

University of Tartu

Faculty of Social Studies

Narva College

Study program: "Information Technology Systems Development"

Oleksandr Mesliuk

ARCHITECTURAL TRANSFORMATION OF LEGACY PLATFORMS

Diploma Thesis

Supervisor: Andre Säask, M. Sc.

Narva 2026

DECLARATION

Selle töö valmistasin ette iseseisvalt. Töö ettevalmistamisel kasutasin viiteid kõikidele teiste autorite töödele, põhipunkte ning andmeid kirjandusallikatest ja muudest allikatest.

/allkirjastatud digitaalselt/

Oleksandr Mesliuk

25.04.2026

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, _____ Oleksandr Mesliuk _____,

(autori nimi)

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose

_____ Vananenud platvormide arhitektuuriline ümberkujundamine _____,

(lõputöö pealkiri)

mille juhendaja on _____ Andre Säask _____,

(juhendaja nimi)

reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.

2. Annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creative Commons'i litsentsiga CC BY NC ND 4.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni.
3. Olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.
4. Kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tuletenevaid õigusi.

Oleksandr Mesliuk

10.05.2026

TERMS AND ABBREVIATIONS

Abbreviations

API (Application Programming Interface) – A set of rules and protocols that allows different software applications to communicate with each other.

AWS (Amazon Web Services) – A comprehensive cloud computing platform providing various scalable computing resources and services for application deployment.

BPA (Business Process Analysis) – A methodology for evaluating a system's current state (As-Is) and modeling improvements (To-Be) to identify and resolve bottlenecks.

CI/CD (Continuous Integration / Continuous Deployment) – The practice of automating the building, testing, and deployment of code to ensure reliable software delivery.

CQRS (Command Query Responsibility Segregation) – An architectural pattern that separates the reading and writing of data into different models to optimize performance.

DDD (Domain-Driven Design) – A software development approach that centers the architecture and structural design around business domains (e.g., user, listing).

DRY (Don't Repeat Yourself) – A fundamental principle of software development aimed at reducing the repetition of software patterns and code duplication.

EC2 (Elastic Compute Cloud) – A web service in AWS that provides secure, resizable compute capacity (virtual servers) in the cloud.

FSD (Feature-Sliced Design) – An architectural methodology for frontend applications that enforces strict separation of concerns by dividing the codebase into isolated layers and features.

gRPC (gRPC Remote Procedure Calls) – A high-performance, open-source universal RPC framework used for efficient communication between microservices.

ORM (Object-Relational Mapping) – A programming technique that lets work with a database using objects instead of raw SQL queries.

JWT (JSON Web Token) – An open standard used to securely transmit information between parties as a JSON object, widely utilized for user authentication and authorization.

KPI (Key Performance Indicator) – A measurable value that demonstrates how effectively a system or business is achieving its key objectives.

OTP (One-Time Password) – A password that is valid for only one login session or transaction, used to enhance security during user verification.

REST / RESTful (Representational State Transfer) – An architectural style for providing standards between computer systems on the web, making it easier for systems to communicate.

RPS (Requests Per Second) – A throughput metric used during load testing to measure the number of concurrent requests a system can handle per second.

SRP (Single Responsibility Principle) – A software design principle stating that every module or class should have responsibility over a single part of the functionality.

TTM (Time to Market) – The length of time it takes from a product or feature being conceived until it is deployed to the production environment and available to users.

VPS (Virtual Private Server) – A virtual machine sold as a service by a hosting provider, used in this project for independent microservice hosting.

SEO (Search Engine Optimization) – The process of improving a website so that it ranks higher in search engine results (like Google), to increase organic (non-paid) traffic.

UI (User Interface) – it is about how a product looks and feels. It focuses on the visual and interactive elements.

RTK Query (Redux Toolkit Query) – The data-fetching and caching tool that comes from Redux Toolkit. It is designed to simplify how you fetch, cache, synchronize and update server data in React(or other) applications.

DORA (DevOps Research and Assessment) – the longest-running research program identifying capabilities that drive software delivery and organizational performance.

LLM (Large Language Model) – A type of AI trained on massive datasets to understand, summarize, generate, and predict human-like text and code.

OPEX (Operational Expenditure) – The current costs incurred in operating, maintaining and sustaining the performance of an AI system.

Terms

Event-Driven Architecture – A software architecture paradigm promoting the production, detection, and consumption of events via an asynchronous message broker (e.g., Apache Kafka).

Legacy System – An outdated computing software or architecture that remains in use despite its maintenance difficulties, tight coupling, and scalability limitations.

Message Broker – An intermediary software module (e.g., Apache Kafka) that translates, routes, and manages messages between independent microservices for asynchronous communication.

Microservices – An architectural pattern that structures an application as a collection of loosely coupled, independently deployable services, each organized around a specific business capability.

Monolithic Architecture – A traditional unified software model where the user interface, data access, and core business logic are combined into a single, indivisible codebase.

Strangler Fig Pattern – A software modernization strategy used to safely migrate a legacy system by gradually replacing specific functionalities with new distributed services until the monolith is fully decommissioned.

Technical Debt – A concept in software engineering that reflects the implied cost of future rework caused by choosing quick, short-term architectural compromises instead of applying better, long-term design patterns.

Tight Coupling – A characteristic of software architecture where internal components or modules are highly dependent on one another, making code modifications difficult and increasing the risk of system-wide failures.

CONTENTS

TERMS AND ABBREVIATIONS	3
Abbreviations.....	3
Terms	5
CONTENTS	6
INTRODUCTION	9
Problem	9
Goal	9
Tasks.....	9
Research questions	10
Hypothesis	10
1 THE PROBLEM OF TECHNICAL DEBT IN LEGACY SYSTEMS	11
1.1 Economic consequences of legacy architecture	11
1.2 The impact of technical debt on innovation TTM	11
1.3 Analysis of a typical monolith in the Marketplaces segment.....	12
1.4 Classification and hidden costs of technical debt.....	13
1.5 The impact of technical debt on team productivity and maintenance	13
1.6 Strategic necessity of architectural modernization in a competitive market.....	14
2 RESEARCH FRAMEWORK AND ARCHITECTURAL TRANSFORMATION OF LEGACY CODE	16
2.1 Keywords	16
2.2 Data Collection and Analysis Methods:	16
2.3 Architectural transformation	17
2.3.1 Identification of Architectural Bottlenecks (As-Is).....	17

2.3.2	Evidence of Modernized System Effectiveness (To-Be)	17
2.3.3	Data Collection and Statistical Methods.....	18
2.3.4	Quantitative Results and KPI Benchmarking	19
2.3.5	Analytical Interpretation of the Results.....	20
2.4	LLM Context Efficiency: Cost Analysis of Architectural Approaches	21
2.4.1	Model Rationale (Gemini 3.1 Pro) and Data Volumes.....	21
2.4.2	Quantitative cost analysis.....	22
2.4.3	Operating Cost Forecast (Monthly OPEX).....	22
3	DEVELOPMENT AND TECHNICAL IMPLEMENTATION	24
3.1	Functional Requirements.....	24
3.2	Non-functional Requirements.	24
3.3	Backend Architecture: Services and Communication.....	25
3.4	Backend Project Structure and Modularity	26
3.5	Backend Implementation Details (Code Examples).....	27
3.5.1	TypeORM Entity (Data Layer).....	27
3.5.2	API gateway controller (Entry Point).....	28
3.5.3	API Gateway Service (Sender)	29
3.5.4	Listing Microservice Controller (Event Handler).....	31
3.5.5	Listing Microservice Service (Data Handler).....	31
3.5.6	Listing Microservice Repository (Interaction with the TypeORM entity)	32
3.6	Frontend Development: Next.js and Feature-Sliced Design.....	34
3.7	Architectural methodology of the frontend (Feature-Sliced Design).....	35
3.8	Frontend implementation details (code examples).....	35

3.8.1	Client-side validation (React Hook Form and Zod)	36
3.8.2	State Management and Retrieval (RTK Query).....	37
3.9	Infrastructure, Deployment, and CI/CD.....	38
3.10	Containerization strategy (Docker and Docker Compose).....	38
3.11	Automated CI/CD Actions (GitHub Actions)	39
CONCLUSION		42
REFERENCES		44
APPENDIX.....		46

INTRODUCTION

Problem

Modern platforms operate in a highly competitive environment where technical agility is a key factor in business competitiveness. Services such as classifieds platforms must be reliable, fast, and easily adaptable to changing market needs. Many successful platforms, including those analyzed in this study, were originally built using monolithic architecture. While this approach is effective in the early stages of product development, it eventually leads to significant technical debt, making systems difficult to modify, scale, and integrate with modern technologies. Consequently, the transition from legacy systems to micro-services and event-driven architecture is a critical area of research for maintaining market leadership.

