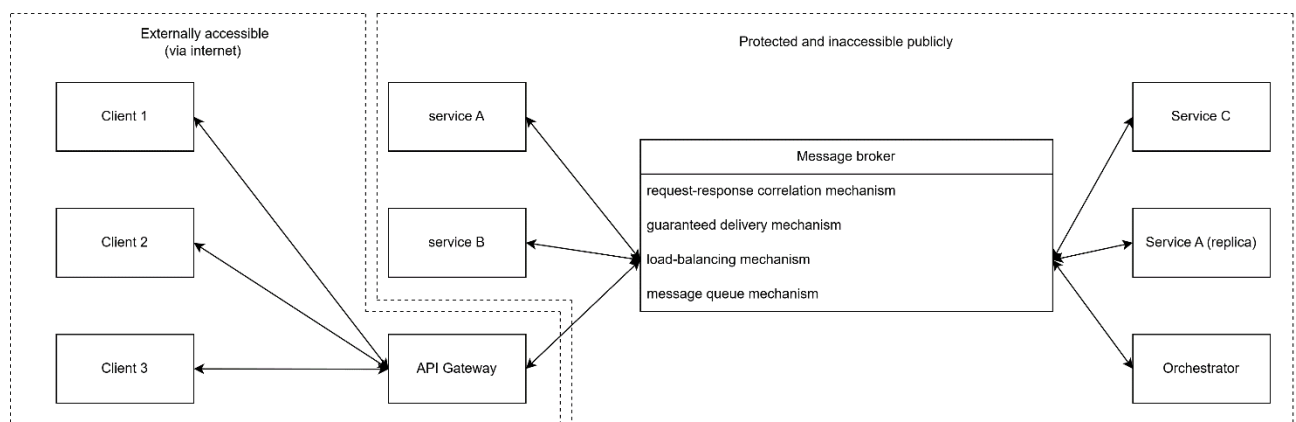


Figure 1.2 – Schematic representation of message broker role in microservice architecture

Among modern message brokers supporting asynchronous request-response, the most prominent solutions include NATS with its built-in support for the request-reply pattern via core functionality, Apache Kafka using correlated requests and responses, RabbitMQ with its implementation via RPC mechanisms and temporary response queues, and Redis Streams [10] with its message handling and identifier correlation capabilities. Each of these brokers offers unique characteristics of

performance, scalability, and reliability, making them applicable in a variety of asynchronous request-response usage scenarios.

The choice of a particular message broker to implement asynchronous request-response depends on many factors, including latency requirements, throughput, message delivery guarantees, data persistence, and the complexity of setting up and maintaining the infrastructure. The key challenge remains ensuring proper request-response correlation, timeout management, and error handling in an asynchronous environment.

1.1.3 Features of messaging in microservice architecture

In a microservice architecture, message exchange via a broker represents a fundamental mechanism for interaction between services. Among the variety of communication patterns, several key approaches stand out, each serving specific purposes and usage scenarios.

Broadcast allows a message to be sent simultaneously to all subscribers of the broker, which is particularly useful for notifications of configuration changes or propagation of system events. This approach provides better consistency in distributed systems, but requires careful design of duplicate handling and idempotency.

Send messages without waiting for a response (fire-and-forget) is an asynchronous one-way pattern where the sender does not wait for an acknowledgement or processing result. This method is ideal for logging, auditing, or asynchronous task processing scenarios where no immediate system response is required.

Publish-subscribe organizes communication through topics or queues where producers publish messages and consumers subscribe to categories of messages that interest them. This pattern is effective for implementing event-driven architecture and decomposition of complex business processes.

However, the request-response pattern, implemented through message broker mechanisms, remains the most widespread and easy to develop. Despite the asynchronous nature of messaging systems, this approach successfully mimics synchronous communication while retaining the advantages of asynchrony. A sender service publishes a request to a specific queue or topic, specifying a reply address, and waits for a correlated response in the specified time queue.

The popularity of request-response is due to its conceptual simplicity and familiarity to developers. The pattern allows natural integration of asynchronous communication into existing workflows, while ensuring high performance and fault tolerance.

Within a broker, request-response can be implemented in different ways: through temporary response queues, correlation identifiers, or specialized mechanisms like NATS Request-Reply. Each approach has its own trade-offs between performance, reliability and complexity of implementation, but all of them retain the key advantages of asynchronous communication - reduced service connectivity, better handling of peak loads and increased fault tolerance of the system as a whole.

1.1.4 Comparison of monolithic and microservice architecture

Comparison of monolithic and microservice architectures reveals fundamentally different approaches to organizing software systems. Monolithic architecture, being a traditional and time-tested model, offers simplicity of initial development and deployment. All system components are tightly integrated and function as a single entity, which provides high performance intraprocess communication and simplified transaction management. However, this simplicity comes at the cost of significant limitations when scaling and evolving the system. As application complexity grows, the monolith demonstrates decreased flexibility, increased development time, and difficulty in adopting new technologies.

Microservice architecture emerged as an answer to the challenges of scaling and evolution of complex systems. Decomposition of an application into a set of independent services allows to achieve unprecedented flexibility and scalability. Each service can be developed, deployed and scaled independently, which ensures high speed of change implementation and optimization of resource usage. Technological heterogeneity allows you to select optimal tools for specific tasks, and fault isolation prevents cascading failures.

However, the benefits of microservice architecture come with a significant increase in operational complexity. The distributed nature of the system requires the implementation of sophisticated mechanisms for service discovery, monitoring, configuration management and data consistency. Interaction between services through network calls introduces latency and potential points of failure not present in monolithic systems.

A key difference between the architectures is in their evolutionary trajectory. Monolithic systems are difficult to maintain and evolve as their complexity grows, while microservice systems are designed for continuous evolution and scaling. The choice between architectures is a trade-off between operational simplicity and evolutionary flexibility, where the optimal solution depends on the specific requirements of the project, the scale of the system and the expertise of the team.

Figure 1.3 shows structural diagrams of three types of information system architectures: monolithic, microservice without gateway (when the microservice itself provides an interface for data exchange with the client part of the application) and microservice with gateway (when there is a separate service that receives client requests and redirects them to services within the infrastructure).



Figure 1.3 – Structural diagram of different information system architectures

Studying figure 1.3, it is worth considering that when it is necessary to scale the application, in case of using the monolithic architecture, it will be necessary to allocate multiple times more resources, because replication will be performed over the entire set of business logic, even over the one that probably does not suffer from overloads. In case of using the microservice architecture, the need for replication will arise only over the overloaded resource. Using a gateway [11] can be useful when there is a need to unify the client interface, but it adds overhead to maintaining the gateway itself, and in case of failure, completely blocks access to the infrastructure for the client.

1.2 Modern approaches to application development in microservice architecture

1.2.1 Modern message brokers and their comparative analysis

In modern microservice architectures, message brokers play a key role in providing reliable and scalable communication between services. Among the many solutions available, NATS, Apache Kafka, RabbitMQ, and Redis Streams are the

most widely used, each with unique characteristics and optimized for specific use cases.

NATS is characterized by exceptional simplicity and performance. Its lightweight design and minimalistic API make it ideal for highly loaded systems where low latency and high throughput are important. The broker is memory-efficient and supports several communication models, including publish-subscribe, request-reply, and queues. However, the simplicity of NATS comes at a price: the lack of default message persistence and limited persistence capabilities make it less suitable for systems requiring guaranteed message persistence.

Apache Kafka is a distributed, data-intensive streaming platform for data processing. Its key advantage is its ability to reliably store and process real-time event streams while providing high throughput and fault tolerance. Kafka is ideal for scenarios that require processing and analyzing large data streams, auditing events, and building complex ETL processes. However, the complexity of setting up and operating Kafka and its relatively high latency make it redundant for simple messaging tasks.

RabbitMQ is one of the most mature and feature-rich message brokers. Its main strength is its flexible routing of messages through different types of exchanges and support for advanced messaging protocols. RabbitMQ provides reliable message delivery with persistence and extensive queue management capabilities. This broker is well suited for complex business workflows that require fine-tuned routing and guaranteed delivery. At the same time, its performance is slower than NATS, and customization requires more advanced knowledge.

Redis Streams offers a unique combination of messaging and in-memory data store capabilities. As part of the Redis ecosystem, it provides high performance and ease of use, supporting message persistence and consumer groups. Redis Streams is effective for scenarios where message processing speed is important and you need to integrate messaging with other Redis capabilities such as caching or working with data structures. However, its messaging functionality is more limited compared to specialized brokers, and scaling has its own peculiarities.

The choice of a particular broker depends on project requirements: NATS is optimal for high-performance systems with ephemeral messaging, Kafka for handling large data streams, RabbitMQ for complex business processes with demanding delivery guarantees, and Redis Streams for scenarios where integration of messaging with other in-memory operations is important.

In the table 1.1 a comparison analysis over described message brokers is perfumed.

Table 1.1 – Comparison analysis of message brokers

Message broker Feature	NATS	Apache Kafka	RabbitMQ	Redis Streams
Persistency	Optional	+	Optional	+
Bandwidth (in request-reply mode)	High	Low	Medium	Medium
Latency	Low	High	Low	Medium
Complexity of administration	Low	High	Medium	Low
Main use-cases	IoT, Real-time	Big data, event sourcing	Business-workflows, queueing tasks	Real-time analytics, caching
Communication model	Pub-Sub, Request-Reply	Pub-Sub	Work Queues, RPC	Pub-Sub, Consumer Groups
Additional aspects	Lightweight, easy to setup	Complex and heavy, highly reliable	extensive queue management	Lightweight, used in project for many purposes (not only as a message broker)

Evaluating the comparison table above, it is difficult to say with precision which message broker is better, they all have their own peculiarities and spheres of application. However, within the framework of this research, NATS will be connected to the experimental information system (information system of a social network) because it is best suited for creating MVP platforms, also, NATS provides high throughput, which is desirable in real-time information systems.

1.2.2 Analyzing products with similar functionality

Having conducted a detailed analysis of existing solutions that solve the problem of unification of methods of service communication in microservice architecture, only two main analogs were found, namely Moleculer and NestJS. It should be noted that NestJS is a complete ecosystem for developing WEB-applications and solves the problem of building a microservice architecture only indirectly. Moleculer is a heavy, complex framework for building microservice architecture. Moleculer is good, but it is poorly suited for small projects and startups because of its overall bulkiness and complexity of configuration.

It is worth noting that the task of developing a unified environment for messaging between microservices arises due to the growth and development of applications and the need to change the technology stack at some point. Also, the reason for such a task is the necessity to re-write for each separate project a wrapper over the actual message broker for the tasks of the project. Startups spend a lot of time and effort on debugging the wrapper and its implementation, and if using Moleculer or NestJS will solve the task of project development, then, due to the heavy-handedness of these solutions, they will be bad for a startup, forcing developers to “reinvent the wheel”.

1.3 Problem statement

Having analyzed the peculiarities and problems of modern application and information system development in the microservice paradigm, we have identified the problem of lack of a lightweight and universal solution for designing microservice architecture that allows to quickly and conveniently provide communication between services using an abstract message broker (i.e., one whose actual implementation does not change the high-level code with which the developer interacts). Such a tool will allow small projects and startups not to implement similar message broker wrappers multiple times, but to focus on implementing useful business logic, and larger projects - to change the message broker technology without rewriting the entire code base. It is required that the solution be implemented as a library for Node.js (since this is the technology most often used by startups due to its lightweight nature, high development speed and high performance). The solution should allow abstract use of such message brokers as NATS, Kafka, Redis Streams, RabbitMQ, client code implementation should not change when changing the message broker. In the framework of this research, the following steps will be carried out:

- Analysis of the varieties of architectures and approaches to the design of applications in microservice architecture, selection of functions to be implemented;
- Analysis and justification of the technical stack;
- Design of the architecture of the developed library (compilation of UML Class Diagrams), documentation of the used algorithms (compilation of UML Activity Diagrams, UML Sequence Diagrams);
- Development of the library program code and description of the interface of the developed library;
- Designing and developing an experimental information system;
- Conducting experimental studies on the developed library;
- Drawing conclusions.

2 DEVELOPMENT OF UNIFIED MESSAGEING ENVIRONMENT TECHNOLOGY

2.1 Developing requirements of messaging environment

In the “Subject Area Analysis” section, a detailed study of the architectural features of microservice architecture was conducted, the most popular message brokers were identified, and a study was conducted as to which message exchange paradigm in microservice architecture is most usable. After conducting a comprehensive analysis, we identified the problem of developing a unified intermediate layer between the business logic and the message broker. The complexity of realizing such a solution depends on the significant variety of different message brokers in terms of their internals and API for interacting with them, which is available to the developer.

In the framework of this work, we propose to design such a middleware layer (wrapper), which does not assume a change of interaction with the API provided by the library depending on the physically used message broker. It is proposed that among the methods of service interaction in a microservice architecture, two messaging patterns will be used: request-reply (as the most useful in designing microservice interaction) and fire-and-forget (as a modification of the request-reply pattern for delivering messages that do not require a response). It is proposed to design the library to support such message brokers as NATS, Apache Kafka, RabbitMQ and Redis Streams. It was decided to write the library in the TypeScript programming language for use in the technology stack with Node.js [12] (version 18 and above).

The library under development should be a lightweight solution that beginner projects or startups can easily and conveniently integrate into their projects (eliminating the need to develop a similar one from scratch), also, this development should be useful for projects that for some reason decided to change the message broker, in this case, they will not have to rewrite their entire code base for the new

message broker, it will be enough only to slightly update the configuration of services. It is also assumed that the developers of the final project, if necessary, will be able to develop their own layer-adapter for any other unrepresented message broker and use it together with the library being developed.

2.2 Justification of technology stack selection

The choice of a technology stack for the realization of a cross-service interaction system is driven by the requirements for performance, ease of integration and meeting the needs of startups and small projects.

Node.js is a V8-based JavaScript runtime designed for building scalable, networked applications. In the context of startups and microservice architectures, Node.js demonstrates exceptional efficiency due to its non-blocking I/O model and event-driven architecture, which allows processing multiple simultaneous connections with minimal resources. The technology accelerates development by unifying the language across frontend and backend, a broad ecosystem of npm packages, and rapid prototyping capabilities.

TypeScript [13] is a strictly typed subset of JavaScript, adding static code analysis. TypeScript provides increased reliability through type checking at compile time. The technology reduces runtime errors and improves code autocompletion and refactoring, which is especially valuable in fast-growing startups with frequent requirements changes.

NATS is a high-performance and lightweight broker for cloud environments, ideal for request-response workloads thanks to its built-in correlation engine. It features low latency and ease of use.

Apache Kafka is a distributed stream processing platform that delivers high throughput and reliable message delivery. It is in demand in mission-critical processes that require guaranteed delivery and data persistence.

RabbitMQ is a classic broker with flexible routing support and an RPC queuing mechanism. It is popular in enterprise environments due to its maturity, predictability, and diverse use cases.

Redis Streams is an in-memory stream processing module that combines high throughput with messaging capabilities. It is effective where low latency and integration with caching or session state are required.

2.3 Defining the architectural features of the library under development

A UML Class Diagram [14] is a static structural diagram in the Unified Modeling Language (UML) used to visually represent the logical architecture of an object-oriented system. It defines the system's classes, their attributes (data), methods (behaviors), and the relationships among them – such as association, inheritance, aggregation, and composition. By providing a clear visual blueprint of object-oriented components, the diagram helps both to design system architecture and to document existing code. Unlike behavioral UML diagrams, which describe dynamic processes, the class diagram provides a static blueprint—essential for both forward design (showing how code should be structured) and reverse engineering (documenting an existing system). As the most widely used UML diagram, it helps developers, analysts, and stakeholders understand the data model and dependencies before any code is written.

Having an idea of the library being designed, we can create a UML Class Diagram to better understand the processes occurring within the infrastructure of the solution being developed. Figure 2.1 shows the class diagram of the library.

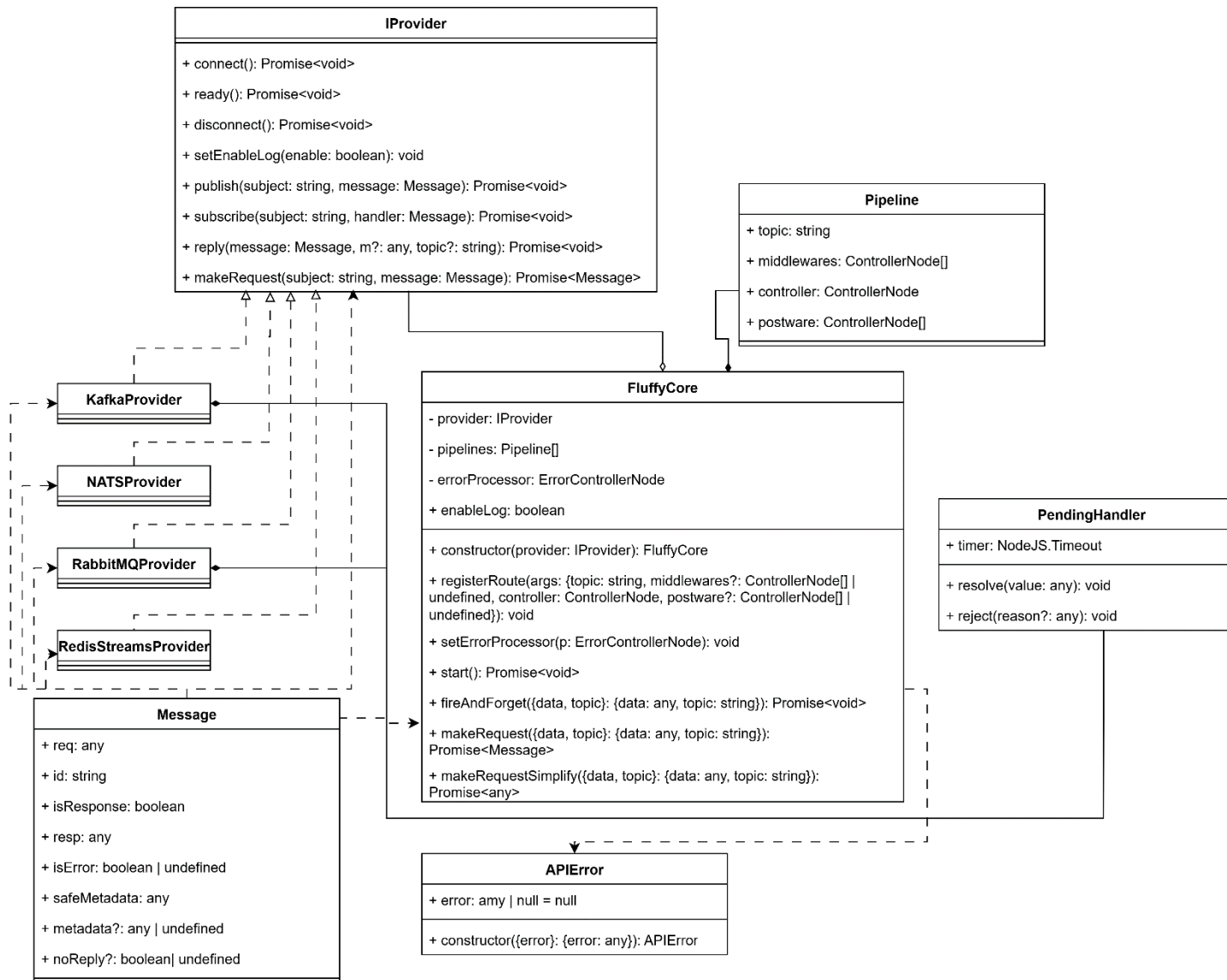


Figure 2.1 – Simplified UML Class diagram of the library

In Figure 2.1 we can see a class diagram of a library designed to implement the logic of message processing in a microservice architecture.