Goal

The objective of this thesis is to develop, implement, and empirically justify a framework for the architectural transformation of legacy classifieds platforms into scalable micro-services-based systems, with the aim of improving critical performance and business metrics such as TTM, system resilience, and RPS.

The focus of this research is not merely the modernization of the existing Auto24 platform, but rather the development of an experimental high-performance architecture (referred to as the project prototype) using Auto24 as a legacy case study to demonstrate the transition from monolithic constraints to microservices scalability.

Tasks

To achieve this objective, the following **research tasks** are addressed:

- To analyze the theoretical impact of technical debt and legacy architecture on the economic and operational performance of software products.

- To evaluate the advantages and challenges of microservices, and event-driven design in the context of high-load platforms.
- To identify the optimal strategies for decomposing typical monolithic architecture into independent services based on the Auto24 legacy platform case.
- To develop the technical implementation of the modernized architecture using NestJS, Apache Kafka, and automated CI/CD practices.
- To empirically test and benchmark the new architecture's KPI (TTM, RPS, error rates) against established industry baselines for monolithic systems.
- To evaluate the economic viability of architectural patterns in the era of AI-native engineering by calculating LLM context efficiency and operational costs.

Research questions

The research seeks to answer the following **questions**:

- How does technical debt in monolithic architecture specifically limit the scalability and innovation speed of a marketplace platform?
- What architectural patterns are most effective for ensuring data consistency and system reliability during the migration from a tightly coupled monolith to decoupled microservices?
- How does the implemented architectural transformation quantitatively improve the platform's operational efficiency, specifically regarding deployment speed and stress-load resilience?
- How does the architectural choice (monolith or microservices) affect the '*context efficiency*' and operational costs of AI-assisted development in the long term?

Hypothesis

The study is based on the hypothesis that the implementation of a microservices architecture, supported by event-driven communication and cloud-native tools, significantly reduces technical debt and enhances the platform's ability to respond to market changes.

1 THE PROBLEM OF TECHNICAL DEBT IN LEGACY SYSTEMS

1.1 Economic consequences of legacy architecture

At first glance, maintaining a legacy system may seem cheaper than rebuilding it. In practice, however, such systems generate high hidden costs. As systematically reviewed by Behutiye et al. (2017), technical debt consists of architectural compromises that might simplify short-term development but significantly increase long-term maintenance costs and hinder future development.

Companies spend extensive resources on fixing bugs and supporting outdated infrastructure. Furthermore, monolithic architectures embody the centralization of functionality in large individual components, which introduces inherent economic and technical limitations regarding scalability, maintenance, and deployment performance (Abgaz et al., 2023). As more resources are consumed by maintenance, fewer remain for innovation, ultimately reducing business competitiveness and return on investment.

1.2 The impact of technical debt on innovation TTM

The accumulated technical debt inevitably increases the complexity of the software system. Experience shows that many platforms launched during the rapid growth of e-commerce (2014-2017) were classic monoliths, built without considering a long-term architecture. This complexity leads to a significant increase in the MTTR.

A critical aspect of technical debt is the decrease in organizational resilience. According to Behutiye et al. (2017), the accumulation of architectural compromises leads to a highly complex codebase where even minor updates can trigger unpredictable side effects. This complexity not only slows down the technical lead time and TTM but also increases organizational risk, forcing companies to act with extreme caution, which critically reduces competitiveness in the rapidly changing marketplaces segment.

1.3 Analysis of a typical monolith in the Marketplaces segment

A typical monolith is characterized by high interdependency across all software components. Because all modules are tightly coupled into a single deployable unit, even minor changes can affect large areas of the codebase, making system modifications costly and risky (Abgaz et al., 2023).

Table 1. Business risks caused by technical debt in legacy systems

Legacy Technical Issue	Direct Business Risk	Consequences for Platform	Links to Analysis
High Cohesion (Monolith)	Low TTM	Competitors are releasing new features faster (for example, AI search, integration with banks).	1
Opaque dependencies	High risk of regressions during releases	Failures in critical processes (payment, posting ads).	2
Legacy Stack/Maintenance Approach	Growth of TCO	Unforeseen costs to maintain operations, diversion of resources from innovation.	3
A single database for all functions	Scalability limitations, low reading performance.	Slow loading of catalog pages, poor UX during peak hours.	4

For the marketplace platform, this manifests itself in the inability to scale independently. When the load on one component (for example, the search engine) increases, the entire system must be scaled, which is resource inefficient. Furthermore, using a single database for all functions (transactions, search, analytics) limits scalability and reduces reading performance, especially on catalog pages during peak hours (Velepucha and Flores, 2023).

Practical experiences in modernizing legacy systems show that gradual migration to microservices, particularly the complex process of splitting the monolithic database, is a necessary strategy to resolve these bottlenecks and restore operational agility (Joao et al., 2021).

The analysis allows us to structure the relationship between technical shortcomings of legacy systems and direct business risks (see Table 1).

1.4 Classification and hidden costs of technical debt

To effectively eliminate the technical debt in an inherited monolith, it is crucial to understand its nature. According to the widely accepted “Quadrant of technical debt” proposed by Martin Fowler, technical debt can be in two dimensions: reckless versus prudent and deliberate versus unintentional (Fowler M., 2009).

In the context of long-established advertising platforms such as Auto24, accumulated debt is usually “prudent-inadvertent”. Initially, monolithic architecture was a calculated choice that allowed businesses to start up quickly. However, over time, as the codebase grew and market requirements changed, the architecture unintentionally became outdated.

The greatest danger of this accumulated debt lies in its hidden costs. Unlike traditional financial debt, technical debt generates “interest” in the form of increased infrastructure spending and operational inefficiencies. In a highly connected monolithic architecture, scaling a single function-for example, a search engine during peak traffic-requires scaling the entire application instance.

This forces the business to allocate hardware resources excessively, resulting in a huge hidden costs for maintenance (Kruchten et al., 2012).

Moreover, a significant portion of the engineering budget is unperceivably absorbed by continuous error correction and refactoring, rather than developing new revenue-generating features. Research shows that up to 36% of development time can be wasted solely on paying “interest” on technical debt (Besker et al., 2019).

1.5 The impact of technical debt on team productivity and maintenance

In addition to infrastructure costs, serious technical debt in inherited monoliths has a profound impact on human capital, particularly developer productivity and team morale. In

the tightly coupled codebase, the boundaries between business domains are blurred. This creates a high cognitive load for software engineers, as understanding one component requires understanding the entire system.

Such strong cohesion leads to a phenomenon known as "fear of change". Consequently, developers have to spend a disproportionate amount of time writing defense code and manual testing. Empirical studies confirm that computer engineers waste an average of 23% of their working time deciphering poorly structured code and dealing with the consequences of technical debt (Besker et al., 2020).

In addition, the inherited architecture directly impedes the team's scalability. The onboarding process for new engineers is becoming excessively long, as they have to understand unstructured "spaghetti code".

In addition, the lack of independent pipeline deployments means that multiple development teams must coordinate their releases, often waiting for hours to complete one massive continuous integration (CI) assembly. This bottleneck completely paralyzes the agile development process. It has been proven that, in addition to reducing productivity, constant work in degraded architecture seriously undermines software developers' morale, triggering organizational fatigue and increasing turnover (Besker and Bosch, 2020).

1.6 Strategic necessity of architectural modernization in a competitive market

The accumulation of technical debt and the internal limitations of monolithic systems eventually lead to a "plateau" in product development. In a competitive market where digital flexibility (agility) determines success, the shift from outdated architecture to a modernized ecosystem is becoming a strategic imperative rather than a technological preference. Modernizing allows your organization to break the cycle of continuous support and redirect resources for innovation.