We can see that at the head of all this logic is the `IProvider` interface [15]. `IProvider` defines the structure that provider classes should have. In our case, provider classes implement the “Adapter” design pattern, they bring initially fundamentally incompatible interfaces of different message brokers to a common interface (`IProvider`) and repeat the behavior of a unified provider as closely as possible, regardless of the actual technology used. The interface declares the `connect` method - to connect to the message broker, `ready` - to confirm that the configuration is complete and you can start processing messages, `disconnect` - to disconnect from the message broker, `setEnableLog` - to modify the logging settings, `publish` - to send a message on a certain topic, `subscribe` - to subscribe and further receive messages on the topic, `reply` - to reply to the message and `makeRequest` - to execute the request.

`NATSProvider` is an adapter class for the `IProvider` interface designed to work with NATS, implements all the methods specified in the interface, also has a constructor that accepts the `url` argument, stores private fields `connection` - an object for communication with the NATS message broker, `codec` - an object that allows correct serialization and deserialization of messages, `enableLog` flag and the `url` parameter (with which the object was initialized).

`KafkaProvider` is an adapter class for the `IProvider` interface designed to work with ApacheKafka, implements all the methods specified in the interface, also has a constructor that accepts the `brokers` argument (an array of brokers' addresses), `clientId` - a string identifier of the current Kafka client, `groupId` - a string identifier of the consumer group. All consumers with the same `groupId` share topics partitions, which provides horizontal scaling, stores private fields `kafka` - entry point for communication with Kafka message broker, `producer` - kafka object responsible for sending messages, `consumer` object responsible for receiving messages and subscribing to topics, `flag` to enable logging `enableLog`, fields passed to the constructor (`brokers`, `clientId`, `groupId`), `pendingRequests` is a field responsible for pending messages, where the key is the `correlationId` of the message and the value is

the PendingHandler object (which is a slightly complicated promise with timeout control), responseTopic is a service topic to which the consumer subscribes at startup, topicHandlers is a map of subscribers, where the key is the topic on which the message was received and the value is the handler.

RabbitMQProvider – class-adaptor for IProvider interface, designed to work with RabbitMQ, implements all methods specified in the interface, also has a constructor that takes the argument url - address of the message broker. Stores private fields connection - entry point for communication with the message broker, channel - working channel on top of connection. All operations are performed through it, enableLog flag, url field, pendingRequests - a field responsible for pending messages, where the key is the correlationId of the message and the value is the PendingHandler object, responseQueue is a service queue for request-reply responses, wasResponseQueueCreated - a flag indicating whether a message response queue has been created, topicHandlers - a map of subscribers, where the key is the topic that received the message and the value is the handler.

RedisStreamsProvider is an adapter class for the IProvider interface designed to work with Redis Streams, implements all the methods specified in the interface, also has a constructor that accepts the argument host, port, consumerId - string-identifier of the current consumer client within the group, groupId - string-identifier of the consumer group, stores private fields producer - ioredis object responsible for sending messages, consumer object, which is responsible for receiving messages and subscribing to the topics, enableLog flag, wasResponseQueueCreated - a flag indicating whether the queue of responses to messages has been created, fields passed to the constructor (host, port, consumerId, groupId), pendingRequests - a field responsible for pending messages, isConsuming - a flag controlling the cycle of receiving messages from the queue, responseTopic - a service topic to which the consumer subscribes at startup.

In addition to adapters and interface, the FluffyCore [16] class plays an important role in the architecture - it is the main class of the developed library, its high-level API. This class can be considered as the central element of the library,

unifying the functionality of providers and providing high-level management of messaging. FluffyCore uses an IProvider instance, stores a set of pipelines and error handling controllers. It can be used to register routes, assign intermediate handlers, start the system, and manage both fire-and-forget and request-reply messaging. Essentially, FluffyCore implements the business logic layer and provides a convenient interface for interacting with microservices, hiding the low-level implementation details of each broker from the developer. The FluffyCore class has fields: provider for storing an object implementing IProvider, pipelines - an array of objects of Pipeline type, errorProcessor - a special controller intended for customization of error handling and a flag responsible for enabling logging - enableLog. The constructor of the object takes an instance of IProvider. The object presents methods: RegisterRoute to add a method to which other services can send a request, setErrorProcessor to add a custom error handler, start to initialize FluffyCore and all internal logic, fireAndForget to send a message without waiting for a response, makeRequest and makeRequestSimplify - to send a request and wait for a response, makeRequest will return a Message object with all the information about the message, makeRequestSimplify will return only the contents of the resp field of the Message object or throw an exception if the isError field of the Message object is true.

Pipeline defines the structure of the message handler, including a sequence of intermediate handlers (for pre-processing of the message) - middlewares, the main controller (where all user business logic is executed) - controller and post-processors - postware, which form a complete cycle of processing the data arriving on a certain topic, Pipeline also contains the topic field - this is the key, by which another service can send a message.

APIError – is a class for storing information about an error on the FluffyCore side, it has an error field with an arbitrary data type and a constructor where this field is initialized.

PendingHandler class, which defines a handler waiting for a response and is used in KafkaProvider, RabbitMQProvider, RedisStreamsProvider classes, it stores a

timer object - a timer waiting for a function, resolve fields - as a callback for successful request completion and reject - as a callback for request completion with an error.

Message – defines the general structure of the message within the infrastructure, Message is a strict contract, thanks to which when sending, processing, receiving a request, handlers know what this message is and what to do with it. An object of type Message stores an arbitrary type req - an object that represents the request data, id - a unique message identifier in UUIDv4 format, isResponse - a flag indicating whether this message is a response to the request, resp - an arbitrary object containing the response to the request directly, isError - a flag indicating whether an error occurred while processing the message, safeMetadata - an object used inside the Pipeline to store intermediate data, this object will be cleared before the message is sent, metadata - additional information of the message that can be accessed by both the service that made the request and the service that received the response, noReply - flag that controls the mode in which the message was sent, request-reply or fire-and-forget, determines whether FluffyCore should respond to the message.

2.4 Defining algorithms for working with message brokers

Earlier we defined the interfaces and entities that the library software solution represents for unifying interaction with different message brokers. Now, we are faced with the task of directly designing algorithms for working with different message brokers and functioning of the system as a whole. For this purpose, let's make UML Activity Diagrams [17], presented in Figures 2.2, 2.3 and 2.4, where algorithms that implement FluffyCore, NATSProvider and KafkaProvider classes directly are shown. Figure 2.2 shows the UML Activity Diagram of high-level request processing, while Figures 2.3 and 2.4 show the algorithm for executing a request and waiting for a response for the NATS and Apache Kafka message brokers. The choice of these particular technologies for depiction is due to the fact that the NATS and Apache Kafka interfaces are significantly different and offer fundamentally different

methodologies for executing request-reply logic, while the logic of working with RabbitMQ and Redis Streams is relatively similar to working with the Apache Kafka broker.

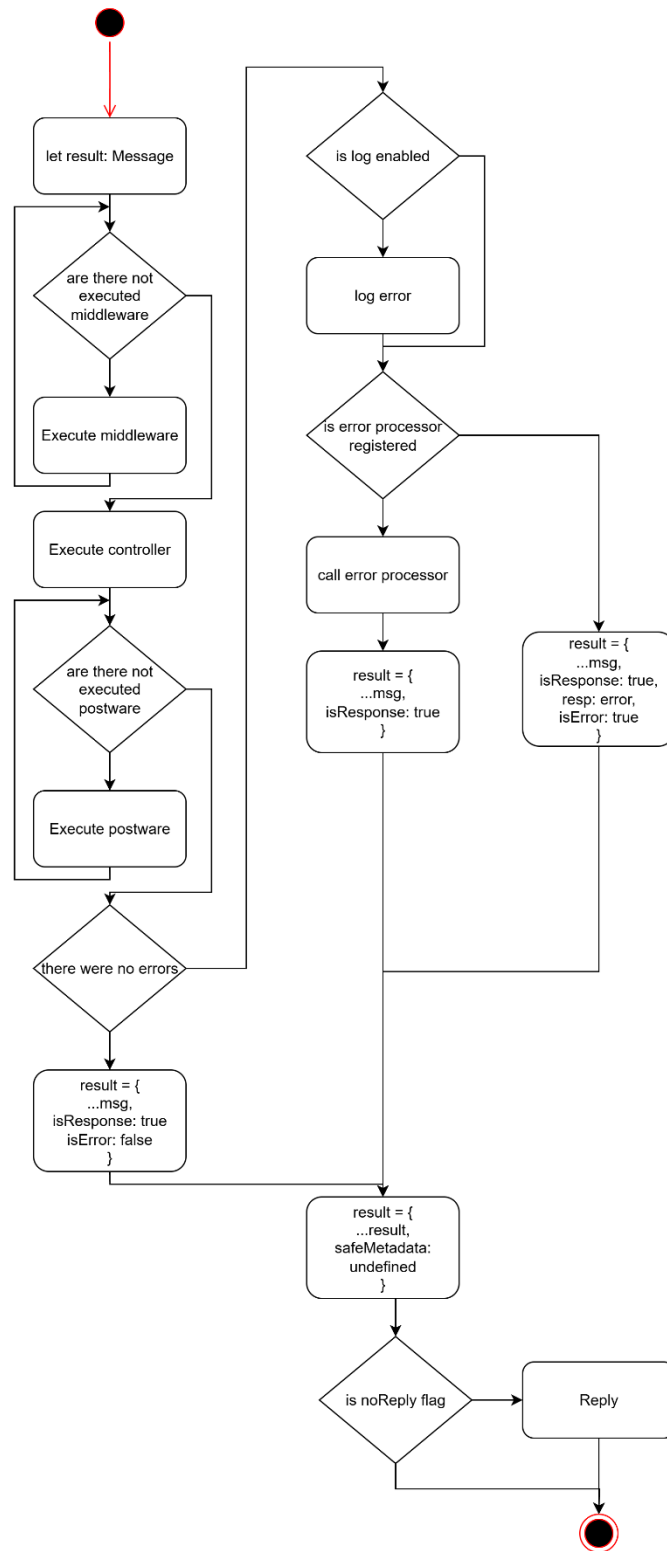


Figure 2.2 – UML Activity diagram of incoming message processing lifecycle

Figure 2.2 shows the UML Activity Diagram of the incoming message processing lifecycle in the FluffyCore class (which is responsible for the core business logic of the library and works on top of the providers).

At the very beginning, the result: Message entity is created. This is the message object that the provider received from the broker and passed to the service. After initialization, the system checks if there are any unprocessed middleware in the chain. Middleware in this architecture act as pre-filters and modifiers: they can change input data, add metadata, check access rights, validate payload. If there are such middleware, the system executes each one in turn, and only after they have passed completely does it move on.

Control is then passed to the controller. The controller is the business logic for which the service is written. It is here that a message turns into a target action: for example, a query in the database, a call to a third-party API or calculation of something.

After the controller, all added postware is executed similarly to middleware. This is a mirror part of middleware, only it is executed after the main action. Here you can log the result, convert output data, add service information or perform side effects that should not interfere with the main logic.

If everything was successful at this stage and no errors occurred, the result is formed as a message object with the attributes `isResponse: true` and `isError: false`. That is, it is a correct response that will be returned to the calling party through the same provider.

If an error occurs during the execution of middleware, controller or postware, the scheme goes to the right, to the error handling block. It first checks whether logging is enabled. If yes, the error is logged. Next, the system checks whether the error processor is registered. This is a special handler that allows you to centrally turn an exception into a correct response. If there is one, the error processor is called and the result is generated containing the original message and the `isResponse: true` sign. If there is no error processor, the service itself collects the response with the fields

isResponse: true, isError: true and puts the error object in the response. In both cases, the calling party will receive a correctly formatted response, and the exchange protocol will not be broken.

Regardless of whether there was a success or an error, the system removes the safeMetadata from the response at the last step so that only safe information is left outside. After that, the noReply flag is checked. If it is set, the system realizes that the response does not need to be sent, and the algorithm is completed. If the flag is absent, the result is sent back to the broker through the corresponding provider.

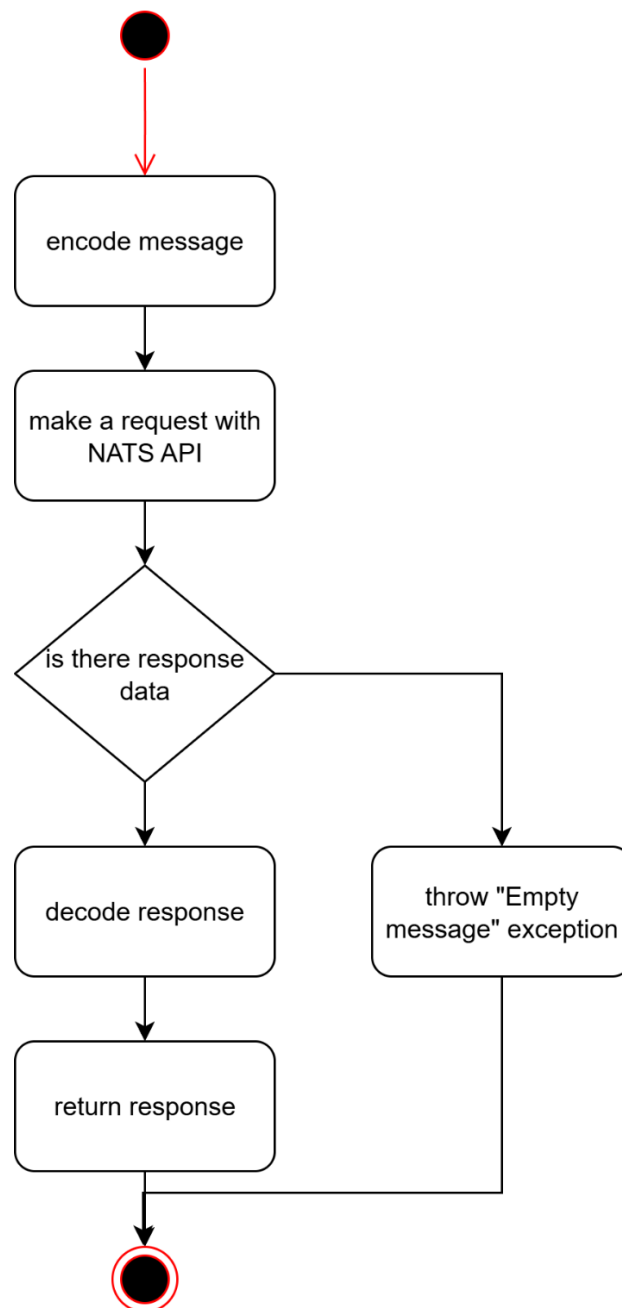


Figure 2.3 – UML Activity diagram of making a request with NATSProvider

Figure 2.3 shows in the form of a UML Activity Diagram an algorithm for executing a request and receiving a response using NATS technology and its corresponding provider (NATSPProvider). As we can see in this figure, the algorithm for executing such a request is quite simple, since NATS out of the box knows how to handle request-response logic.

In this activity diagram we can see that the message is initially serialized using the NATS serializer, then the request is executed using the native API, the NATS message broker takes care of the response topic and the correlation id of the message. The method that is called here, namely request will wait for the response to be received and will return the received response in the form of serialized data.

The next step is to check if there is data in the response, if there is not, the system will throw an exception, this behavior can occur when, for example, the request timeout has expired.

The useful response data is further deserialized by a complete deserializer and the method returns the useful data to the calling client.

Unlike message brokers with native request-reply support (e.g., ActiveMQ or RabbitMQ), the Apache Kafka message broker was originally designed around the producer-consumer pattern. Consequently, Kafka has no built-in mechanisms for message correlation – that is, it does not automatically link a request message to its corresponding reply. Implementing the request-reply pattern on top of Kafka is therefore not trivial; it requires manual correlation IDs, separate reply topics, or stateful consumers. However, such an implementation becomes relevant and practical mainly in two scenarios: first, when Kafka is already the central messaging backbone of the project and introducing another broker would add unnecessary complexity; second, when the application demands the high durability, fault tolerance, and exactly-once semantics that Kafka provides out of the box. In other words, while Kafka is not a natural fit for synchronous request-reply interactions, its reliability guarantees can justify the extra engineering effort when the broker is already present in the stack or when message loss is unacceptable.

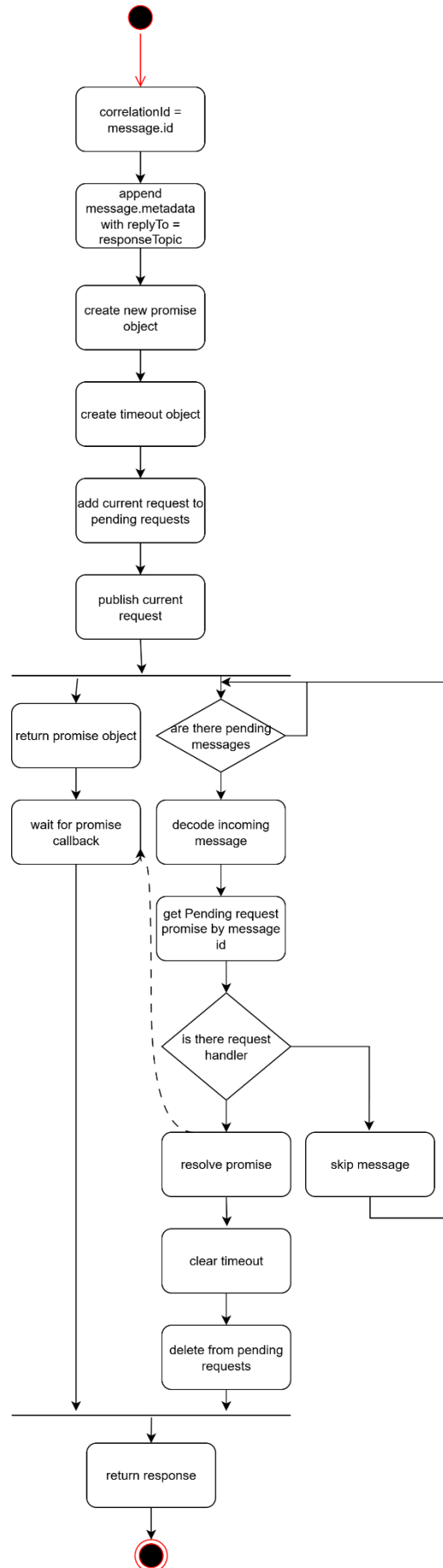


Figure 2.4 – UML Activity diagram of making a request with KafkaProvider

Figure 2.4 shows a UML Activity Diagram of a request-reply algorithm using the Kafka message broker. Apache Kafka does not have native tools for request-reply mode, does not keep track of reply tops and does not take into account correlationId of messages, therefore, the implementation of this logic falls on the developer's shoulders.

The flow starts with the preparation of the correlation id: the message.id generated as UUIDv4 is read from the source message, which then acts as the only key for matching the request and response. A replyTo is added to the message metadata, which is the name of the topic where the executor should send the reply. This is important for Kafka because the “return address” is not a built-in mechanism, it has to be passed to the application.

Then a “dangling” promise is created: a Promise is formed, whose resolve/reject are stored in the internal registry of expected requests together with the correlationId. In parallel, a timeout object is created, which is guaranteed to end the Promise with an error after a specified interval and perform a cleanup if no response is received. After that, the current request is registered in pendingRequests. Only after registration is the request published to Kafka: the message is sent to the “command” topic with the headers correlationId and replyTo. The method call immediately returns the created Promise to the calling code; from that point on, the caller can await the result, and the computation actually continues in the background through the incoming message handler.

The right branch of the diagram shows a continuous cycle of consuming responses from a responseTopic. The consumer retrieves messages, decodes the payload, and reads the correlationId from the headers. It then performs a dot search for the corresponding “pending” request in the registry. If an entry is found, it is assumed to be a valid response to a previously sent request; if there is no matching handler, the message is ignored as irrelevant for this node. In case of a match, resolve the stored promise is called, after which the timer is cleared, the record is removed from the registry, and the response is returned up the chain - to the same await hanging in the calling code. The dotted arc on the diagram reflects the cause-and-

effect relationship: resolving a promise in the consumer automatically “wakes up” the awaiting context and allows you to return the result from the public method. If the message is missed, the loop simply continues processing the next incoming data without affecting the waiting state.

Promise mechanics are critical here and rely entirely on the Node.js event loop model. The public method synchronously creates and gives Promise without blocking the thread; when a response comes from Kafka, the handler calls the stored resolve. This places the execution of the continuation in the microtask queue, and the nearest event loop tick delivers the result to the context where await was executed. This design is resistant to competition between multiple concurrently active requests, because matching is done by correlationId rather than by message delivery order, so responses can arrive in any order without violating correctness. Timeout eliminates “perpetual waiting” in the face of at least once-delivery Kafka and network failures, ensuring that resources are released even if a response is lost. Clearing the timer and deleting a registry entry after resolving eliminates memory leaks and prevents re-resolving the same promise in the event of possible duplicate delivery.

2.5 Development of library components and features of integration into the project

During the development of the library's direct program code, a number of additional TypeScript dependencies were used, and the following npm packages were applied:

- uuid is a package for generating unique message identifiers of the UUIDv4 standard. This library provides cryptographically strong identifiers, which is critical for correlating requests and responses in asynchronous request-reply communication between microservices. The use of UUIDv4 avoids identifier collisions in a distributed system and ensures global uniqueness of each message.
- nats is a native client library for accessing the NATS message broker API. The package provides a high-level API for interacting with the NATS

server, supports all major communication patterns (pub-sub, request-reply), and provides mechanisms for connection management, timeouts, and error handling. The library is optimized for high load and minimal latency.

- `kafkajs` is a modern client for Apache Kafka written in pure JavaScript with full TypeScript support. The library provides an intuitive API for producers and consumers, supports transactions, exactly-once delivery and consumer group management. `kafkajs` was chosen for its active development, detailed documentation, and lack of native dependencies.

- `amqplib` is a popular client library for RabbitMQ that implements the AMQP 0-9-1 protocol. The package provides low-level control over broker interactions, including queue declaration, exchange and binding configuration, and quality of service (QoS) management. The library supports all major RabbitMQ patterns, including RPC for request-reply.

- `ioredis` is a full-featured Redis client for Node.js with support for clustering, Sentinel, and threads. The library provides a convenient API for working with Redis Streams, including managing consumer groups, reading messages by range, and waiting for new entries. `ioredis` was chosen for its high performance, `promis` support, and advanced connection monitoring capabilities.

These packages were chosen due to their reliability, active community, full functionality and high-quality TypeScript support, which meets the requirements for developing a library for microservice architecture.

As mentioned in the class diagram, architecturally, the library consists of a high-level module, `FluffyCore`, and a set of low-level adapters (providers) that directly implement the behavior management of message brokers and unify the interfaces for interacting with the high-level core.

The high-level `FluffyCore` is an object that developers will use directly to interact with the library. One of its most important functions is registering roots, each root is a `Pipeline` type object that stores middlewares, controller, postware and a topic to which it can be accessed. The call to register a root must be made before calling the start method. During registration, `FluffyCore` simply writes the `Pipeline` array. Next, the start method does all the logic necessary to initialize the provider correctly. In the start method, for each `Pipeline`, the subscription to the topic specified during

registration is called, an anonymous function is created to handle each Pipeline, and within this anonymous function, the algorithm shown in Figure 2.2 is executed. At the very end of the start method, the ready method is called on top of the provider, which tells the provider that the roam registration is complete and it can start processing messages. FluffyCore also provides methods fireAndForget to send a message without waiting for a response; makeRequest and makeRequestSimplify to send a message with a pending response. Figure 2.5 shows the program code that implements the above logic.

Each provider, to work with FluffyCore object, must implement the IProvider interface. This interface provides common internal methods such as connect to initialize the connection to the broker, ready to switch to message processing mode, disconnect to disconnect from the message broker, setEnableLog to modify the logging options, publish to send a message to a topic, subscribe to subscribe to a topic and receive messages by the corresponding handler, reply to reply to a message, and makeRequest to execute a request to a specific topic.

Basically, a class that implements the IProvider interface has to implement an OOP adapter pattern. The Adapter pattern is a structural design pattern that allows objects with incompatible interfaces to collaborate. It acts as a wrapper that converts the interface of a class into another interface expected by the client. In this context, the class implementing IProvider serves as the adapter: it receives calls from the higher-level fluffy-core class using the IProvider interface, translates or delegates those calls to an adaptee, and returns the result back in the format expected by the client. This way, the adapter hides any incompatibilities – such as mismatched method names, behaviors, parameter types, or data formats – without modifying the adaptee's original code.

Figure 2.6 shows the program code of the IProvider interface.

```

async start() {
    this.provider.setEnableLog(this.enableLog)
    await this.provider.connect();

    for (const pipeline of this.pipelines) {
        this.provider.subscribe(pipeline.topic, async (msg: Message, m:
any) => {
            let result: Message;
            try {
                for (const middleware of pipeline.middlewares) {
                    await middleware(msg)
                }
                await pipeline.controller(msg)
                for (const postware of pipeline.postware) {
                    await postware(msg)
                }
                result = {
                    ...msg,
                    isResponse: true,
                    isError: msg.isError === undefined ? false :
msg.isError,
                }
            } catch (e) {
                if (this.enableLog) console.error(e)
                if (this.errorProcessor) {
                    await this.errorProcessor({ msg, e })
                    result = {
                        ...msg,
                        isResponse: true,
                    }
                } else {
                    if (this.enableLog) console.warn('Consider using custom
error controller, providing full error to a client can be dangerous to data
privacy')

                    result = {
                        ...msg,
                        isResponse: true,
                        resp: JSON.stringify(e),
                        isError: true
                    }
                }
            }
            result = {
                ...result, safeMetadata: undefined
            }
            if (this.enableLog) console.log(`${new Date().toISOString()}
sending reply`, result)
            if (!result.noReply) await this.provider.reply({ m, message:
result, topic: pipeline.topic })
        });
    }

    await this.provider.ready();
}

```

Figure 2.5 – Program code of start() method of FluffyCore class

```

export interface IProvider {
  connect(): Promise<void>;
  ready(): Promise<void>;
  disconnect(): Promise<void>;
  setEnableLog(enable: boolean): void

  publish(subject: string, message: Message): Promise<void>;

  subscribe(
    subject: string,
    handler: (msg: Message, m: any) => Promise<void> | void
  ): Promise<void>;

  reply({
    message,
    m,
    topic,
      } : {
        message: Message,
        m?: any, // for nats
        topic?: string
      }
  ): Promise<void>;

  makeRequest(subject: string, message: Message): Promise<Message>;
}

```

Figure 2.6 – Program code of IProvider interface

Let's see how a developer who decides to use the described library will implement his code. Figure 2.7 shows an example of creating a simple endpoint. First, the developer creates an instance of type IProvider, in the case of this example, NATSProvider, then creates a high-level FluffyCore developer instance with the constructor argument, the created provider. Next, the registerRoute method is called on FluffyCore, here the developer can pass a topic to the controller that executes the main business logic, middlewares and postware, in the case of the example in Figure 2.7, the developer creates an endpoint on the test-topic, where the controller places an object {ok: true} in the response field. Middleware, in this case writes an object to the safeMetadata field, with a field messageArrived, which puts the time when the request hits the pipelines. Postware is executed one by one, the first postware adds the messageReadyToRespond field to the safeMetadata field and puts the current time into it (after the controller), the second postware supplements the response field with the timeToProcess field, which indicates how long the request has been in the

Pipeline. Next, the developer sets the `errorProcessor` as an asynchronous anonymous function that somehow handles the error and modifies the endpoint response. Finally, the `start` method is called, which registers that the service is ready to accept messages.

```
const serverProvider = new NATSProvider('localhost:4222')
const server = new FluffyCore(serverProvider)
server.registerRoute({
  topic: 'test-topic',
  controller:
    async (msg: Message) => {
      console.log("Controller got message", msg)
      msg.resp = {
        ok: true
      }
    },
  middlewares: [
    async (msg: Message) => {
      msg.safeMetadata = {
        messageArrived: new Date(),
      }
    },
  ],
  postware: [
    async (msg: Message) => {
      msg.safeMetadata = {
        ...msg.safeMetadata,
        messageReadyToRespond: new Date(),
      }
    },
    async (msg: Message) => {
      msg.resp = {
        ...msg.resp,
        timeToProcess:
          msg.safeMetadata.messageReadyToRespond.getTime() -
          msg.safeMetadata.messageArrived.getTime()
      }
    },
  ],
})
server.setErrorProcessor(async ({ msg, e }: { msg: Message, e: any }) => {
  msg.resp = e
  msg.isError = true;
})
await server.start()
```

Figure 2.7 – Example of registering a route with fluffy-core library

Next, let's send a request to this endpoint, for this purpose, similarly to the previous example, let's create a provider and `FluffyCore` instance. Since we don't plan

to register roots in this example, let's immediately execute the start method. Next, to execute the request, we execute the makeRequest method on the FluffyCore entity, with arguments topic: 'test-topic' and data: {hello: 'world'}. When the request is executed, we will see a log with the service response. An example of query execution is shown in Figure 2.8.

```
const clientProvider = new NATSProvider('localhost:4222')
const client = new FluffyCore(clientProvider)
await client.start()

const response = await client.makeRequest({
  topic: 'test-topic',
  data: {
    hello: 'world'
  }
})

console.log("responder returned response", response)
```

Figure 2.8 – Example making a request with fluffy-core library

As we can see from the examples of interaction with the library, using it for the end developer is extremely simple and intuitive, which should greatly facilitate the development of the final product.

To make the developed software product a library in its usual sense, let's add the developed solution to the npm (package manager for JavaScript that serves as the default registry for Node.js and front-end projects, hosting over two million reusable open-source packages. It not only handles dependency resolution and version management (following semantic versioning) but also provides tools for publishing, sharing, and discovering libraries, along with lockfile support for reproducible installs) repository, for this purpose we will add a workflow to the github repository of the library, which, when a new version of the library enters the main branch, will publish the corresponding version in npm, such a workflow is shown in Figure 2.9.

```

name: Publish to npm

on:
  push:
    branches:
      - main

jobs:
  publish:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3

      - uses: actions/setup-node@v3
        with:
          node-version: 20
          registry-url: 'https://registry.npmjs.org/'

      - run: npm install
      - run: npm run build

      - run: npm publish --access public
        env:
          NODE_AUTH_TOKEN: ${ secrets.NPM_TOKEN }

```

Figure 2.9 – publish.yml

This workflow (shown in Figure 2.9) automates the process of publishing the library to the npm registry. It is activated each time the main branch is pushed to the main branch, ensuring that new versions of the package are delivered continuously. The process is run on the latest version of Ubuntu in an isolated environment, ensuring a reproducible and reliable build.

The workflow starts by retrieving the source code from the repository using `actions/checkout`. The Node.js version 20 runtime environment is then configured via action `actions/setup-node`, which also configures access to the npm registry. An important point is to install the project dependencies with the `npm install` command, which ensures that all the necessary packages are available for later builds.

The key step is to build the project with the `npm run build` command, which transforms the source TypeScript code into a distribution-ready JavaScript package. The final step is to publish the package to the npm registry using the `npm publish` command with the `--access public` flag to make the package publicly available. For

authentication, the NPM_TOKEN token is used, which is securely stored in the repository secrets and provides the necessary permissions for publishing.

The entire process enforces semantic versioning, as the package version is automatically determined from the package.json file.

As a result of this workflow, we can see that our library - fluffy-core is published in the npm registry and is available for installation in the project. A screenshot of the npm package page is shown in Figure 2.10.

Search packages

fluffy-core TS

0.0.49 • Public • Published a day ago

[Readme](#) [Code](#) [Beta](#) [10 Dependencies](#) [0 Dependents](#) [46 Versions](#) [Settings](#)

Fluffy core

Fluffy core is a TS library for simplifying communication between microservices in microservice architecture. It supports multiple providers such as NATS, Kafka, RabbitMQ and RedisStreams and provides unified interface for fast integration in a project.

[GitHub of a project](#)

THIS PROJECT IS STILL UNDER DEVELOPMENT!

Please be careful in case of using it in production!

Currently providers tested in request-response mode:

- NATS
- Kafka
- RabbitMQ
- RedisStreams

Please use it carefully and report any issues you find. I'll do my best to fix them. But some things may not work as expected at current stage.

How to use

Install

```
> npm i fluffy-core
```

Weekly Downloads

653

Version	License
0.0.49	ISC

Unpacked Size	Total Files
90.9 kB	46

Last publish
a day ago

Collaborators

[Try on RunKit](#)

[Report malware](#)

Figure 2.10 – Library page in the npm repository

3 DESIGN AND DEVELOPMENT OF AN EXPERIMENTAL INFORMATION SYSTEM

The objective of this section is to prepare the information system for application testing of the developed library. This section of the qualification work involves the design and development of an information system, followed by the integration of the developed messaging library into a microservice architecture. Since individual components of the information system are only part of the work presented for demonstrating and testing a previously developed product, their implementation can be significantly simplified due to the fact that their correct implementation will not impact the testing of the final product. Essentially, this section describes how the final product developer can use the developed library in example of developing the information system described below.

3.1 Description of the subject area

Internet technologies have become an essential aspect of contemporary living, facilitating work, education, information access, communication, and entertainment. Social networks, in particular, serve as widespread platforms for interaction and as sources of timely and engaging content. The information system under development is centered on the entertainment sector, allowing users to browse news and materials aligned with their interests. It supports seamless user interaction through posts and comments, with content relevance determined by a combination of post rating and publication date—a crucial feature for maintaining an engaging user experience.

As unifying digital environments, social networks foster communities among people with diverse interests and perspectives. The relevance of this project is underscored by the fundamental human need for communication, a need further amplified by the increase in remote work and study during the COVID-19 pandemic. The proposed system introduces several features not commonly found in existing platforms, such as the ability to create discussion threads from comments, a

bidirectional rating system for both posts and comments, and a shared feed that highlights the most interesting content as determined by users. By offering these capabilities, the project serves not only as a technically relevant endeavor but also as a means to support social connection during challenging times, helping to build community and bring people together across the globe.

The primary target audience consists of young adults aged 18–35, who are frequent computer users and prefer online communication. Content relevance will be dynamically managed through the bidirectional rating mechanism. This group also includes content creators whose contributions are consumed and evaluated by other users.

In essence, a social network is an online platform that enables users to create profiles, share information, and interact with others, often through interest-based groups or communities. This section addresses the development of MVP components for such a platform.