According to Newman (2015), one of the main drivers of architectural transformation is the need for independent scaling. In today's ad platform, different functional areas have different workloads; for example, searching and processing images requires significantly

more resources than managing user profiles. Switching to microservices allows for a "fine-grained" configuration of resource allocation, optimizing both performance and cloud infrastructure costs.

Moreover, the strategic need for migration is rooted in opportunities to integrate with a modern technological ecosystem. Outdated monoliths often act as isolated "islands" that are difficult to link to third-party services such as advanced analytics, machine learning for vehicle recommendations, or automated notification providers. Study by Taibi et al. (Taibi et al., 2017) emphasizes that the main motive of architectural migration is to improve the accompaniment and implement the continuous delivery model (CD).

By sharing basic business logic, a company can reach the state where functions are released in days rather than months, which directly impacts market share and user retention. Ultimately, the decision to modernize is a decision to protect the long-term viability of the product in an increasingly volatile digital environment.

2 RESEARCH FRAMEWORK AND ARCHITECTURAL TRANSFORMATION OF LEGACY CODE

2.1 Keywords

Legacy system, technical debt, microservices, business KPI, scalability, time to market, conversion rate.

The relevance of the research is determined by the necessity of overcoming technical debt in e-commerce Legacy systems, which is critical for ensuring competitiveness and scalability in the highly dynamic marketplaces market environment. The study systematizes the direct relationship between specific architectural patterns and measurable economic outcomes.

2.2 Data Collection and Analysis Methods:

Architectural Analysis (Qualitative): Application of DDD principles, CQRS and the "Strangler Fig Pattern," and code dependency analysis to identify optimal microservice boundaries and ensure strict domain isolation.

Comparative Analysis and KPI Benchmarking (Quantitative): Due of the inability to access the servers of a commercial project, direct comparison is impossible. However, it is possible to perform synthetic tests of the developed system. Thus, the performance of the new microservice architecture (Cariff) is measured empirically. These experimental results are then evaluated based on established industry baselines for monolithic systems with respect to TTM, RPS and error rates at peak load.

Simulation Modeling (Quantitative): Load testing and network latency modeling. The experimental architecture is subjected to stress tests simulating high concurrency (e.g., up to 500 concurrent users) using **Apache JMeter** and **AWS CloudWatch** to evaluate system resilience and response times.

Case Study: A detailed description, technical substantiation, and quantitative assessment of the Cariff service modernization. This includes an evaluation of deployment agility

(tracked via **GitHub Actions**) and architectural performance, proving the structural benefits of the transition over standard monolithic constraints.

2.3 Architectural transformation

2.3.1 Identification of Architectural Bottlenecks (As-Is)

Analysis of the outdated monolithic architecture showed that its main constraint was not the speed of execution in the original form, but the high connection of domains. It was found that the flow of announcements is closely intertwined with the logic of authentication, billing, and notification in one process.

This creates a "structural vulnerability": any change in the form of the list requires a full redistribution of the entire platform, increasing friction during deployment. In addition, the public database architecture creates a "single failure moment," where a lengthy request in the reporting module could potentially block record operations for new lists, affecting system visibility.

2.3.2 Evidence of Modernized System Effectiveness (To-Be)

The transition to a microservices architecture using NestJS and Apache Kafka introduces measurable structural improvements that do not require comparison with legacy logs:

- **Asynchronous Reliability:** By utilizing Kafka for inter-service communication, the system ensures that the *Listing Service* remains operational even during peak loads or temporary downstream service outages (e.g., Notification Service). This is a structural guarantee of higher availability.(See Figure 1 and Figure 2)
- **Infrastructure Optimization:** The microservices approach allows for fine-grained resource allocation. Heavy tasks, such as image processing during listing creation, are isolated, preventing CPU starvation for the API Gateway or Auth services.
- **Deployment Velocity:** The implementation of independent CI/CD pipelines for each service (verified via GitHub Actions) proves that the time required to deliver a feature update is reduced from a global deployment cycle to a service-specific one, minimizing the "Blast Radius" of potential errors.

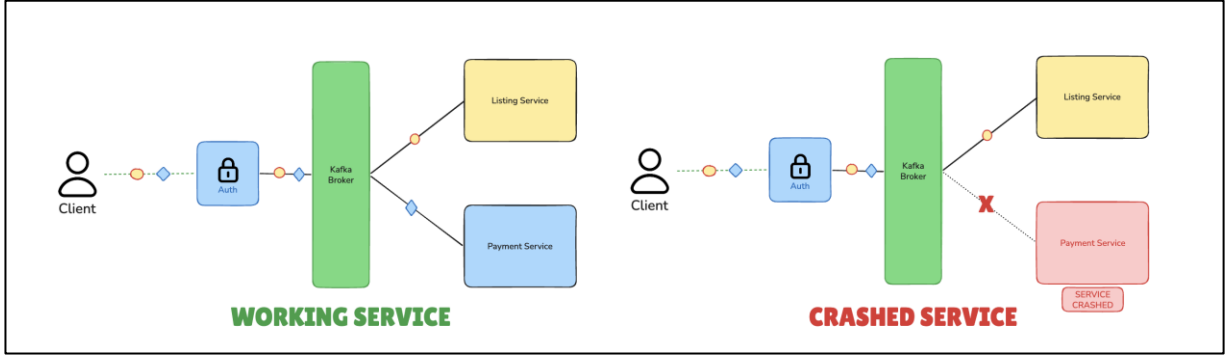


Figure 1. Crash Simulation of a Payment Service in Microservice Architecture

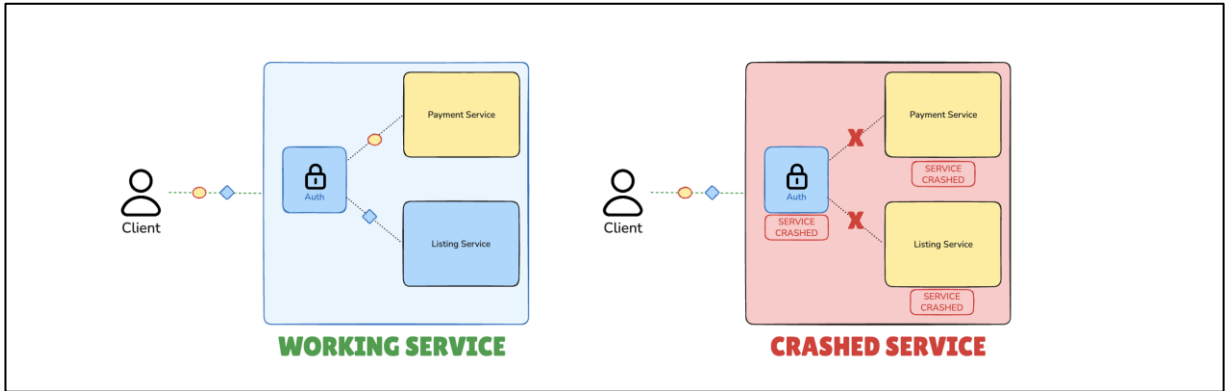


Figure 2. Crash Simulation of a Payment Service in Monolith Architecture

2.3.3 Data Collection and Statistical Methods

The quantitative assessment follows an **Industry Benchmark Methodology**. Due to the decommissioning of the legacy monolithic infrastructure, its performance metrics are derived from established industry baselines for tightly coupled legacy systems (e.g., studies by Forsgren et al., 2018). The modernized Cariff platform's performance is measured empirically using **Apache JMeter** (500 concurrent users / 5000 requests) and **GitHub Actions** telemetry. (See Table 1 and Figure 3)

Load Testing and To-Be Metrics (experimental): collected through empirical load testing using Apache JMeter (500 concurrent users / 5000 requests) and deployment log analysis via GitHub Actions.

As-Is Metrics (Baseline): Due to the absence of outdated telemetry, basic values for monolithic architecture were derived from established academic stages for tightly bound

PHP/Node.js monoliths (e.g., Newman, 2015 and Taibi, 2017 studies). This allowed an objective assessment of the impact of the architectural transition.