The core business process in this system is post creation. Users can make posts that are visible to others, who may rate or comment on them. The commenting functionality supports nested replies, and comments themselves can also be rated. A centralized feed employs a specialized index that combines rating and time of publication to ensure the most relevant posts are prioritized. This method of content curation helps surface interesting and timely material to every user, regardless of geography, age, gender, or other characteristics. Additionally, the rating of a post or comment directly affects the reputation of its author, whose overall rating is adjusted accordingly.

3.2 Development of functional requirements for the web-application

The functional requirements for the social network information system in this qualification work are developed through functional modeling using IDEF0 standard [18] diagrams. This approach helps to determine how various factors influence the

system's business processes, which involve two user roles: the authorized user and the guest.

DEF0 is a functional modeling methodology that helps analysts and stakeholders understand, document, and improve complex processes by answering the question “what does the system do?”. Each function is represented as a box with four arrow types, which forces a disciplined analysis of resources, constraints, and results. The main practical benefit of IDEF0 lies in its hierarchical decomposition: a large, complex process is broken down into small, manageable sub-functions (3-6 per diagram), revealing hidden dependencies, redundant steps, and missing controls. This structured approach solves common problems such as unclear responsibilities, undocumented workflows, and communication gaps between business and technical teams. By requiring a defined purpose and viewpoint, IDEF0 ensures that models stay focused on solving the right problems – whether it is reengineering a business process, documenting legacy systems, or identifying bottlenecks before automation.

The process begins with the construction of a top-level context diagram, illustrated in Figure 3.1, which depicts the interaction between the main business process and its external influences. This context diagram is subsequently decomposed into more detailed diagrams, shown in Figures 3.2 through 3.5, to provide a deeper understanding of the constituent processes. A node tree diagram, which provides a hierarchical overview, is presented in Figure 3.6.

These diagrams serve as a visual tool to observe the interactions within the system's processes. They form a foundational basis for specifying the nature of business processes, defining precise requirements, and identifying key aspects and mechanisms of the software product during the system analysis phase. Each successive level of decomposition enables a more detailed analysis of the processes, facilitating the formulation of specific criteria to verify that the developed module meets the stated requirements.

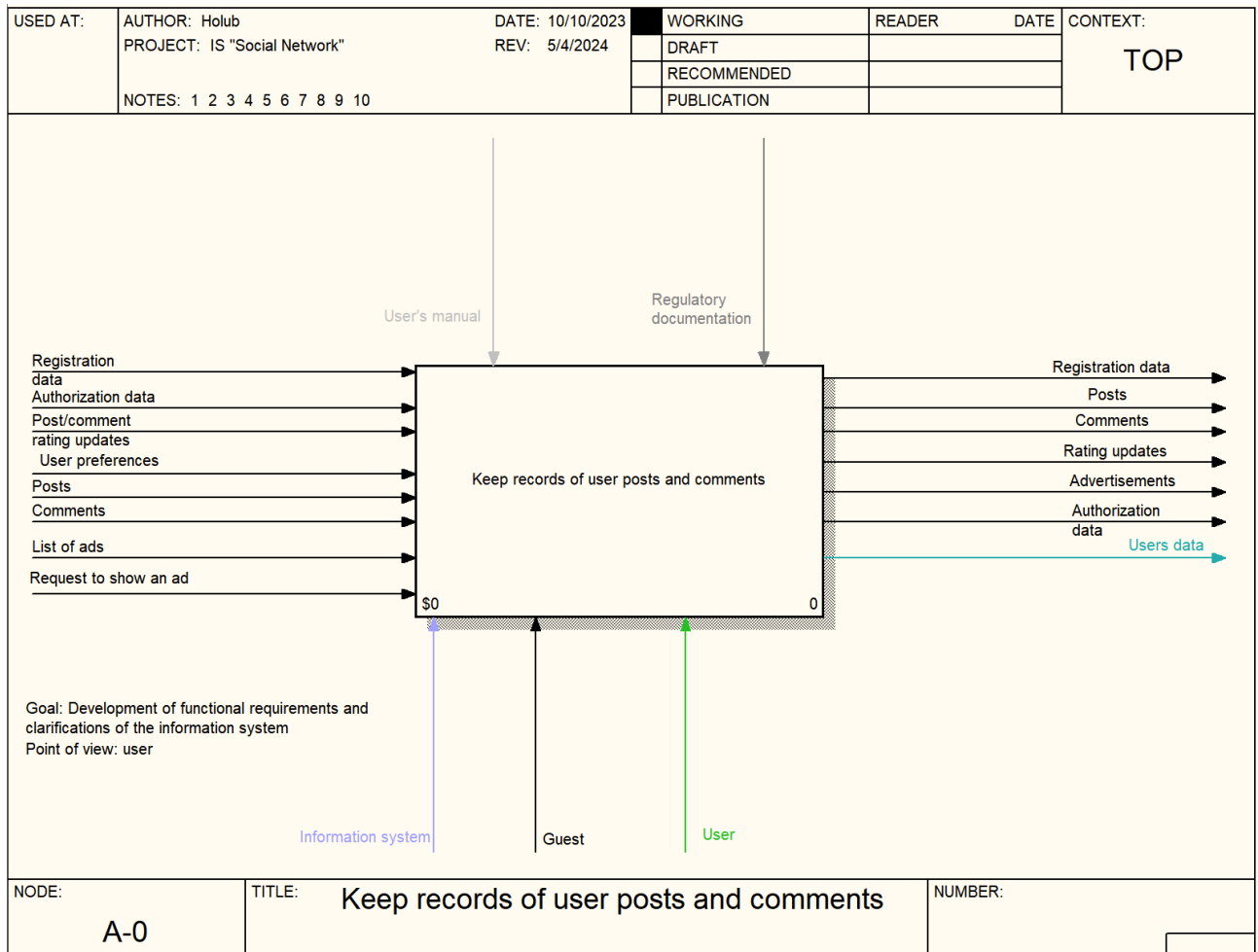


Figure 3.1 – Context diagram of the functional model

The provided diagram outlines the functional boundaries of the information system and details the inputs and outputs of its core business process. Input data includes registration and authorization details, rating updates for posts and comments, user preferences and settings, user-generated posts and comments, as well as a list of advertisements and requests to display targeted ads.

The process is governed by control elements, specifically the user manual and regulatory documentation, which encompasses relevant laws, government acts, and internal company policies. The necessary resources for the process are the information system itself, along with its guest and authorized users.

The resulting outputs from the process consist of processed registration and authorization data, published posts and comments, updated rating information, served advertisements, and modified user data.

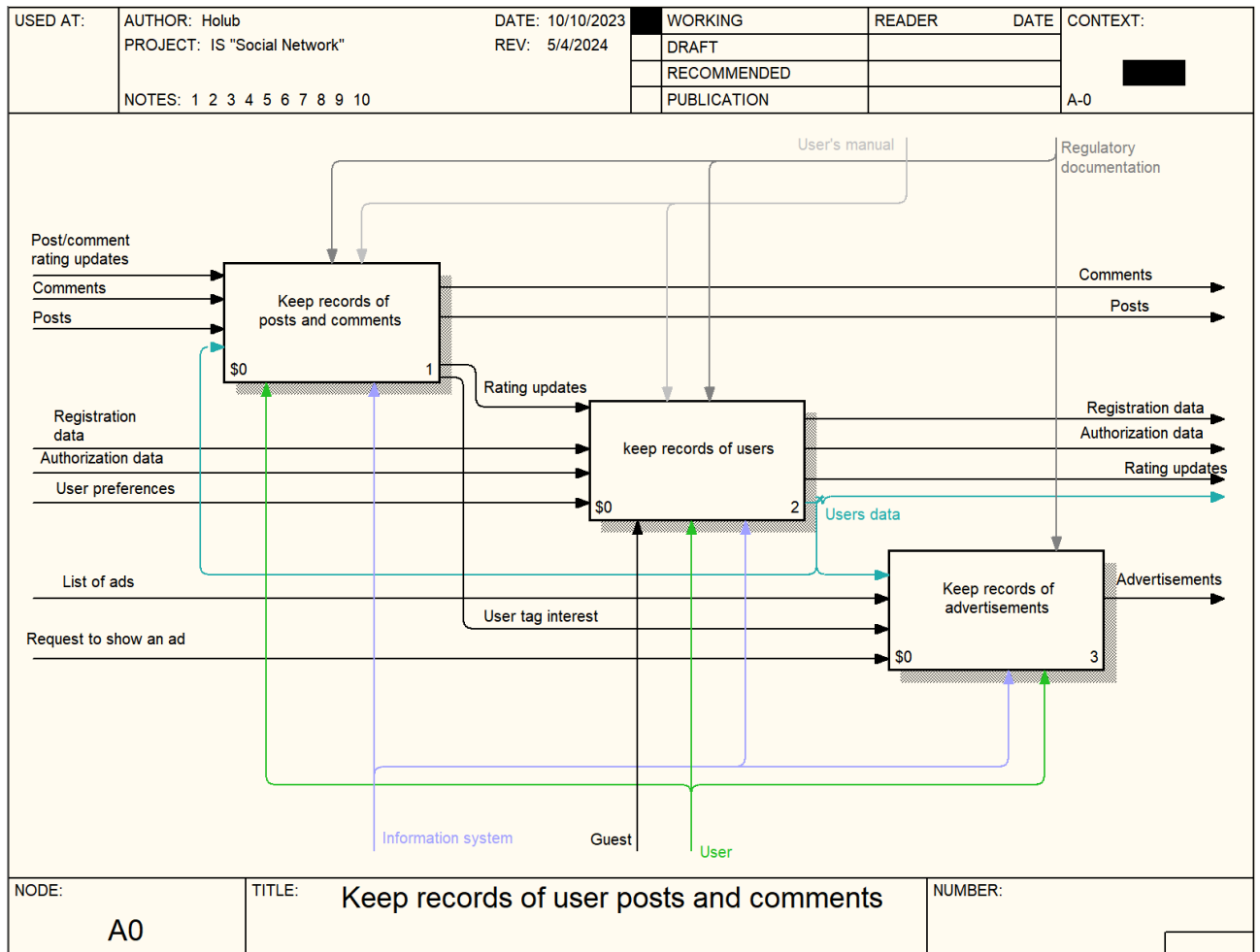


Figure 3.2 – Decomposition diagram showing system functions

The diagram in Figure 3.2 illustrates the relationships between the functions and processes within the information system, identifying several core functions. Including:

- Keep records of posts and comments;
- Keep records of users;
- Keep records of advertisements.

As depicted, a user with "Guest" status is permitted access only to authorization and registration functions. Once logged in, the system uses the authenticated user's data to enable primary business functions, such as managing post and comment records or displaying targeted advertisements. It is noteworthy that the functional block for managing advertisements does not require a user manual among

its controls, as all related operations are performed automatically by the information system without user intervention.

These three functional blocks form the foundation of the system's main business processes: “Authorization and profile management”, “Content sharing within the social network”, and “Receiving advertisements”.

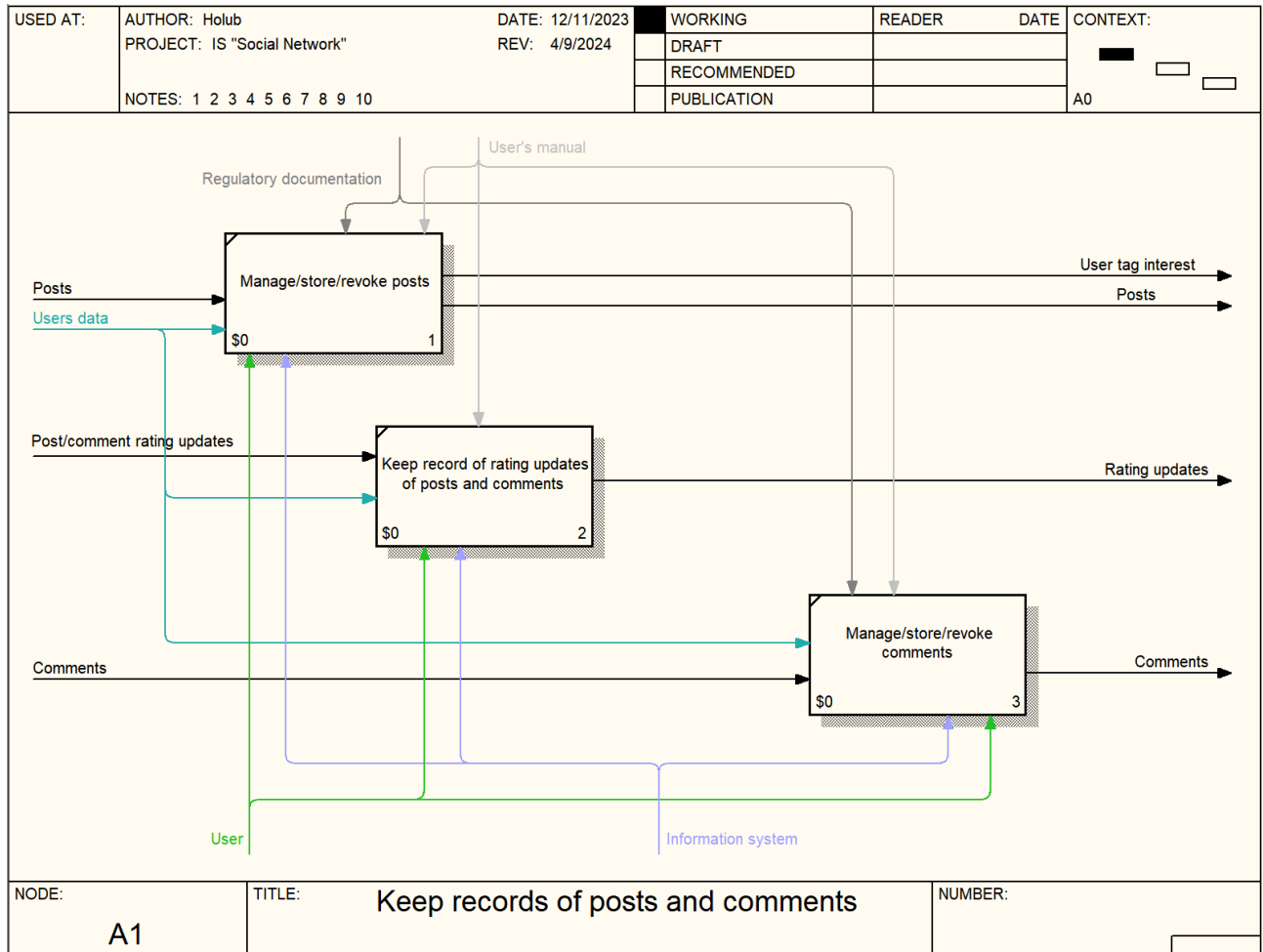


Figure 3.3 – Decomposition diagram of the "Keep records of posts and comments" process

The presented diagram includes subprocesses such as: Manage/store/revoke posts; keep record of rating updates for posts and comments; and manage/store/revoke comments. Each of these subprocesses features corresponding input and output data flows.

The diagram includes an arrow labeled "post/comment rating updates." This represents the process where a change in a post or comment's rating triggers a corresponding update to the user's rating. However, since the user's rating is managed

within the "User Management" function block, this results in an output arrow from this process, signaling a rating update notification.

Additionally, an output arrow labeled "user tag interest" is shown. This serves as a notification to the advertising functional block indicating that a user has shown interest in a post containing specific tags.

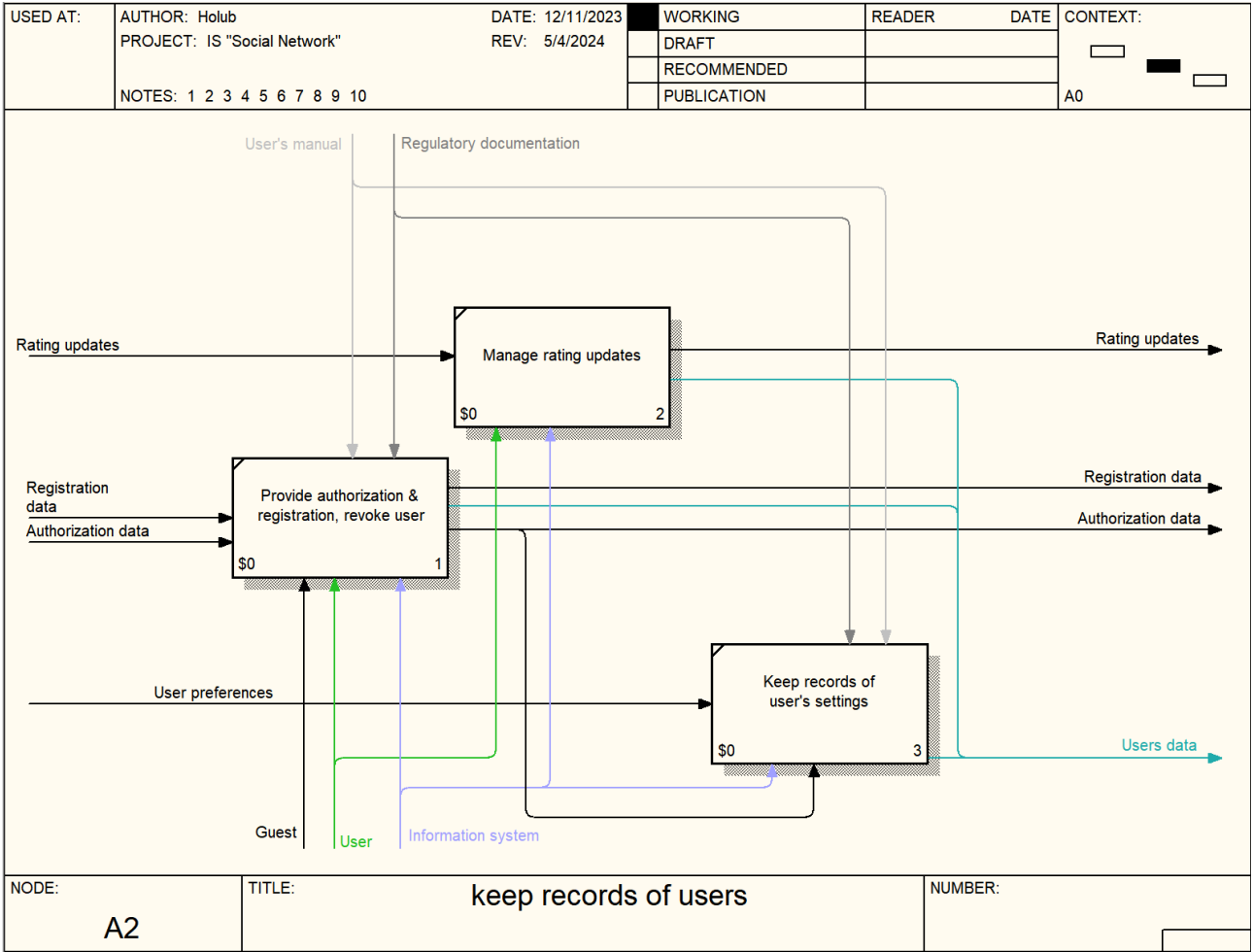


Figure 3.4 – Decomposition diagram of the "Keep records of users" process

The displayed diagram includes the following subprocesses: manage rating updates, provide authorization & registration / revoke user, and keep records of user's settings.