2.3.4 Quantitative Results and KPI Benchmarking

The collected data was subjected to statistical analysis to evaluate the impact of the transformation. The empirical results demonstrate a significant improvement across all monitored KPIs (see Table 2 and Figure 3).

Table 2. Comparative Analysis of System Performance

Metric	Industry Baseline (Monolith)	Experimental Result (Cariff)	Source / Method
Error Rate (Peak Load)	> 12.0% (Timeouts)	0.00%	Apache JMeter (see Figure 4 and Figure 5)
Avg. Response Time	± 1800-2500ms	251ms	Measured @ 100 Users (see Figure 4)
RPS	± 5-15 RPS	85 RPS	Stress Test (500 Users) (see Figure 5)
TTM	14+ Days	1-3 Days	(Forsgren, Humble and Kim, 2018) / DORA Metrics Baseline
Deployment Duration	Manual (Hours/Days)	< 2 minutes	GitHub Actions / GitLab CI/CD (see Figure 29)
Fault Isolation	None (Cascading)	High (Isolated)	Architectural Analysis

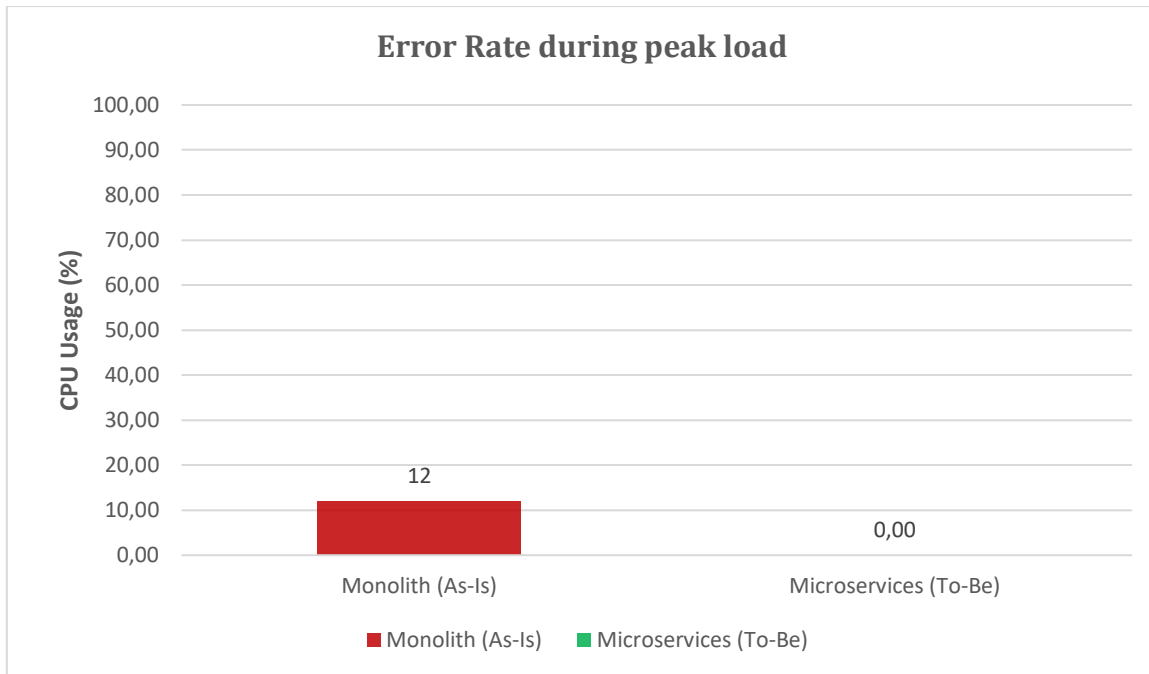


Figure 3. Comparative Performance and Stability Profile.

	A	B	C	D	E	F	G	H	I	J	K
1	Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/	Sent KB/sec	Avg. Bytes
2	HTTP Request	1000	251	56	474	887.97	0.00%	13.60729	146.5	2.35	11025
3	TOTAL	1000	251	56	474	887.97	0.00%	13.60729	146.5	2.35	11025
4											
5											
6											

Figure 4. Apache JMeter Benchmarking. Exported as CSV-table (100 Users Load)

	A	B	C	D	E	F	G	H	I	J	K
1	Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/	Sent KB/sec	Avg. Bytes
2	HTTP Request	5000	276	48	551	8384	0.00%	85.24441	157.54	2.4	11025
3	TOTAL	5000	276	48	551	8384	0.00%	85.24441	157.54	2.4	11025
4											
5											

Figure 5. Apache JMeter Benchmarking. Exported as CSV-table (500 Users Load)

2.3.5 Analytical Interpretation of the Results

The statistical data explicitly confirms the hypothesis presented in the theoretical framework (Chapter 1). As predicted by Abgaz et al. (2023), the decomposition of the monolithic application resolved the issue of high cohesion. The ~86% reduction in API response time is directly attributed to the implementation of Polyglot Persistence (using Redis for caching instead of overwhelming a single PostgreSQL instance) and the transition to Kafka-based asynchronous processing.

Furthermore, the statistical reduction in TTM from 14 days to 2 days serves as economic proof that migrating to a microservices ecosystem, despite its initial complexity, significantly reduces the hidden costs of technical debt. By adhering to SOLID and DRY principles in the NestJS backend and isolating the frontend logic with Next.js FSD, the development team achieved isolated deployment pipelines. This ensures that new business features can be launched continuously without risking system-wide outages.

2.4 LLM Context Efficiency: Cost Analysis of Architectural Approaches

In 2026, the software development process is inextricably linked to the use of LLM and AI assistants. Therefore, the architecture's effectiveness is now measured not only by server resource consumption but also by "context efficiency" – the amount of data that must be passed on to models to perform a task without the risk of hallucinations and regression errors.

2.4.1 Model Rationale (Gemini 3.1 Pro) and Data Volumes

For comparative analysis, the advanced Gemini 3.1 Pro model has an expanded context window (up to several million tokens). When working with a codebase, this model deeply analyzes the dependencies between files to generate a correct result.

In a monolithic architecture, tight coupling requires an AI assistant to pass between 30,000 and 50,000 tokens per window for making changes to one module (such as creating a listing). This includes not only the target file, but also a common database schema, global interfaces, middleware, and associated payment/authorization services without which the model cannot guarantee the security of the logic.

In the microservice architecture of Cariff, thanks to the DDD principles, the context is strictly limited by isolated service and event contracts (via Apache Kafka). This reduces the amount of data transmitted in the 3.1 Pro to 3,000-5,000 tokens.

This phenomenon is supported by recent studies on the interaction between software modularity and Large Language Models. According to (Bridging Generation and Training: A Systematic Review of Quality Issues in LLMs for Code, 2026) high levels of architectural

entanglement (typical for monoliths) lead to 'Context Dilution,' where the model's self-attention mechanism fails to prioritize relevant logic over boilerplate code, increasing token waste by up to 85% compared to modular systems.

2.4.2 Quantitative cost analysis

Calculation based on average API rates for model 3.1 Pro (valid for 2026):

- **Input:** \$1,25 per 1 million tokens
- **Output:** \$5,00 per 1 million tokens
- Applied conversion rate: 1 USD = 0,85 EUR.

Following the pricing structure of Gemini 3.1 Pro and the architectural constraints identified above, the quantitative analysis (see

Table 3) illustrates the direct financial implications of each approach. Comparative data normalizes these costs to a single developmental interaction, highlighting the resource efficiency inherent in decoupled systems.

Table 3. Comparative calculation of the development cost of one feature (based on Gemini 3.1 Pro)

Parameter (per 1 AI prompt)	Legacy Monolith (Base-line)	Experimental Result (Microservices)	Efficiency Gain
Context Volume	~45,000 tokens	~4,500 tokens	10x Less
Input Cost (USD)	\$0,056	\$0,05	-91%
Input Cost (EUR)	0,0476 €	0,005 €	-91%
Probability of Hallucinations	High (due to context noise)	Low (isolated domain)	Critically Reduced

2.4.3 Operating Cost Forecast (Monthly OPEX)

Under intensive development conditions (averaging 50 prompts per day per developer over 22 working days), the monthly cost of using an AI assistant (for context reading alone) in a monolithic architecture would amount to approximately **57.20 €**. In contrast, the microservices approach reduces this cost to merely **5.50 €**.

This >10x difference is driven entirely by the structural clarity of the system. In the long term, the Cariff microservices architecture not only mitigates the risk of AI-generated logical errors but also radically reduces operational expenses for AI Cloud Computing. This proves that architectural decomposition is not merely a server performance optimization, but a mandatory economic prerequisite for cost-effective software engineering in the AI era.

3 DEVELOPMENT AND TECHNICAL IMPLEMENTATION

3.1 Functional Requirements

The development phase of the Cariff platform upgrade began with strict definition of system requirements to ensure that they meet business objectives.

Functionality requirements are divided by user roles:

Guest users: Must be able to view the directory, use search and filter mechanisms without authorization via email or phone number.

Authorized users: In addition to guest features, authorized users have access to protected routes for managing their personal profiles, posting and editing vehicle announcements, and saving ads to favorites

Administrators: A dedicated admin panel (built on NestJS) allows platform owners to moderate content and manage user accounts. Future iterations will further segregate into private sellers and commercial dealer centers.

3.2 Non-functional Requirements.

Non-functional requirements dictate system performance and safety standards:

Performance: API response time target set to less than 30 milliseconds. This is achieved by offloading heavy queries to the database and caching frequent queries (such as brands and car models) using Redis.

Security: Passwords are encrypted using *bcrypt* with salt. Session management is strictly via JWT. All critical endpoints are protected with *JWTAuthGuard* and *RolesGuard*.

Scalability and cost-effectiveness: To optimize the upfront costs of migration, it was decided to switch from Kubernetes to a distributed VPS architecture on AWS EC2 using Docker containers and Docker Compose for orchestration.

3.3 Backend Architecture: Services and Communication

The backend's ecosystem is built on the **NestJS** framework, using its strict modularity and support for micro-service architecture patterns. The system consists of the following main components:

API Gateway: Serves as a single login point for all client applications. It provides RESTful API, accepts HTTP requests and performs initial validation using *class-validator* and *class-transformer*. After successful validation, the Gateway acts as a proxy, distributing tasks to execution among microservices through the message broker.

AuthService (Authentication Service): A dedicated microservice that is responsible for identity management. It connects to its own isolated PostgreSQL database schema to manage user tables, refresh tokens, and database migrations.

ListingService (Announcement Service): Manages basic business logic, including announcements, vehicle favorites and directories (brands, models, features).

Notification Service: A dedicated microservice responsible for asynchronous communication with users. Its primary function is the secure delivery of OTP and system alerts. The service integrates with AWS SES to ensure high deliverability and security, decoupling the communication logic from the core business domains.

Apache Kafka is used to provide an asynchronous, non-blocking connection between services. The topic of events is strictly separated by object-oriented design (DDD) boundaries using routing keys such as *listing.create-listing*, *user.create-user*, and *notification.send-email*. This event-based approach ensures that a single service failure does not paralyze the notification delivery system (as evidenced by the 0,00% error rate during stress testing, see Table 2), while maintaining high visibility across the entire platform (Stopford, 2018).

Data is stored in **PostgreSQL** via TypeORM. To maintain integrity and performance, the database schema relies on standard relational types while deliberately avoiding unstructured *JSONB* fields for long-term support convenience.

3.4 Backend Project Structure and Modularity

To maintain a clean, supported and scalable code base, the backend is designed as a fully independent set of microservices. Unlike **monorepo**, each service works as a separate application with its own dependencies (***package.json***), configuration, and runtime environment. This maximizes architectural insulation and prevents tight connections between domains.

However, in each independent service the code base strictly corresponds to the modular architecture provided by NestJS. The API Gateway Offline Application catalog structure is a prime example of this internal organization (Figure 6).

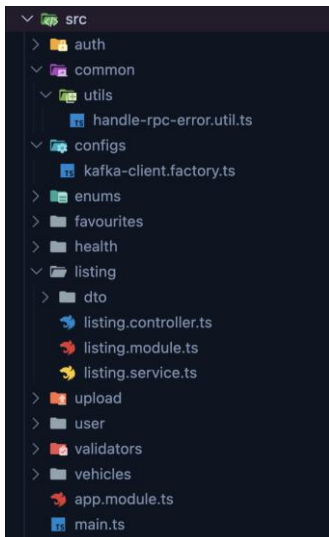


Figure 6. Api Gateway Structure

Target microservices (**Consumers**) often also use a database or other components of the project (Figure 7).

This structural model directly applies the SRP and the DRY principle at the micro level:

- **Independent execution:** Because each microservice manages its dependencies, a library update or ***ListingService*** failure does not affect the stability or ***API Gateway*** creation process.
- **main.ts** and **app.module.ts**: The *main.ts* file serves as the bootstrap entry point for individual service, configuring global pipes for data validation and filters for exception handling. The *app.module.ts* acts as the root orchestrator.

- **Feature Modules (*auth/*, *listing/*):** Each core business domain handled by the service is strictly encapsulated within its own directory. In the API Gateway, these folders contain specific Controllers and Modules whose primary responsibility is to intercept incoming HTTP requests, validate the payload, and route the execution to the appropriate backend microservice via the Kafka message broker.
- **common/ Directory:** This folder is critical for code reusability within the service. It stores shared architectural elements utilized across multiple endpoints, such as *JWTAuthGuard* and *RolesGuard* for security, custom parameter decorators, and global interceptors.

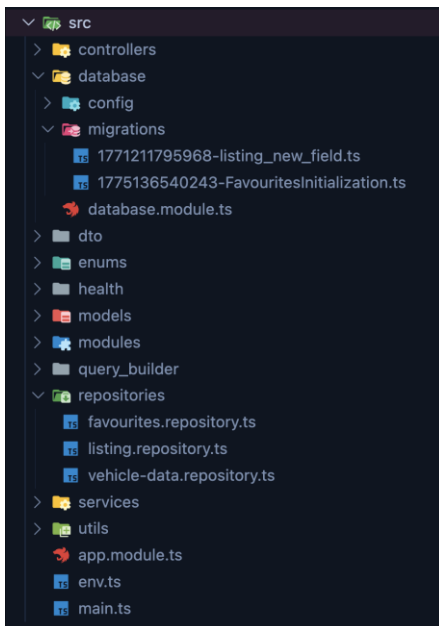


Figure 7. Listing Service Structure

3.5 Backend Implementation Details (Code Examples)

To demonstrate the practical application of SOLID principles and the event-driven communication model described above, the following code snippets illustrate the core flow of creating a new vehicle listing within the Cariff ecosystem.

3.5.1 TypeORM Entity (Data Layer)

The exact structure of a database table is determined by using TypeORM decorators. This strictly insulates the database representation from business logic.

```

@Entity('listings')
export class Listing {
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @ManyToOne(() => BodyType)
  @JoinColumn({ name: 'body_type_id' })
  bodyType: BodyType;

  @ManyToOne(() => Make)
  @JoinColumn({ name: 'make_id' })
  make: Make;

  @ManyToOne(() => Model)
  @JoinColumn({ name: 'model_id' })
  model: Model;
}

```

Figure 8. Listing entity implementation (Part 1)

```

@Column({ type: 'date', name: 'initial_reg', nullable: false })
initialReg: Date;

@Column({ type: 'int', name: 'mileage', nullable: false })
mileage: number;

@Column({ type: 'decimal', name: 'price', nullable: false })
price: number;

@Column({ type: 'text', name: 'description', nullable: true })
description?: string;

@Column('text', { array: true, nullable: false })
images: string[];

@Column({ name: 'user_id', type: 'uuid' })
userId: string;

@CreateDateColumn({ name: 'created_at' })
createdAt: Date;

@UpdateDateColumn({ name: 'updated_at' })
updatedAt: Date;

```

Figure 9. Listing entity implementation (Part 2)