The diagram illustrates that any activity affecting a user's profile results in an update to the user's data. When a post or comment belonging to a user receives a rating, the user's own rating is correspondingly incremented or decremented. An output arrow labeled "Users data" reflects this updated information, which is

subsequently used during the creation of posts and comments, as well as in the selection of advertisements presented to the user.

It is important to note that the “manage rating updates” subprocess operates internally and is not initiated directly by the user. Furthermore, a user assigned the “Guest” role is permitted access solely to the authorization and registration process. Here, registration data conveys the status of a registration request, while authorization data reflects the status of a login request. Upon successful authorization, this output includes the user’s authorization token.

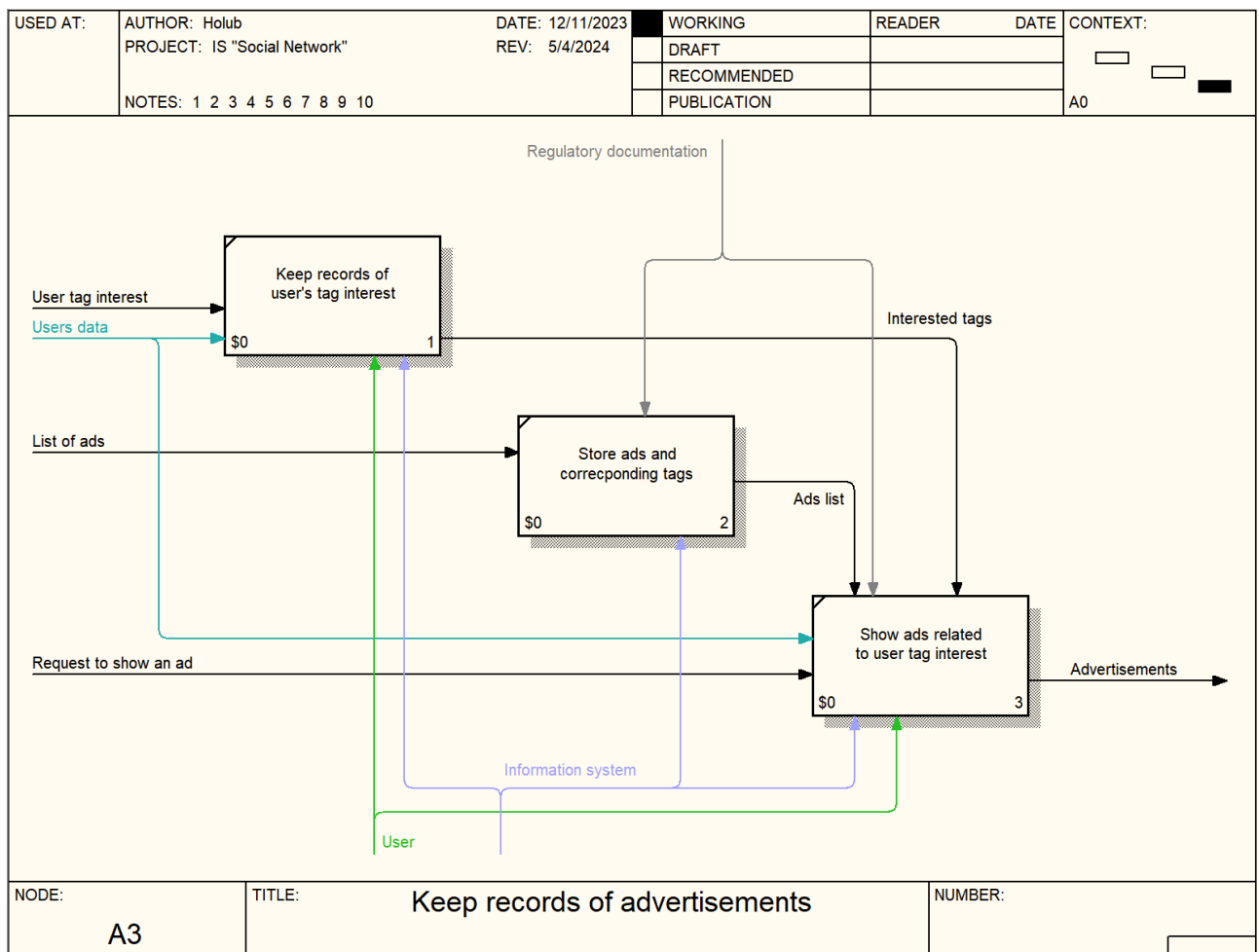


Figure 3.5 – Decomposition diagram of the "Keep records of advertisements" process

The displayed diagram includes the following subprocesses: keep records of user’s tag interest, store ads and corresponding tags, and show ads related to user tag interest.

The diagram illustrates that notifications regarding a user's interest in specific tags are collected and compiled. A stored list of advertisements is maintained alongside their appropriate tags. When a request to display an advertisement is received, a relevant ad is selected from the available inventory based on tags that match the user's interests.

It is important to note that the “keep records of user’s tag interest” subprocess is not controlled by the user and is performed automatically by the information system itself.

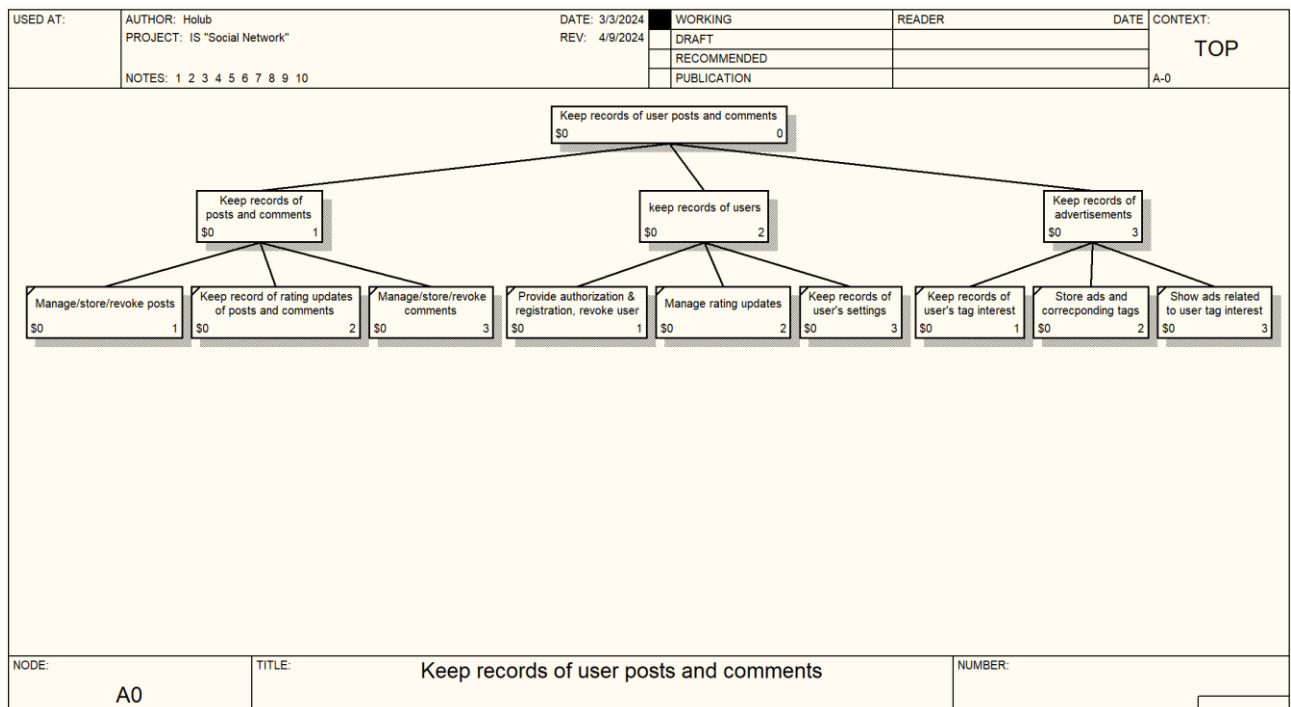


Figure 3.6 – Node tree diagram

As illustrated, the node tree diagram reflects the business process decomposition logic previously established. The primary element remains "keep records of user posts and comments", which encompasses the sub-processes "keep records of posts and comments", "keep records of users", and "keep records of advertisements".

The function "keep records of posts and comments" is further decomposed into the sub-processes "Manage/store/revoke posts", "Manage/store/revoke comments", and "keep record of rating updates of posts and comments".

The "keep records of users" function includes the sub-processes "provide authorization & registration / revoke user", "manage rating updates", and "keep records of user's settings".

Finally, the "keep records of advertisements" function consists of the sub-processes "keep records of user's tag interest", "store ads and corresponding tags", and "show ads related to user tag interest".

In the analysis and formulation of requirements for the information system under development, the creation of a data flow diagram [19] plays a significant role. This type of diagram, rendered in its corresponding notation, provides a broad illustration of data exchange between processes, data stores, and external entities. It serves to visualize the system's tasks and the objectives set for the information system, effectively mapping the processes involved in achieving those goals.

A context diagram of these data flows for the developed social network information system is presented in Figure 3.7.

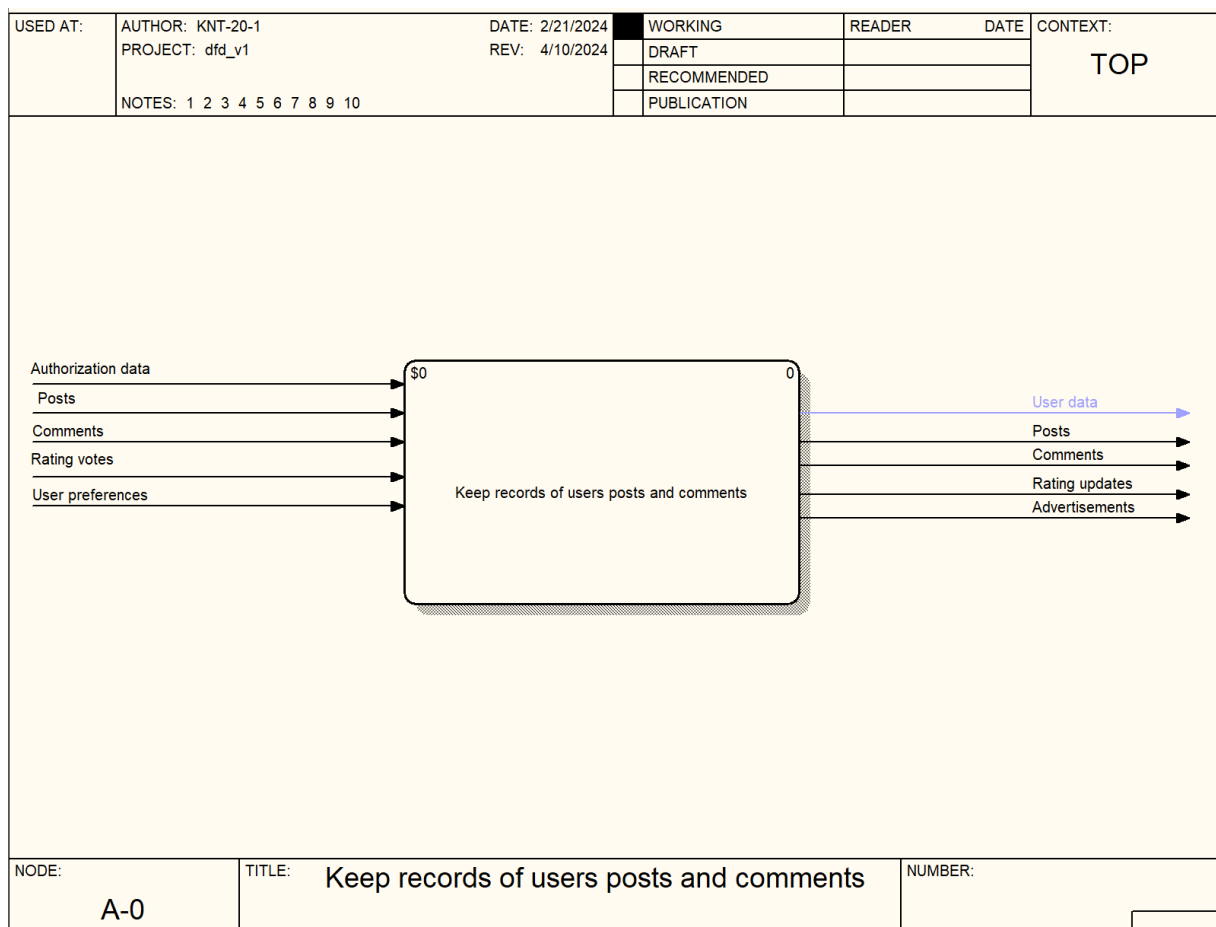


Figure 3.7 – Context diagram of the data flow model

Similar to the functional model in IDEF0 notation, the data flow diagram requires decomposition to examine the processes within the system in greater detail. The decomposition of the contextual data flow diagram is provided in Figure 3.8.

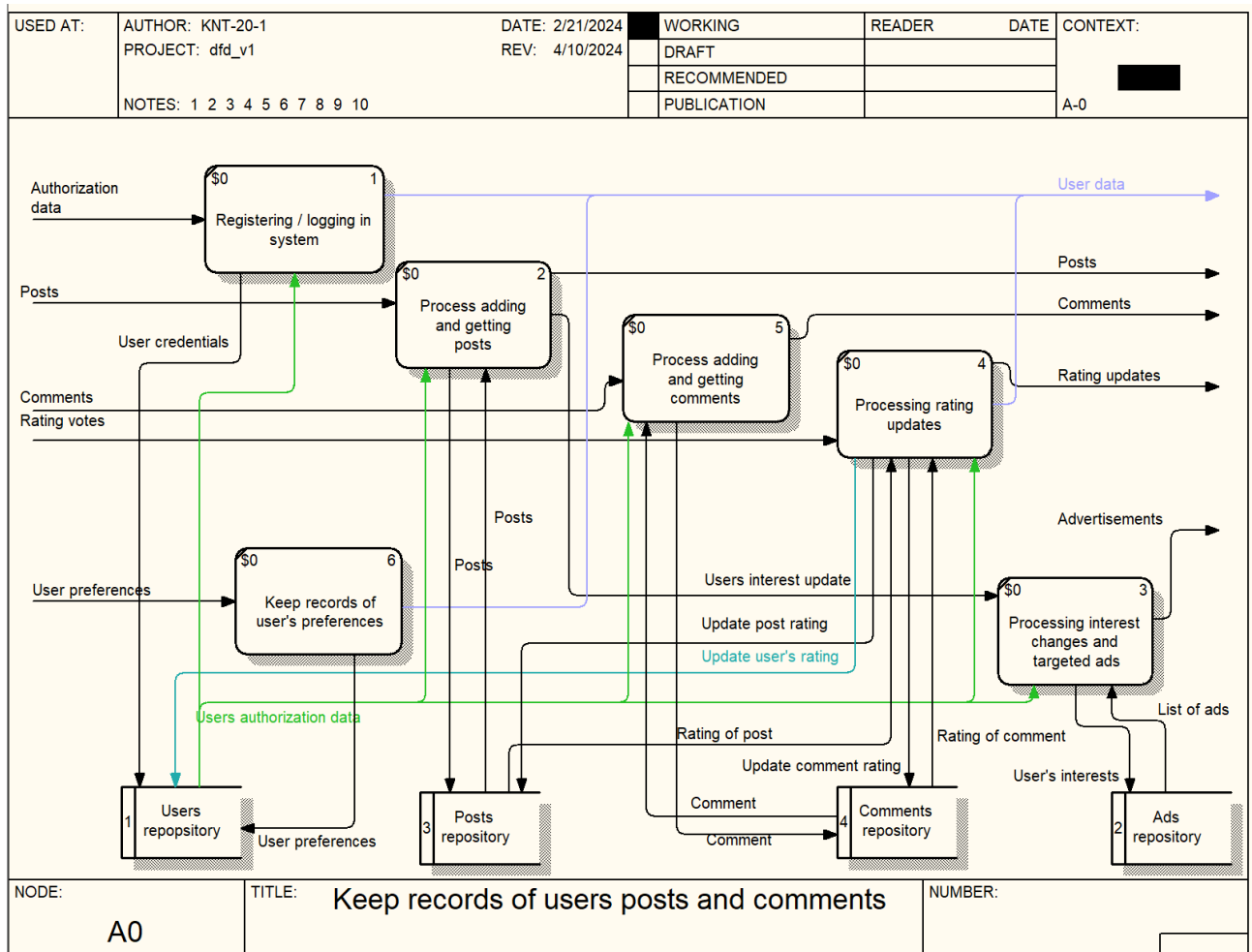


Figure 3.8 – Decomposition diagram of the data flow model

The decomposition of the context diagram includes several key processes: "registering / logging in system", "process adding and getting posts", "process adding and getting comments", "processing rating updates", "keep records of user's preferences", and "processing interest changes and targeted ads". The diagram also features several data repositories, including "users repository", "posts repository", "comments repository", and "ads repository".

The "registering / logging in system" function accepts the user's authentication data as input and returns a user data segment as output. The authorization data

retrieved from the "users repository" is subsequently used by all other functions to identify the user on whose behalf an operation is performed.

The "process adding and getting posts" function manages posts within the system. Its inputs and outputs are posts, and it stores all related data in the "posts repository". Once full post data is retrieved for a user, this function notifies the "processing interest changes and targeted ads" block of the user's interest in the post's tags.