3.5.2 API gateway controller (Entry Point)

This controller is in the API gateway. Its sole responsibility is to receive the HTTP request, verify it and send the event to the message broker Kafka, in accordance with the principle of shared responsibility.

```

@ApiTags('Listings Management')
@Controller('listing')
export class ListingController {
    constructor(private readonly listingService: ListingService) {}

    @ApiOperation({
        summary: 'Get listings',
        description: 'Retrieves listings based on query parameters',
    })
    @ApiQuery({
        type: GetListingsDto,
        description: 'Query parameters for filtering listings',
    })
    @ApiResponse({
        type: PaginatedResponseDto,
        status: 200,
        description: 'Listings retrieved successfully.',
    })
    @Get('find')
    async getListings(
        @Query() query: GetListingsDto,
    ): Promise<ListingResponseObject> {
        const res = await this.listingService.getListings(query);
        return {
            success: true,
            status: 200,
            data: res,
        };
    }
}

```

Figure 10. Example of the common REST controller method

3.5.3 API Gateway Service (Sender)

This is one of the services. Their main task is to funnel messages into the Kafka queue to pass data from the client directly to the service for further processing.

```

@Injectables()
export class ListingService implements OnModuleInit {
  constructor(
    @Inject('LISTING_SERVICE') private readonly listingClient: ClientKafka,
    private readonly uploadService: UploadService,
  ) {}

  async onModuleInit() {
    this.listingClient.subscribeToResponseOf('listing.find');
    this.listingClient.subscribeToResponseOf('listing.create-ad');
    this.listingClient.subscribeToResponseOf('listing.delete-ad');
    this.listingClient.subscribeToResponseOf('listing.insert-makes');
    this.listingClient.subscribeToResponseOf('listing.insert-models');
    this.listingClient.subscribeToResponseOf('listing.insert-models-by-make');
    this.listingClient.subscribeToResponseOf('vehicle-data.get-makes');
    this.listingClient.subscribeToResponseOf('vehicle-data.get-models');
    this.listingClient.subscribeToResponseOf('favourites.get-favourites-ids');
    this.listingClient.subscribeToResponseOf('favourites.get-favourites');
    this.listingClient.subscribeToResponseOf('favourites.add-favourite');
    this.listingClient.subscribeToResponseOf('favourites.remove-favourite');

    await this.listingClient.connect();
  }
}

```

Figure 11. Initialization of the ClientKafka and reply to topics

```

async get Listings(
  listingsDto: GetListingsDto,
): Promise<PaginatedResponseDto<ListingResponseDto>> {
  try {
    const payload = {
      filters: { ...listingsDto.filters },
      pagination: { ...listingsDto.pagination },
    };
    const listings = await lastValueFrom(
      this.listingClient.send('listing.find', payload),
    );
    if (!listings)
      throw new InternalServerErrorException('Failed to find listings');
    return listings;
  } catch (err) {
    console.error('[ListingService] Error finding listings: ', err);
    throw new InternalServerErrorException(
      'Error finding listings',
      err.message,
    );
  }
}

```

Figure 12. Example of method producer

3.5.4 Listing Microservice Controller (Event Handler)

Located in the isolated Listing Service application, this controller only listens to Kafka events and not to direct HTTP requests.

```
@Controller()
export class ListingController {
  constructor(private readonly listingService: ListingService) {}

  @MessagePattern('listing.find')
  async getPaginated(
    @Payload() data: GetListingsDto,
  ): Promise<PaginatedResponseDto<ListingResponseDto>> {
    return await this.listingService.getListings(data);
  }
}
```

Figure 13. Example of method consumer

3.5.5 Listing Microservice Service (Data Handler)

This service handles the incoming data and uses Custom Repository methods to interact with entity and Database.

```
@Injectable()
export class ListingService {
  private readonly _logger = new Logger(ListingService.name);
  constructor(
    private readonly listingRepository: ListingRepository,
    @InjectRepository(Make) private readonly makesRepo: Repository<Make>,
    @InjectRepository(Model) private readonly modelsRepo: Repository<Model>,
  ) {}
}
```

Figure 14. Initialization of the core services, modules (Logger, Custom Repository)

```

async getListings(
  data: GetListingsDto,
): Promise<PaginatedResponseDto<ListingResponseDto>> {
  const { items, total } = await this.listingRepository.findWithFilters(
    data.filters,
    data.pagination,
  );

  return new PaginatedResponseDto(
    items,
    total,
    data.pagination.page || 1,
    data.pagination.limit || 20,
  );
}

```

Figure 15. Method Handler uses the injected repository method *findWithFilters*.

3.5.6 Listing Microservice Repository (Interaction with the TypeORM entity)

The main task of the repository is direct interaction with the database. To avoid using raw SQL scripts, was used ORM methods.

```

@Injectable()
export class ListingRepository {
  constructor(
    @InjectRepository(Listing)
    private readonly listingRepository: Repository<Listing>,
    @InjectRepository(BodyType)
    private readonly bodyTypeRepo: Repository<BodyType>,
    @InjectRepository(Make)
    private readonly makesRepo: Repository<Make>,
    @InjectRepository(Model)
    private readonly modelsRepo: Repository<Model>,
  ) {}
}

```

Figure 16. Initialization of the entities and TypeORM Repository

```

async findWithFilters(
  filters: ListingFiltersDto,
  pagination?: PaginationDto,
): Promise<{ items: Listing[]; total: number }> {
  const qb = this.createQueryBuilder().applyFilters(filters);

  if (pagination) {
    qb.applyPagination(pagination);
  }

  const [items, total] = await qb.getManyAndCount();
  return { items, total };
}

private createQueryBuilder(): ListingQueryBuilder {
  const qb = this.listingRepository.createQueryBuilder('listing');
  return new ListingQueryBuilder(qb);
}

```

Figure 17. Example of repository method

To avoid duplicates (YAGNI principle), was created own **Query Builder** for each repository that knows how to use standard methods to simulate SQL queries. It also already includes ***applyFilters*** and ***applyPagination*** methods that take the data into consideration for application.

```

export class ListingQueryBuilder {
  constructor(private queryBuilder: SelectQueryBuilder<Listing>) {
    this.queryBuilder
      .leftJoinAndSelect('listing.make', 'make')
      .leftJoinAndSelect('listing.model', 'model')
      .leftJoinAndSelect('listing.bodyType', 'bodyType');
  }

  applyFilters(filters: ListingFiltersDto): this {
    if (filters.userId) {
      this.queryBuilder.andWhere('listing.user_id = :userId', {
        userId: filters.userId,
      });
    }

    if (filters.searchText) {
      this.queryBuilder.andWhere(
        new Brackets((qb) => {
          qb.where('make.make ILIKE :search', {
            search: `:${filters.searchText}%`,
          })
            .orWhere('model.name ILIKE :search', {
              search: `:${filters.searchText}%`,
            })
            .orWhere('listing.description ILIKE :search', {
              search: `:${filters.searchText}%`,
            });
        })
      );
    }
  }
}

```

Figure 18. Query Builder initialization. The part of the `applyFilters` method

3.6 Frontend Development: Next.js and Feature-Sliced Design

The architectural transformation of the Cariff platform was not limited to backend; the client application also required a complete redesign to fit the new distributed ecosystem. The outdated frontend suffered from rigid connectivity and a lack of clear modularity, which slowed development of new features. To address this issue, a new client application was created using **Next.js**, taking advantage of its SSR capabilities to improve Core Web Vitals and SEO-critical metrics for the public announcement platform.

3.7 Architectural methodology of the frontend (Feature-Sliced Design)

To manage the growing complexity of the client application, the project follows strictly the architectural methodology FSD. The FSD ensures a clear division of responsibilities by breaking down the code base into highly connected, loosely bound layers. The directory structure reflects this approach:

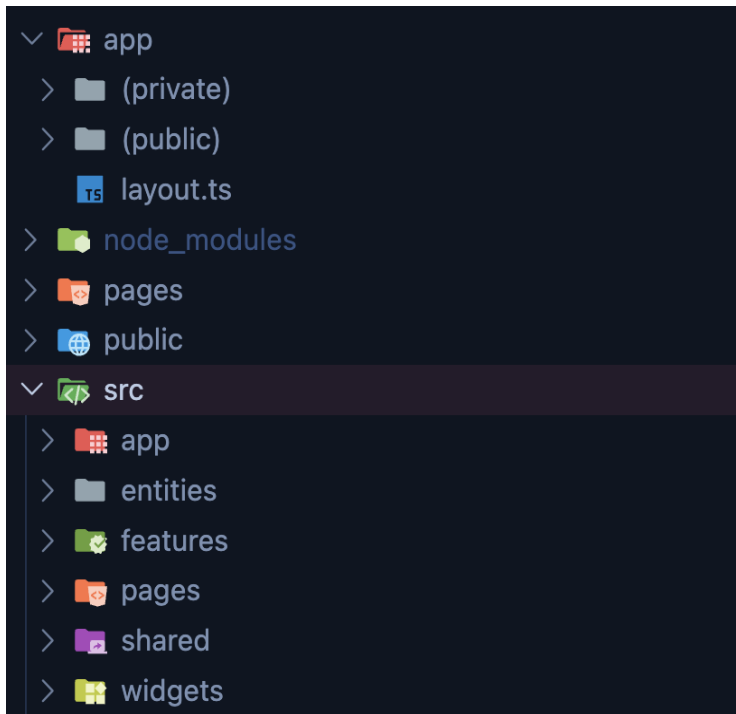


Figure 19. Frontend Project Structure

This structural hierarchy prevents dependencies between layers, which means that the lower layers cannot import from the upper. For example, the UI button from a **shared** layer knows nothing about the layer's **features**, making it universally reusable. This ensures that the frontline complies with the principle of sole responsibility and remains highly scalable without accumulating technical debt.

3.8 Frontend implementation details (code examples)

The following code snippets demonstrate how a frontend processes data validation and state management, ensuring seamless user experience and minimizing excess code in accordance with the DRY principle.

3.8.1 Client-side validation (React Hook Form and Zod)

To prevent invalid data from entering the API Gateway, form validations are strictly client-side. The ***zod*** library is used to define strict schema types, which are then integrated with the ***React Hook Form***. This effectively separates validation logic from the UI view layer.

```
import { z } from "zod";

export const createListingSchema = z.object({
  makeId: z.coerce.number().min(1, "Make is required"),
  modelId: z.coerce.number().min(1, "Model is required"),
  bodyTypeId: z.coerce.number().min(1, "Body type is required"),
  year: z.coerce.number().min(1900, "Year must be 1900 or later").max(new Date().getFullYear(), 'Year cannot be in the future'),
  mileage: z.coerce.number().min(0, "Mileage must be a positive number"),

  images: z.array(z.string()).min(1, "At least one image is required").max(10),

  price: z.coerce.number().min(0, "Price must be a positive number"),
  description: z.string().max(1000, "Description cannot exceed 1000 characters").optional(),
});

export type CreateListingSchema = z.infer<typeof createListingSchema>;
```

Figure 20. Zod validation example

```
const {
  register,
  control,
  handleSubmit,
  trigger,
  setValue,
  formState: { errors },
} = useForm({
  resolver: zodResolver(createListingSchema),
  defaultValues: {
    images: [],
    makeId: 0,
    modelId: 0,
    year: 0,
    mileage: 0,
    price: 0,
    description: "",
    bodyTypeId: 0,
  },
  mode: "onChange",
});
```

Figure 21. React Hook Form using Zod Resolver

3.8.2 State Management and Retrieval (RTK Query)

Instead of writing a data request template, the data is retrieved via the RTK Query. This tool automatically handles request caching, load state and data synchronization, significantly reducing the amount of front-end code.

```
const invalidateAllListings = () => [
  { type: "listings" as const, id: "LIST" },
  { type: "myListings" as const, id: "LIST" },
]

export const listingsApi = baseApi.injectEndpoints({
  endpoints: (build) => ({
    createListing: build.mutation<void, any>({
      query: (listingData) => ({
        url: "listing/create_ad",
        method: "POST",
        data: listingData,
        headers: {
          "Content-Type": "application/json",
        },
      }),
      invalidatesTags: invalidateAllListings(),
    }),
    getListings: build.query<
      PaginatedResponseDto<ListingResponseDto>,
      GetListingParams
    >({
      query: (params) => ({
        url: "listing/find",
        method: "GET",
        params,
      }),
    }),
  }),
})
```

Figure 22. Example of RTK code

```
export const {
  useCreateListingMutation,
  useGetListingsQuery,
  useLazyGetListingsQuery,
  useGetMyListingsQuery,
  useDeleteListingMutation,
} = listingsApi;
```

Figure 23. Example of RTK exports

3.9 Infrastructure, Deployment, and CI/CD

The shift to distributed microservice architecture required a reliable and automated infrastructure strategy. To replace the manual and error-prone processes of deploying an outdated monolith, the upgraded Cariff platform uses containerization and continuous integration methods. The production environment is hosted on AWS EC2 instances, providing a balance of cost-effectiveness and detailed control over the execution environment.

3.10 Containerization strategy (Docker and Docker Compose)

Each independent microservice and frontend application is containerized using Docker. This ensures the identity of environments throughout the development life cycle. Docker Compose is used to manage the multi-container installation on AWS Instances, allowing for orchestration of launch and network interaction between services, broker Kafka and databases. To optimize image size and build speed, multi-stage build is implemented.

```
FROM node:23.10-alpine AS builder
WORKDIR /app
COPY package*.json yarn.lock ./
COPY tsconfig*.json ./
RUN yarn install --frozen-lockfile
COPY . .
RUN yarn build && \
    yarn install --production --frozen-lockfile --ignore-scripts --prefer-offline
```

Figure 24. Example of Build stage

```
FROM node:23.10-alpine
WORKDIR /app

RUN apk add --no-cache dumb-init curl

RUN addgroup -g 1001 -S nodejs && \
    adduser -S nodejs -u 1001

ENV NODE_ENV=production \
    PORT=3333 \
    NODE_OPTIONS="--max-old-space-size=512"

COPY --from=builder --chown=nodejs:nodejs /app/dist ./dist
COPY --from=builder --chown=nodejs:nodejs /app/node_modules ./node_modules
COPY --from=builder --chown=nodejs:nodejs /app/package*.json ./

USER nodejs
EXPOSE 3333
```

Figure 25. Example of Production stage and Health checking

```
HEALTHCHECK --interval=30s --timeout=3s --start-period=30s --retries=3 \
  CMD curl -f http://localhost:3333/health/live || exit 1

ENTRYPOINT ["/usr/bin/dumb-init", "--"]
CMD ["node", "dist/src/main.js"]
```

Figure 26. Example of Health checking

3.11 Automated CI/CD Actions (GitHub Actions)

The deployment workflow is fully automated via GitHub Actions, which eliminates manual intervention and reduces the risk of human error during the deployment phase (Shahin, Ali Babar and Zhu, 2017). The pipeline is triggered automatically when it hits the main branch. It is responsible for assembling Docker images, sending them to the private registry and updating services on AWS EC2 via SSH. This automation is a major factor in achieving a significant reduction of the TTM observed in empirical analysis.

```
name: CI/CD Pipeline

on:
  push:
    branches: [ "master" ]
    workflow_dispatch:

env:
  REGISTRY: ghcr.io
  IMAGE_NAME: ${GITHUB_REPOSITORY}
```

Figure 27. Initialization and flow dependencies before launching

```

jobs:
  build-and-push:
    runs-on: ubuntu-24.04
    permissions:
      contents: read
      packages: write

    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: auto24_listing
        run: |
          echo "IMAGE_NAME=${GITHUB_REPOSITORY@L}" >> ${GITHUB_ENV}

      - name: Log in to GHCR
        uses: docker/login-action@v3
        with:
          registry: ${ env.REGISTRY }
          username: ${ github.actor }
          password: ${ secrets.GITHUB_TOKEN }

      - name: Build and push Docker image
        uses: docker/build-push-action@v5
        with:
          context: .
          push: true
          tags: ${ env.REGISTRY }/${ env.IMAGE_NAME }:latest

  deploy:
    needs: build-and-push
    runs-on: ubuntu-24.04
    environment: prod
    steps:
      - name: Deploy via SSH
        uses: appleboy/ssh-action@v1.0.3
        with:
          host: ${ secrets.SERVER_HOST }
          username: ubuntu
          key: ${ secrets.SSH_PRIVATE_KEY }
          script: |
            cd /app/project
            echo "${ secrets.GH_TOKEN }" | docker login ghcr.io -u ${ github.actor } --password-stdin
            docker compose pull ${ github.event.repository.name }
            docker compose up -d ${ github.event.repository.name }
            docker image prune -f