The "process adding and getting comments" function handles comments within the system. Its inputs and outputs are comments, and it stores comment data in the "comments repository".

The "processing rating updates" function manages user ratings for posts and comments. It validates whether a user is permitted to submit a rating. This function accepts the user's rating as input and produces updated rating data as output. When a post or comment is rated, the function updates the corresponding repository and also calls the "users repository" to adjust the user's overall rating. These user ratings form part of the "user data" output.

The "processing interest changes and targeted ads" function monitors notifications about user interest in posts with specific tags and records this data in the "ads repository". When an advertisement needs to be displayed, it is selected from the available list based on tags that align with the user's interests.

The "keep records of user's preferences" function manages user preferences, such as nickname or avatar. It interacts directly with the "users repository" to update this information, and the processed settings constitute a portion of the "user data" output.

Based on an analysis of the information system's data flow diagram, a list of relevant entities for the database has been compiled. These entities are as follows:

- User – This table stores core user information, including a unique id (primary key), email, password hash, login, rating, registration date, and a status (foreign key referencing the UserStatus table).
- UserStatus – This table contains the possible statuses a user can have, such as "ok" or "banned". Users with an "ok" status are permitted to authorize within the system. It has an id (primary key) to serve as a foreign key in the User table and a name field for the status.
- UserProfilePic – Designed to store a user's profile picture, this table contains an id (primary key), the user's id (foreign key referencing the User table), and the image stored as reference to picture service.
- UserSession – This table maintains user authorization sessions. It stores a session id (primary key), the user id (foreign key referencing the User table), a JWT authorization token, and the session creation date.
- Tag – This table holds all tags available within the system, containing fields for the tag id (primary key) and the tag name itself.
- PostTag – A junction table representing a many-to-many relationship between tags and posts. It contains the tag id (foreign key referencing the Tag table) and the post id (foreign key referencing the Post table), indicating which tags are associated with which posts.
- PostImage – This table stores images associated with posts. It includes an image id (primary key), the post id (foreign key referencing the Post table), and the image link to picture service.
- Post – The central table for storing posts. It contains fields for the post id (primary key), the user id (foreign key referencing the User table) of the author, the post title, the post text, the creation date, and the post rating.
- RatedPosts – A junction table that stores user ratings for posts to enforce that a user cannot rate the same post more than once. It contains the user id (foreign key referencing the User table), the post id (foreign key referencing the Post table), and the rating value ("LIKE" or "DISLIKE"). The entity is deleted if a rating is removed.

- Comment – This table stores comments on posts. Its fields include a comment id (primary key), the post id (foreign key referencing the Post table), the user id (foreign key referencing the User table) of the comment author, an isHidden flag, a parentId (foreign key referencing the Comment table, which can be null for top-level comments), the comment text, creation date, and rating.

- RatedComments – A junction table storing user ratings for comments, preventing users from rating the same comment multiple times. It holds the user id (foreign key referencing the User table), the comment id (foreign key referencing the Comment table), and the rating ("LIKE" or "DISLIKE"). The entity is deleted if the rating is removed.

- UserTagInterest – This junction table tracks tags a user has shown interest in by viewing related posts. It contains the tag id (foreign key referencing the Tag table), the user id (foreign key referencing the User table), a count of viewed posts with the tag, and the timestamp of the last viewing.

- AdvTag – A junction table for the many-to-many relationship between advertisements and tags. It stores the tag id (foreign key referencing the Tag table) and the advertisement id (foreign key referencing the Adv table), defining which tags are associated with which advertisements.

- AdvImage – This table stores images for advertisements. It contains an image id (primary key), the advertisement id (foreign key referencing the Adv table), and the image data as link to picture service.

- Adv – This table stores the advertisements themselves. It contains an advertisement id (primary key), a title, and the advertisement text.

In order to formalize the above-described business functions of the developed information system of the social network, a use-case diagram was developed. The developed diagram of usage options is shown in Figure 3.9.

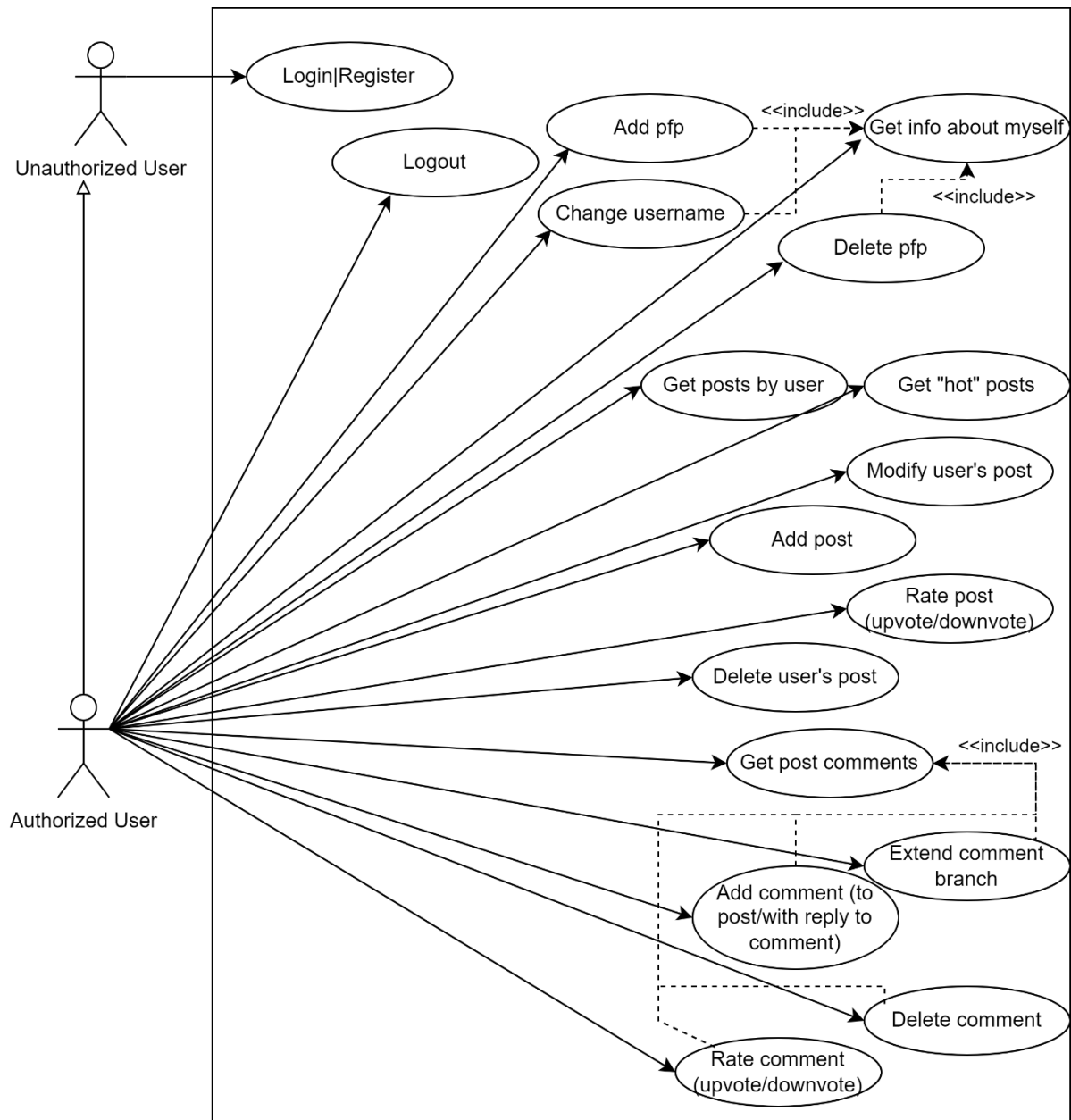


Figure 3.9 – Developed use-case diagram

Based on the use case diagram, two primary user roles can be identified: the guest and the authorized user. A guest is an unidentified user who has not yet been authorized by the system and whose access is restricted solely to authorization functions. An authorized user is a fully authenticated individual with access to the system's complete suite of functionalities, including the ability to create posts, submit ratings, and perform other interactive operations.

The use cases are described in greater detail below:

Login – This use case enables a user to access the system by providing credentials, specifically a nickname and password, allowing the system to identify them.

Registration – This function allows a new user to create an account on the platform by providing a nickname, password, and email address.

Add profile photo – This function permits an authorized user to upload a new profile picture. The most recently added photo is displayed as the user's avatar alongside their posts and comments.

Change nickname – This provides a user with the ability to modify their nickname, provided the new nickname is not already registered to another user in the system.

Delete profile photo – This function allows a user to remove one of their previously uploaded profile pictures.

Get information about yourself – This allows a user to retrieve data related to their own profile, including their nickname, email, rating, and the last 10 avatars they have used.

Get user posts – This function retrieves a paginated list of posts created by a specific user on the social network.

Get hot posts – This provides a paginated list of posts sorted by a relevance algorithm. This algorithm calculates a weighted average index based on the post's rating and its creation date.

Add post – This allows an authorized user to create a new post. The user can provide a title, optional text, optional images, and optional tags for the post.

Modify user post – This function enables the author of a post to alter its content, including the title, text, images, and associated tags.

Rate post – This allows a user to assign a rating to a post, which can be a "like," "dislike," or neutral. Rating a post also impacts the overall rating of the post's author.

Delete post – This function allows a user to remove one of their own posts from the system.

Get comments on a post – This retrieves a list of all comments associated with a specific post.

Open comment thread – This function is used to view comments that are replies to a specific parent comment, displaying a nested thread.

Add comment – This allows a user to add a comment to a post. The comment can be a direct reply to the post or a response to an existing comment within a thread.

Delete comment – This function permits a user to delete a comment they have authored.

Rate comment – This allows a user to assign a rating to a comment, using the same discrete scale of "like," "dislike," or neutral. Rating a comment affects the overall rating of the comment's author.

3.3 Design and development of information system components

When designing the information system, it was decided to implement an IS consisting of three functional services: user-service - which is responsible for authorization, storage of user data, etc.; content-service - service responsible for storing posts and comments; ads-service - responsible for providing contextual advertising. Also, during the development, will be applied pattern API Gteway, API Gateway is a service integration software, passing user requests to the services responsible for processing the request. API Gateway allows efficient and convenient horizontal scaling of the project. The overall structure of the designed system is shown in Figure 3.10.

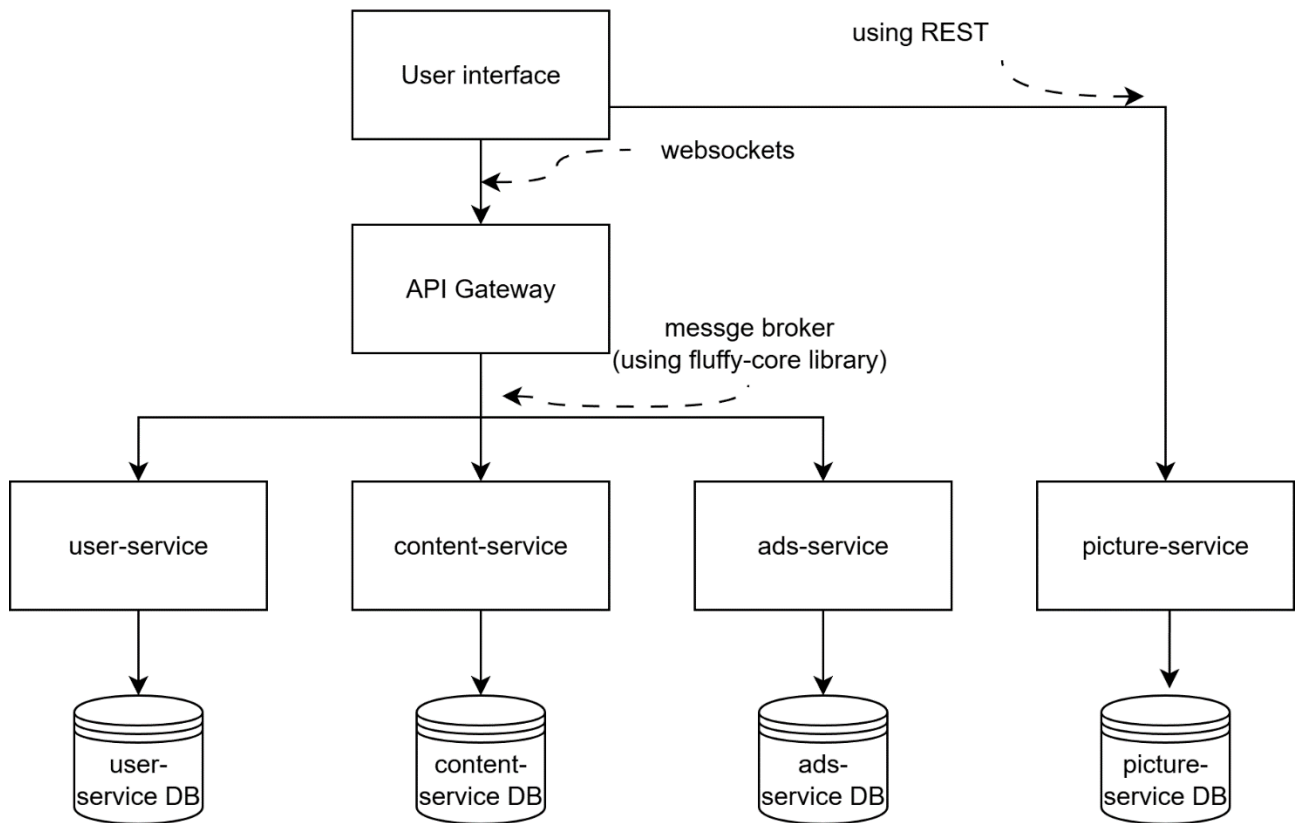


Figure 3.10 – Structural diagram of information system architecture

Studying the structural diagram shown in Figure 3.10 we can trace the life cycle of a request in the developed information system. User-service sends a request using WebSockets technology [20] to API Gateway, the gateway determines whether authorization is required, if necessary, sends a request to user-service to validate the provided credentials and redirects the request to the target service (using the fluffy-core library developed in Section 2), then the target service performs its own part of the business logic, possibly communicating with other services with the help of fluffy-core library. This approach to the implementation of the information system allows us to flexibly scale the architecture, add universal intermediate layers (e.g., authorization) and manage message routing within the infrastructure.

Understanding the request lifecycle is critical for diagnosing performance issues and validating architectural decisions. By following the path of a single request from the user-facing service down to internal components, developers can identify potential bottlenecks, security gaps, and points of failure.

In addition to the services described above, another one was added – picture-service. During testing it turned out that the infrastructure built in this way feels a significant load under the influence of a constant flow of pictures stored as base64 and redirected by the network stack. Moreover, it is absolutely unprofitable to implement image caching with this storage format. To optimize the work of the information system, we implemented a service that stores user images and interacts with the client directly, using REST, which significantly increases the performance of the entire system.

Among all services presented above API Gateway takes a separate place, it performs functions of access control for external requests, i.e. does not allow to call hidden (service) methods from outside the infrastructure and for some methods serves as a common middleware, controlling authorization and passing authorization data further along the pipeline. API Gateway receives a request via WebSockets (using `websockets-rush-api` library [21]), performs internal flow over the request data, creates a request with the request data inside the infrastructure (using `fluffy-core`), receives the response and forwards the response via WebSockets.

This gateway-based design also improves the system's resilience. Since all external traffic passes through a single entry point, it becomes straightforward to implement rate limiting, request retries, and circuit breakers at the perimeter. For example, if the user-service becomes temporarily unavailable, the API Gateway can cache authorization decisions for a short window or return a graceful error response, preventing cascading failures to downstream services. Moreover, the gateway can be horizontally scaled behind a load balancer, allowing the infrastructure to handle spikes in external traffic without modifying the internal service mesh.

The algorithm of the API Gateway service is shown in Figure 3.11.

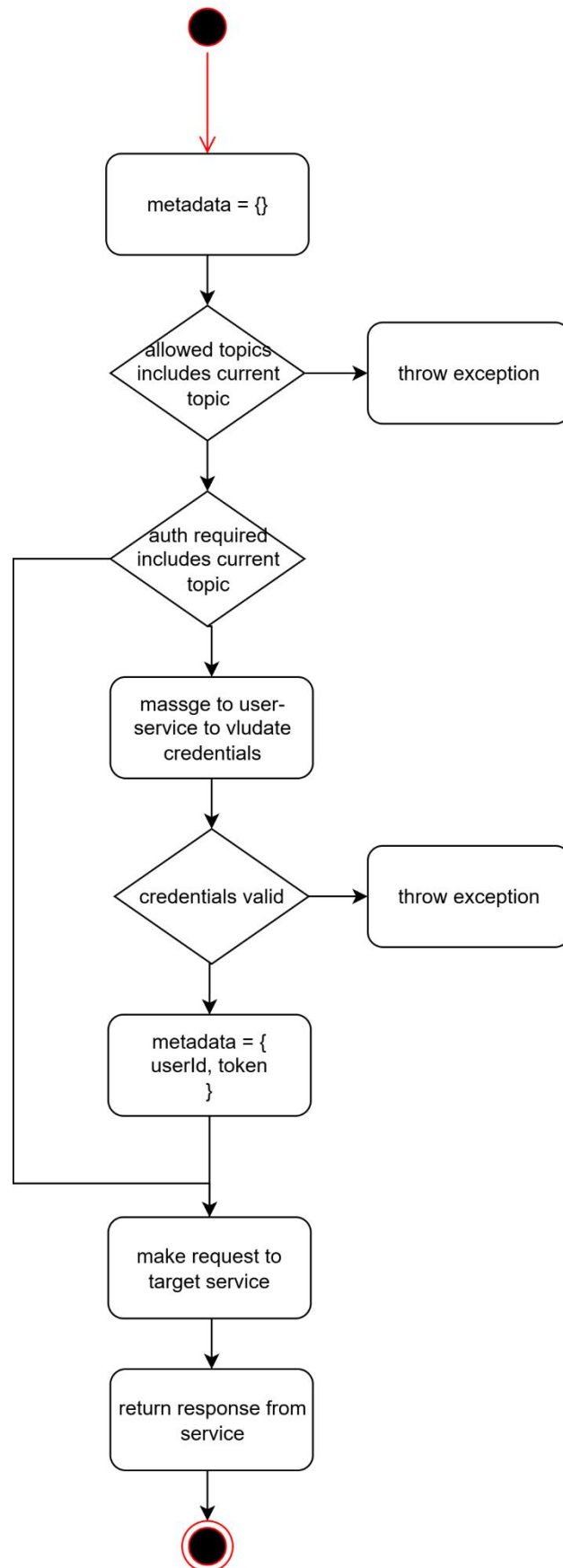


Figure 3.11 – UML Activity Diagram of API Gateway request life cycle

Since the development of a microservice system directly implies weak cohesion of modules and independence of services from each other, the databases [22] of developed services will be separated, ER-diagram of databases is shown in Figure 3.12.