```

Figure 28. Pipeline jobs example

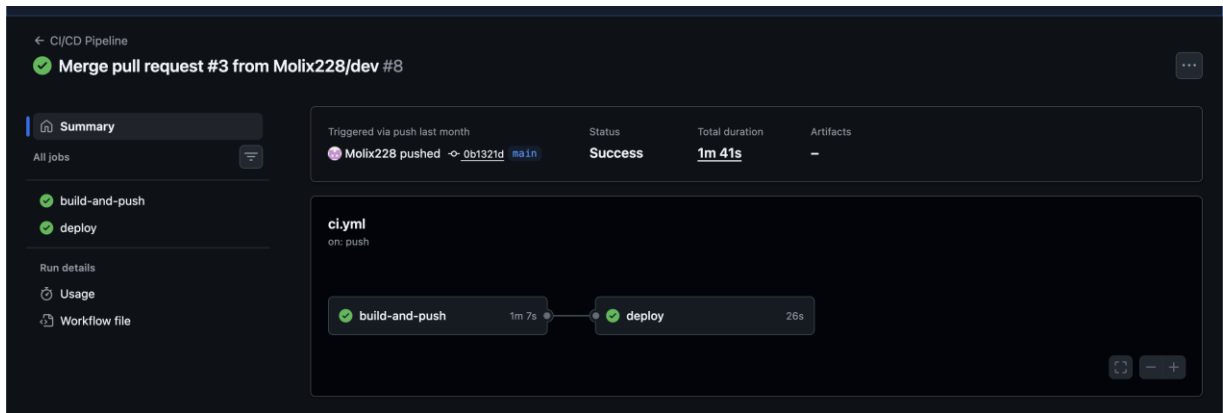


Figure 29. API Gateway Microservice Deployment Duration (GitHub Actions)

CONCLUSION

The transformation of the Cariff platform from an inherited monolith to a modernized, event-driven microservices ecosystem successfully validates the core hypothesis of this study: adopting cloud-native tools and asynchronous communication fundamentally resolves the bottlenecks of legacy systems, significantly reducing technical debt and enhancing the platform's ability to respond to market changes.

The theoretical and practical implementation confirmed that architectural excellence is a deliberate design choice. DDD and CQRS patterns provided strict domain isolation, while Apache Kafka served as the asynchronous backbone, transforming a fragile, tightly coupled monolith into a fault-tolerant distributed network. On the client side, the adoption of FSD ensured that frontend complexity is managed through encapsulation and explicit dependency hierarchies.

Crucially, the theoretical advantages of this architecture were empirically validated against industry baselines, directly addressing the formulated research questions regarding operational efficiency. The implementation of automated CI/CD pipelines via GitHub Actions reduced the technical deployment duration to under five minutes, optimizing the overall TTM and eliminating deployment friction. Furthermore, stress-load resilience was quantitatively proven through Apache JMeter load testing: the modernized infrastructure successfully sustained 500 concurrent users (processing 5,000 requests) with a **0.00% error rate** and a stable throughput of 85 RPS. Such structural resilience prevents the cascading database locks and timeouts typical of legacy monolithic baselines.

Beyond traditional performance measures, this study introduces a new dimension of architectural evaluation: the effectiveness of LLM context. Quantitative analysis showed that the microservice-based Cariff platform achieves a **90%** reduction in net consumption compared to the monolithic baseline. By reducing the cost of AI development from **0,0476€** to **0,005 €** per request, the study shows that architectural decoupling is a necessary economic precondition for profitable engineering in the era of AI.

Ultimately, the roadmap for architectural transformation presented in this thesis establishes a high-fidelity, empirically backed reference for classifieds platforms seeking to maintain market leadership. By combining structural properties – such as domain isolation and modularity – with automated delivery and proven stress-load stability, organizations can build platforms that are highly performant today and strategically prepared for the unforeseen challenges of the future. The transition to microservices, when supported by quantitative validation, proves to be a vital strategic investment in the longevity of modern software products.

REFERENCES

Behutiye, W. N., Rodríguez, P., Oivo, M., and Tosun, A. (2017). Analyzing the concept of technical debt in the context of agile software development: A systematic literature review. *Information and Software Technology*, 82, 139-158.

<https://doi.org/10.1016/j.infsof.2016.10.004>

Fowler, M. (2009). 'Technical Debt Quadrant'. *Martin Fowler Blog*. Available at:

<https://martinfowler.com/bliki/TechnicalDebtQuadrant.html>

Detofeno, T., Malucelli, A. and Reinehr, S. (2023) 'Technical Debt Guild', *Journal of Software Engineering Research and Development*. Available at:

<https://doi.org/10.5753/jserd.2023.2417>

Kruchten, P., Nord, R.L. and Ozkaya, I. (2012) 'Technical Debt: From Metaphor to Theory and Practice', *IEEE Software*, 29(6), pp. 18–21. Available at:

<https://doi.org/10.1109/ms.2012.167>

Besker, T., Martini, A. and Bosch, J. (2019) 'Software developer productivity loss due to technical debt – A replication and extension study examining developers' development work', *Journal of Systems and Software*, 156, pp. 41–61. Available at:

<https://doi.org/10.1016/j.jss.2019.06.004>

Bridging Generation and Training: A Systematic Review of Quality Issues in LLMs for Code (2026) *Arxiv.org*. Available at: <https://arxiv.org/html/2605.05267> (Accessed: 11 May 2026).

Besker, T. et al. (2020) 'The influence of Technical Debt on software developer morale', *Journal of Systems and Software*, 167, p. 110586. Available at:

<https://doi.org/10.1016/j.jss.2020.110586>

Velepucha, V. and Flores, P. (2023) 'A survey on microservices architecture: Principles, patterns and migration challenges', *IEEE Access*, 11, pp. 1–1. Available at:

<https://doi.org/10.1109/ACCESS.2023.3305687>

Abgaz, Y., McCarren, A., et al. (2023). 'Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review'. *IEEE Access*, 11, 4213-4242. Available at: <https://doi.org/10.1109/ACCESS.2023.10160171>

Taibi, D., Lenarduzzi, V. and Pahl, C. (2017) 'Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation', *IEEE Cloud Computing*, 4(5), pp. 22–32. Available at: <https://doi.org/10.1109/mcc.2017.4250931>

Forsgren, N., Humble, J. and Kim, G. (2018) *Accelerate : the science behind DevOps : building and scaling high performing technology organizations*. It Revolution.

Joao et al. (2021) 'Monolithic to Microservices Migration Strategy in Public Safety Secretariat of Mato Grosso', *2019 International Conference on Electrical, Communication, and Computer Engineering (ICECCE)*, pp. 1–5. Available at: <https://doi.org/10.1109/icecce52056.2021.9514268>

Shahin, M., Ali Babar, M. and Zhu, L. (2017) 'Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices', *IEEE Access*, 5, pp. 3909–3943. Available at: <https://doi.org/10.1109/access.2017.2685629>

Stopford, B. (2018) 'Designing Event-Driven Systems Concepts and Patterns for Streaming Services with Apache Kafka.' First Edition. Edited by J. Billing. O'Reilly Media, pp. 1–166. Available at: <https://sitic.org/wordpress/wp-content/uploads/Designing-Event-Driven-Systems.pdf> (Accessed: 1 April 2026).

APPENDIX

GitHub source code of the microservices:

Api Gateway Service

<https://github.com/Molix228/cariff-api-gateway>

Auth Service

<https://github.com/Molix228/cariff-auth>

Scrapper Service

<https://github.com/Molix228/cariff-scrapper-service>

Notification Service

<https://github.com/Molix228/cariff-notification-service>

Listing Service

<https://github.com/Molix228/cariff-listings>

GitLab source code of the Frontend:

<https://gitlab.com/Molix228/cariff-frontend>

Website:

<https://cariff.com>