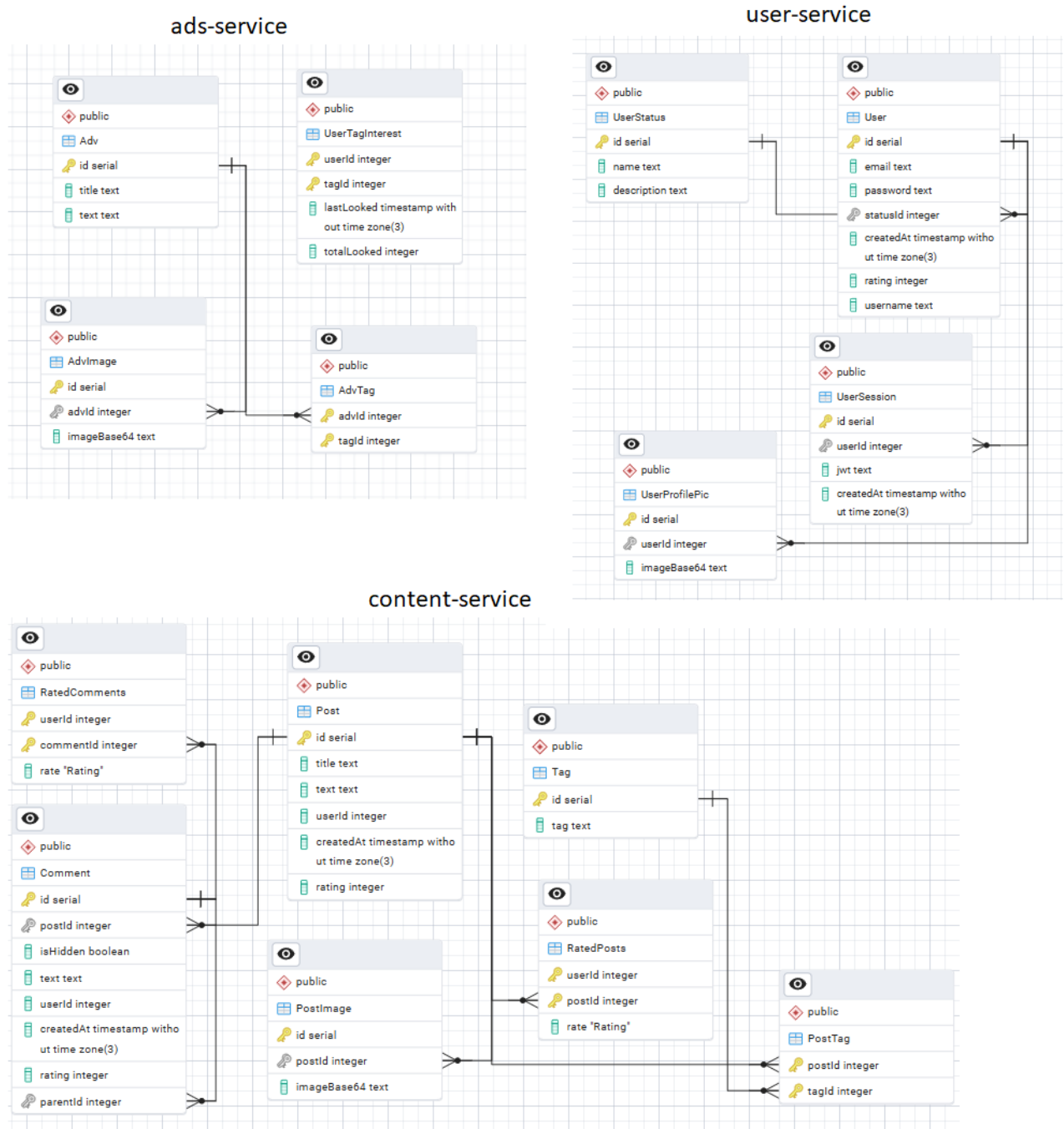


Figure 3.12 – ER diagram of services databases (generated with PGAdmin software)

The diagram provided in Figure 3.12 shows the physical structure of the information system organized according to the microservice architecture. Each group of tables allocated for a certain service reflects a separate functional module of the social network.

The ads-service concentrates the tables that are responsible for storing ads and tagging them. The main entity is Ad, which stores the title and text of the ad. One or more images can be linked to it via the AdImage table. Additionally, there is an AdvTag table that implements the association of ads with tags, which allows categorizing them and providing more flexible search. The UserTagInterest table is used to keep track of user interests: it records which user is interested in which tag, when he last interacted with content on this tag and how many times he viewed it. This part of the database allows you to recommend ads to users based on their activity and interests.

The user-service contains entities that describe the users themselves and their authorization data. The central table is User, which stores the basic data: email address, password, username, rating and status reference. The UserStatus reference stores possible user states, and the UserSession table stores information about active sessions, including JWT tokens and the time they were created. A separate UserProfilePic table allows storing user avatars in base64 image references from the picture-service. This part of the schema is fully responsible for identification and authentication of users, as well as for managing their profiles.

The content-service implements the logic of creating and interacting with user content. The central entity is Post, where title, text, authorship, creation time and publication rating are recorded. Images are linked to posts via PostImage, as well as comments via the Comment table, which supports a hierarchical structure by referring to the parent comment. Separate mechanisms are provided for content rating: the RatedPosts table stores user ratings of posts, and RatedComments stores ratings of comments. The Tag directory defines available tags, and the connection between posts and tags is realized through the PostTag intermediate table. Thus, the

content service supports a full-fledged cycle of publishing, commenting and rating materials.

As mentioned earlier, the fluffy-core library is also used for inter-service communication. For example, in order to get data about a particular post, a client accesses content-service, which stores data about posts and comments, but, for example, the essence of the post implies username and avatar of the author. In order to collect all the necessary data about the post and provide it to the client, content-service has to send a user-service request to get information about the user. In Figure 3.13, we can see the inner logics of the `getPost` method. When the method has prepared the post to be sent, at the very end it calls the `getUser` helper (whose program code is shown in Figure 3.14), which in turn sends a query to user-service and returns a response with user data. In cases when it is necessary to get a set of users (for example, to prepare feed posts), the `getUserMany` method is called in the same way, which returns data about a collection of users to optimize data processing and don't make several similar requests.

```
public static async getPost(id: number, requester: AuthInterface | undefined):
Promise<Post | any> {
  const post = await database.post.findFirst({
    where: { id: id },
    include: this.defaultPostIncludes
  })
  if (!post) throw new APIError('Post not found', 404)
  let rating = null
  if (requester) {
    rating = await PostRatingService.getMyRating(requester.id, post.id)
    await this.noifyTagInterest(requester.id, post.PostTag.map((t) =>
t.tagId))
  }
  const comments = await this.calculateComments(post.id)

  const user = await getUser(post.userId)

  return { ...post, myRate: rating, comments, user}
}
```

Figure 3.13 – Program code of `getPost` method

```

async function getUser (userId: number) {
  const resp = await fluffyCore.makeRequestSimplify({ data: { id: userId },
topic: 'user_get_one' })
  return resp
}

```

Figure 3.14 – Program code of getPost method

Also, according to the business logic, when a user visits an individual post, we should record that the user was interested in the corresponding tags. The query for the post is in content-service, and the data about which tags the user was interested in is in ads-service. Accordingly, we can conclude that we need to make a request from content-service to ads-service notifying that the user was interested in certain tags. Moreover, since the content-service does not care about the result of the query, we can optimize this interaction without requiring ads-service to respond by using the fire-and-forget inter-service interaction pattern, which is also implemented in fluffy-core. In Figure 3.13 we can see that if authorization data has been received, we call the noifyTagInterest method where we associate the user id with the tags viewed. The noifyTagInterest method itself is shown in Figure 3.15 and performs the task of sending the corresponding ads-service message.

```

public static async noifyTagInterest(uid: number, tagIds: number[]):
Promise<void> {
  await fluffyCore.fireAndForget({ data: { uid, ids: tagIds }, topic:
'ads_notify_tag_interest' })
}

```

Figure 3.15 – Program code of noifyTagInterest method

Now that the whole project uses the fluffy-core library, we can switch to a different message broker at any time by replacing a couple lines in the configuration file and pulling in new environment variables.

4 CONDUCTING EXPERIMENTS AND COMPARING THE RESULTS OBTAINED

4.1 Performance evaluation of the developed library in the context of different message brokers

During this qualification work we developed flyffy-core library, which is a layer of abstraction over various message brokers and allows to quickly and conveniently implement microservice architecture or efficiently change the provider without having to significantly change the internal business logic of the whole project. This section proposes to evaluate the overall messaging performance using the developed library “in a vacuum” - laboratory conditions. In this chapter, two performance tests will be performed: sequential query execution and multi-threaded query execution.

In a single-threaded implementation, testing goes as follows: a responder provider is created, a responder fluffy-core instance is created, a root with minimal business logic is registered, and the responder is started. The requesting party provider is created, the requesting party fluffy-core instance is created. Next, the requesting instance makes one test request to make sure that the connection is established and basic testing can be performed. Then, in a loop, the requesting party sequentially makes 100, 1000, 10000, 100000, 1000000 requests. The execution time of the requests is recorded. A request is considered successfully executed when the requesting party has received a response from the responding party. Only one query is executed at a time. The table contains the message broker, the number of requests, the total request execution time and the RPS (Requests Per Second), which is calculated as the number of requests divided by the time ($RPS = \frac{r_0}{t_0}$), it is an average that indicates how many requests per second the infrastructure can handle.

In a multi-threaded implementation, the test proceeds as follows: a responder provider is created, a responder fluffy-core instance is created, a router with minimal business logic is registered, and the responder is started. The loop creates n

requesting party providers and n requesting party fluffy-core instances, where n is the number of threads, so that a test running even within a single code section works as a distributed system. Next, the requesting instances make a single test request to make sure that the connection is established and the basic testing can be performed. If testing of at least one instance fails, an exception is thrown. Next, for each instance of the requesting party, a multithreaded testing thread is started, each instance takes $r_i = r_0/n$ requests. The execution time of the requests is recorded. A request is considered successfully executed when the requesting party has received a response from the responding party. There are n requests executed simultaneously. The test execution time is the total time until all instances of the requesting party have completed their execution. The table contains the message broker, the number of requests, the total request execution time, and the RPS. In this particular case, this test will be performed with $n = 10$, meaning there are 10 simultaneous requests at maximum.

Let us perform the testing according to the above methodologies and record the results in Tables 4.1 and 4.2 for single-threaded mode - execution time and RPS metric respectively and Tables 4.3 and 4.4 for multi-threaded mode - execution time and RPS metric respectively.

Table 4.1 – Fluffy-core time performance evaluation with different message brokers (single thread)

Requests (r_0) Message broker	100	1000	10000	100000	1000000
	Time of execution $t_0(s)$				
NATS	0.15	1.27	11.5	112.72	1393.45
Apache Kafka	0.42	3.2	27.02	271.87	2737.32
Redis Streams	0.15	1.43	13.89	195.96	1496.75
RabbitMQ	0.25	1.56	13.41	127.97	1335.67

Table 4.2 – Fluffy-core RPS performance evaluation with different message brokers (single thread)

Requests (r_0) Message broker	100	1000	10000	100000	1000000	RPS_{avg}
	RPS $t_0(s)$					
NATS	655	789	869	887	717	783.4
Apache Kafka	236	312	370	367	365	330
Redis Streams	648	701	720	510	668	649.4
RabbitMQ	408	642	745	781	748	664.8

Table 4.3 – Fluffy-core time performance evaluation with different message brokers (multi thread)

Requests (r_0) Message broker	100	1000	10000	100000	1000000
	Time of execution $t_0(s)$				
NATS	0.1	0.42	2.99	31.25	326.24
Apache Kafka	0.3	2.12	13.69	110.35	1167.71
Redis Streams	0.11	1.14	8.66	86.92	822.59
RabbitMQ	0.09	0.38	2.34	24.19	263.81

Table 4.4 – Fluffy-core RPS performance evaluation with different message brokers (multi thread)

Requests (r_0) Message broker	100	1000	10000	100000	1000000	RPS_{avg}
	RPS $t_0(s)$					
NATS	1052	2385	3343	3200	3065	2609
Apache Kafka	332	472	730	906	856	659.2
Redis Streams	895	876	1154	1150	1215	1058
RabbitMQ	1098	2606	4275	4134	3790	3180.6

Based on Tables 4.1 and 4.2 we can draw conclusions about the performance of the developed solution when working in single-threaded mode. From these tables we can see that for all message brokers, averaged, as the number of requests within a single test increases, their performance increases, which is expected since the amount of overhead decreases relative to the amount of payload. We can see that for all message brokers the maximum performance is reached on the order of 10000 and 100000 queries within a test, approaching 1000000 queries, the performance does not decrease significantly. Carrying out comparisons between message brokers, we can quite build the top leaders by performance in request-reply mode, we will take average RPS as an evaluation metric. The most productive in this test was NATS (average RPS - 783.4), which natively supports request-reply mode; the next in terms of performance is rabbitMQ message broker with built-in support for the RPC pattern (average RPS - 664.8); a bit lower is Redis Streams (average RPS - 649.4), which although it is a solution that effectively implements the pub-sub pattern, it is not really suitable for exchanging a large number of messages, we can see in Table 4.2 that Redis Streams loses its efficiency after 10000 requests, although it shows a very good result on small numbers of requests; and last is Apache Kafka (average RPS -

330), in request-reply mode Kafka is very inferior to all the above mentioned message brokers. Kafka implements high coherence of requests, logging, delivery guarantees, which naturally adds a significant amount of overhead and slows down its work in request-reply mode, in principle, it was not designed to work in this mode, however, in practice, it is often used in this way, especially if Kafka has already been integrated into the project to perform certain tasks.

Now consider Tables 4.3 and 4.4, we can draw conclusions about the performance of the developed solution when working in multithreaded mode. The tests were performed with the number of threads - 10. Of course, in Table 4.4 we won't see a linear performance growth relative to the provided resources, but we will see a significant performance growth for sure. On these tables we see a very similar picture to the single-threaded test, for all message brokers, on average, as the number of requests within a single test increases, their performance increases, which is expected, since the amount of overhead decreases relative to the amount of payload. We can see that for all message brokers the maximum performance is reached on the order of 10000 and 100000 queries within a test, approaching 1000000 queries, the performance does not decrease significantly. However, Redis Streams, as the number of queries within a single test increases, the performance gains noticeably until the end of the table arguments. Making comparisons between message brokers, we can quite build the top leaders in terms of performance in request-reply mode, taking the average RPS as an evaluation metric. The best performing RabbitMQ (average RPS - 3180. 6), which is probably the most widely used message broker in microservice architectures at the moment; next in performance is the NATS message broker (average RPS - 2609); a bit lower is Redis Streams (average RPS - 1058), which, although not designed as a request-reply broker for processing a large number of requests, still goes beyond 1000 RPS (which, subjectively, should be enough for a very large number of projects); and last is Apache Kafka (average RPS - 659. 2), as it was said earlier Apache Kafka is poorly suited for the request-reply mode of operation and accepts such an assignment rather by the fact of its presence in the

project and besides this approach to its use it has many other purposes in the realization of which it is superior to the above-mentioned brokers.

Such a multi-threaded test illustrates quite closely the behavior observed in a real-world environment. When many users make requests, these requests are processed multithreaded, therefore, the multithreaded performance test can be considered close to reality.

Figures 4.1 and 4.2 show comparative plots of the performance dependency (RPS) when using fluffy-core with different message brokers against the number of requests within the experiment.

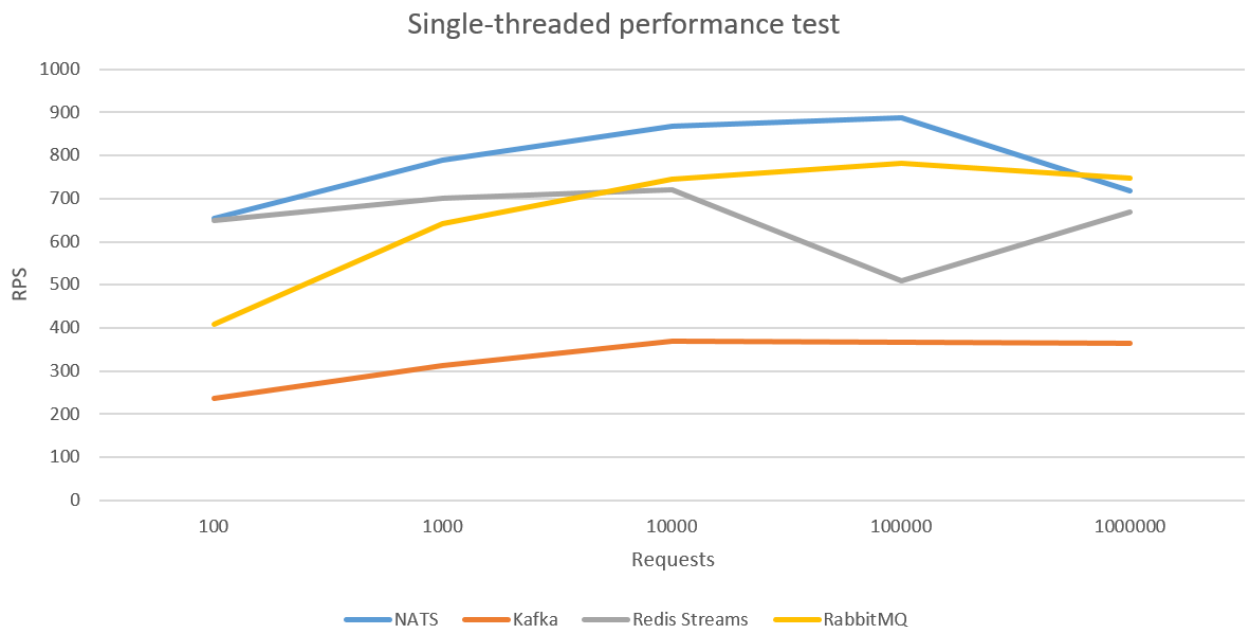


Figure 4.1 – Single-threaded performance test for different message brokers

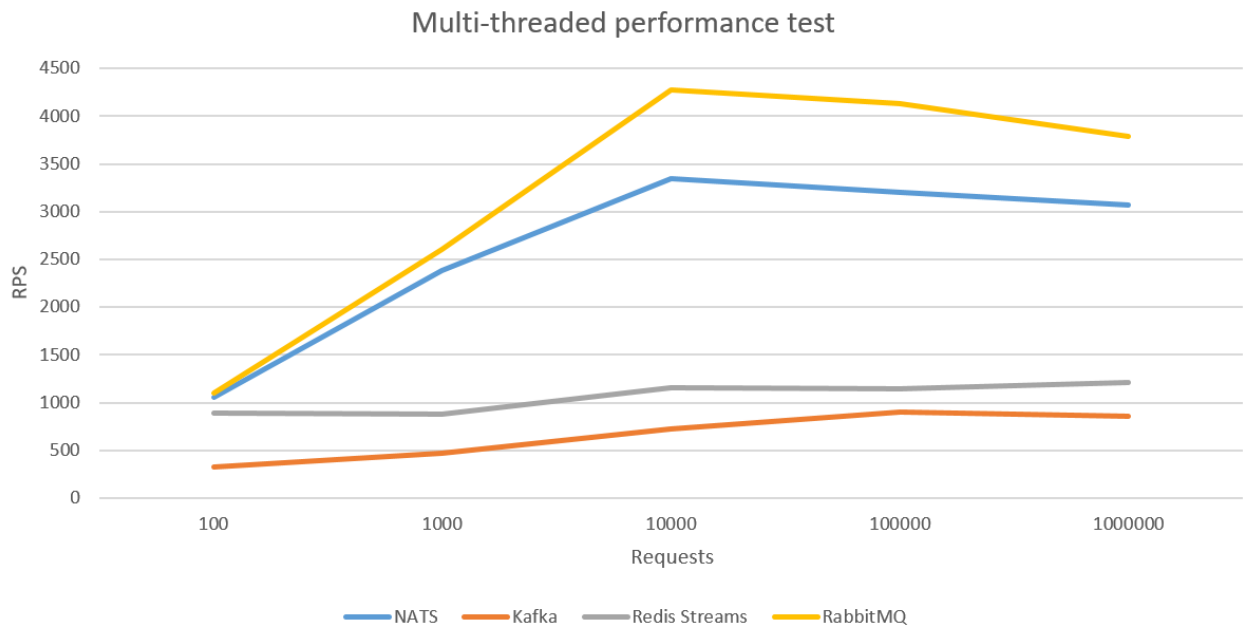


Figure 4.2 – Multi-threaded performance test for different message brokers

As we can see in Figure 4.1 - the performance curves of RabbitMQ and NATS are very close, on a small number of messages NATS outperforms RabbitMQ, but by the end of the test RabbitMQ outperforms NATS just a bit. Surprisingly, Redis Streams comes close to RabbitMQ and by the end of the experiment is almost on par with RabbitMQ and NATS, even though it is not a specialized broker. Apache Kafka's performance increases gradually and at any point of measurement is significantly lower than the other brokers.

In Figure 4.2 we can notice that RabbitMQ outperforms NATS at any stage of the experiment, the RPS of these brokers are quite close and after reaching 10000 requests, the performance of RabbitMQ and NATS almost reach a plateau. Redis Streams goes very close to Kafka in terms of performance, but at any stage of measurement Redis Streams is more efficient than Apache Kafka. The performance of all brokers grows up to 10000 (in case of Kafka - up to 100000 requests) and then reaches a plateau with insignificant performance decrease.

Considering that RabbitMQ and NATS are approximately equivalently easy to configure and RabbitMQ shows slightly better results in multithreaded mode, which is closer to reality, in the practical construction of information system in microservice

architecture, where it is important to high performance of its components, it is recommended to use RabbitMQ as a message broker to work in request-reply mode, but the use of NATS for these purposes is also not excluded, because between these technologies the difference in performance is very insignificant. Redis Streams are recommended only if the project already uses Redis infrastructure and the task is to implement request-reply interaction between services to process a small number of messages. Apache Kafka is recommended only if the project already uses Kafka and the load on request-reply logic is minimal.

4.2 Analysis of the potential use of the developed software solution in practice

In Section 3 of this qualification work, an information system based on a microservice architecture was implemented using the fluffy-core library designed and developed in Section 2. The development task of fluffy-core was to develop a lightweight, easy-to-integrate layer that unifies the interaction with the message broker in the microservice architecture. The developed solution implements all the set tasks and allows to quickly and conveniently develop projects in the microservice paradigm. An example of the interface [23] of the developed information system is shown in Figure 4.3.

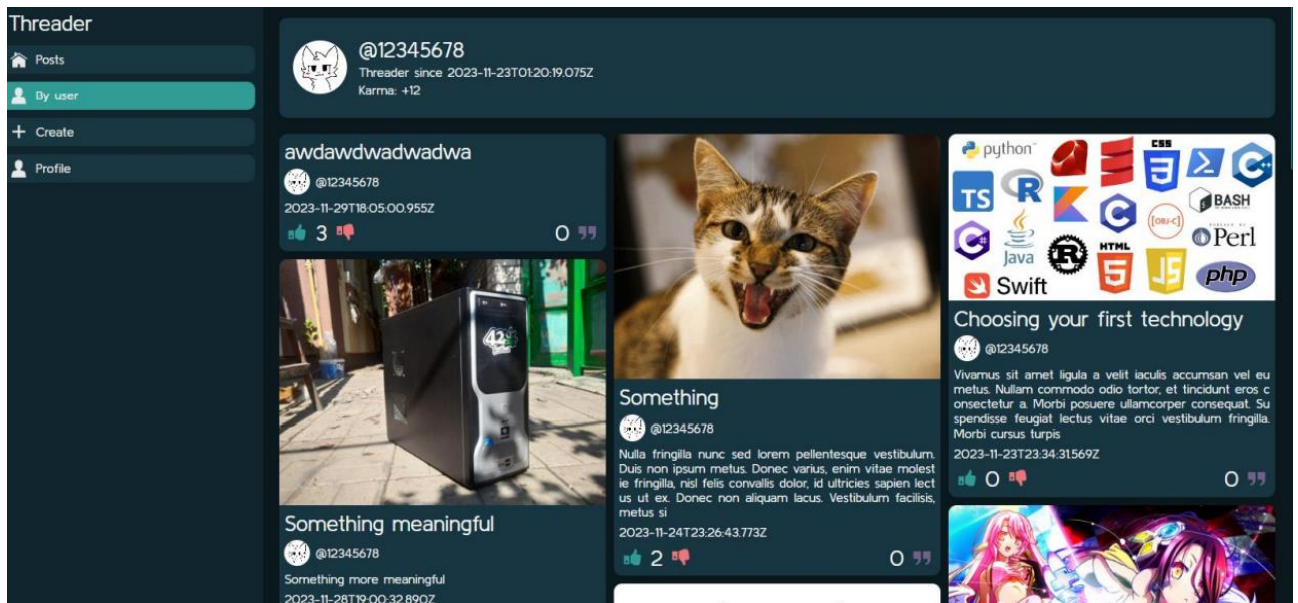


Figure 4.3 – Practical application of the developed library

Integration of the library into the project is very easy, Figure 4.4 shows a sample code from the content-service that initializes the library.

```
import {FluffyCore, NATSProvider} from "fluffy-core";
import {NATS_URL} from "@config/env";

const provider = new NATSProvider(NATS_URL);
const fluffyCore = new FluffyCore(provider);

export default fluffyCore;
```

Figure 4.4 – fluffy-core initialization

After the steps in Figure 4.4, you can access the `fluffyCore` instance and use all of its methods. In case it is necessary to change the message broker, you can simply replace the value of the `provider` variable and all business logic will work as before.

The developed library is available in the public domain and can be installed in the project with the command `npm install fluffy-core` [24], also, it is open source, i.e. you can always study the source code of the project, suggest edits or, based on it, develop your own library for the specific needs of the developer.

One of the development requirements mentioned earlier was the requirement to be able to use a custom provider in case the end developer lacks the ones offered by the library. The developer can do this by implementing the `IProvider` interface and then use this provider simply by passing it to the `FluffyCore` constructor.

Taking advantage of the ease of migration to other message brokers, lightweighting and ease of integration of the developed library, `fluffy-core` has all the potential to be an excellent choice for small projects and startups where speed of development, ease of technology change, lightweighting and performance are important. The performance values obtained in the experimental phase are more than sufficient for very many projects on the market and as the experimental and practical part of this work has shown, the developed library is really convenient and easy to integrate.

4.3 Exploring the advantages of using microservice architecture

Microservice architecture is an evolutionary approach to the design of complex software systems based on the decomposition of functionality into independent, loosely coupled services. This approach has replaced traditional monolithic solutions, offering fundamentally new opportunities for software development and operation.

The key advantage of microservice architecture is the possibility of independent deployment and scaling of individual system components. Each service can be deployed, updated and scaled independently from the others, which significantly accelerates the delivery process and allows to optimize the use of resources. This is especially important in the conditions of rapidly changing requirements and the need to respond quickly to changes in the load on different parts of the system.

Technological heterogeneity represents another significant advantage of the approach. Different microservices can be implemented using different technology stacks, programming languages and databases that are most suitable for specific

tasks. This allows choosing the optimal tools for each subject area and gradually introducing new technologies without the need to rewrite the entire system.

Resilience to failure is greatly enhanced by service isolation. The failure of one microservice does not lead to a complete failure of the entire system, as other components continue to function. The architecture allows the implementation of sophisticated failure handling mechanisms such as circuit breakers and graceful degradation, which increases the overall system reliability.

System development and support is simplified by a clear division of responsibilities between services. Small teams can work independently on individual microservices, which speeds up the development process and reduces coordination costs. Each service has clearly defined boundaries and interfaces, making it easier to understand, test, and modify.

The evolutionary nature of the architecture allows the system to evolve gradually, replacing or modifying individual services without impacting the entire system. This is especially important for large projects with a long lifecycle, where the possibility of gradual modernization becomes a critical success factor.

Optimization of development processes is achieved through the ability to organize small autonomous teams around individual services. Each team can independently make decisions about technologies, processes and development pace, which increases the efficiency and motivation of participants.

Conformance to Domain-Driven Design principles [25] is evident in the natural mapping of business domains to architectural components. Microservices typically correspond to specific business capabilities or subdomains, which improves system understanding and alignment with business requirements.

Cost-effectiveness in the long run is achieved by optimizing resources, accelerating time-to-market, and reducing the cost of change. While the initial investment in a microservice architecture may be higher than in a monolith, the total cost of ownership of a complex system is often lower due to these benefits.

CONCLUSIONS

During the fulfillment of this qualification work, the research of methods of service communication in microservice architecture was carried out. We analyzed the subject area, where we identified the main approaches to the development of information systems in the monolithic and microservice paradigm, identified the main features of these architectures. We analyzed the peculiarities of communication in microservice architecture, variants of service access implementations (using Gateway API and without), the main approaches to the implementation of inter-service interaction - choreography / orchestration, request-reply, pub-sub, fire-and-forget. Evaluated modern approaches to application development in microservice architecture, researched modern message brokers and compared them by a number of criteria. Analyzed existing solutions on the market that allow message brokers to be integrated into a technical solution. In the process of subject area analysis we identified a significant problem that systematically, for a long period of time, complicates the development of modern microservice information systems for small projects and start-ups - the problem of developing a modern, lightweight, easy-to-integrate and universal solution for the initialization of inter-service communication in microservice architecture.

The design and development of a universal environment for application development in microservice architecture was carried out, the requirements for the designed software solution were developed, the technological stack was defined and justified, the architecture of the software solution in the form of a library for Node.js and the functions that this library should implement were defined. The algorithms by which the developed library should function were defined. Finally, the development and publication for public access of the software product is carried out, the features of working with the package repository npm are specified.

In section 3, on the basis of the previously developed library, the design and development of an experimental information system was carried out, the purpose of such in this qualification work is to demonstrate the ease of integration of technology

and to demonstrate the typical infrastructure of such a project. During the course of the project the subject area of social networks was analyzed, the functional design of the information system was carried out. The development of information system components was carried out, where it was demonstrated how inter-service communication functions using the developed library.

Further, the actual testing of the developed software product was carried out, a test bench and testing methodology were developed to test and numerically evaluate the performance of the developed library with various message brokers (given in the analysis of the subject area), based on this numerical evaluation, the message brokers that are best suited to work in request-reply mode were identified, for each message broker recommendations for use were determined. Further, the potential of using the developed solution in real-world conditions was analyzed and the advantages of using microservice architecture in the development of web applications were analyzed.

Conclusions were drawn.

As a result of this qualification work, we have obtained a tested and validated technology for solving the problem of inter-service communication in microservice architecture, the source code of the developed library can be found here: <https://github.com/dmytryG/fluffy-core> and installed in your project here: <https://www.npmjs.com/package/fluffy-core?activeTab=readme>. For the developed solution a number of experiments were conducted to determine the performance of the infrastructure, which showed quite specific and clean results. Also in the process of this qualification work were developed test beds in the form of an experimental information system, which was developed using the knowledge gained during the implementation of this qualification work and using the library developed in section 2 of this work, with the source code of stress tests and experimental information system can be found here: <https://gitlab.com/dmytrygolub/soa/-/tree/masters>.

At the current moment of the qualification work, we can say that all the tasks and goals set in the section analyzing the subject area have been successfully completed.

LIST OF REFERENCED SOURCES

1. M. Richards, N. Ford Fundamentals of Software Architecture: Textbook. – Sebastopol: O'Reilly, 2020. – 383p.
2. S. Newman Monolith to Microservices. Evolutionary Patterns to Transform Your Monolith: Textbook. – Sebastopol: O'Reilly, 2020. – 274p.
3. M. Massé REST API Design Rulebook: Textbook. – Sebastopol: O'Reilly, 2012. – 114p.
4. S. Newman Building Microservices: Textbook. – Sebastopol: O'Reilly, 2015. – 280p.
5. RabbitMQ 4.2 Documentation [Electronic source] – Resource access mode: <https://www.rabbitmq.com/docs> (last accessed: 01.02.2026).
6. Documentation for Apache Kafka [Electronic source] – Resource access mode: <https://kafka.apache.org/41/getting-started/introduction/> (last accessed: 01.02.2026).
7. The official NATS documentation [Electronic source] – Resource access mode: <https://docs.nats.io/> (last accessed: 01.02.2026).
8. A. Mead Learning Node.js Development: Textbook. – Birmingham: Packt Publishing, 2018. – 649p.
9. C. Cowell, N. Lotz, C. Timberlake Automating DevOps with GitLab CI/CD Pipelines: Textbook. – Birmingham: Packt Publishing, 2023. – 348p.
10. Introduction to Redis streams [Electronic source] – Resource access mode: <https://redis.io/docs/latest/develop/data-types/streams/> (last accessed: 01.02.2026).
11. T. Bayer T. Polley The API Gateway Handbook: Textbook. – Bonn: predic8 GmbH, 2025. – 278p.
12. S. Pasquali Mastering Node.js: Textbook. – Birmingham: Packt Publishing Ltd., 2013. – 347p.
13. TypeScript Documentation [Electronic source] – Resource access mode: <https://www.typescriptlang.org/docs/> (last accessed: 01.02.2026).
14. M. Fowler, K. Scott UML Distilled: A Brief Guide to the Standard Object Modelling Language: Textbook. – Massachusetts: Addison Wesley Longman, Inc., 1999. – 234p.
15. S. Stefanov K. Sharma Object-Oriented JavaScript: Textbook. – Birmingham: Packt Publishing, 2013. – 382.

16. Голуб Д. К. Розробка технології для забезпечення міжсервісної комунікації у мікросервісній архітектурі. Матеріали Міжнародної науково-технічної конференції «Інформаційно-комунікаційні технології та кібербезпека» – Харків: ХНУРЕ. 2025. – с. 367-369.
17. K. Hamilton, R. Miles Learning UML 2.0: Textbook. – Sebastopol: O'Reilly Media, 2006. – 303p.
18. R. Jemnica Functional and Information Modeling of Production Using IDEF0 Methods: Textbook. – Belgrade: Journal of mechanical engineering 55, 2009. – 134p.
19. C. Simsion, C. Witt Data Modeling Essentials: Textbook. – San Francisco: Morgan Kaufmann Publishers, 2005. – 531p.
20. The WebSocket Protocol. RFC-6455 [Electronic source] – Resource access mode: <https://datatracker.ietf.org/doc/html/rfc6455> (last accessed: 01.02.2026).
21. Голуб Д. К. Розробка компонентів інформаційної системи із функцією підбору таргетованої реклами. // 28-й Міжнародний молодіжний форум «Радіоелектроніка та молодь у XXI столітті». Зб. матеріалів форуму. Т. 6. Конференція «Інформаційні інтелектуальні системи» – Харків: ХНУРЕ. 2024. – с. 610.
22. PostgreSQL Documentation [Electronic source] – Resource access mode: <https://www.postgresql.org/docs/> (last accessed: 01.02.2026).
23. O. Galitz The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques: Textbook. – Indianapolis: Wiley Publishing, 2007. – 888p.
24. npm Docs [Electronic source] – Resource access mode: <https://docs.npmjs.com/> (last accessed: 01.02.2026).
25. S. Millett, N. Tune Patterns, Principles, and Practices of Domain-Driven Design: Textbook. – Indianapolis: John Wiley & Sons, Inc., 2015. – 795p.