

A MODULAR MONOLITH ARCHITECTURE FOR HIGH-AVAILABILITY ELECTRIC VEHICLE CHARGING STATION MANAGEMENT SOFTWARE

Volodymyr Horshkov¹, Nataliia Lishchyna², Andrii Yashchuk², Olena Syrynovuch², Valerii Lishchyna²

¹Mint Innovations LTD, Lutsk, Ukraine, ²Lutsk National Technical University, Software Engineering Department, Lutsk, Ukraine

Abstract. The article presents the results of research and development of a highly available, fault-tolerant software system for managing electric vehicle charging stations. The proposed solution is based on a modular monolith architecture, which combines ease of implementation and maintenance with the ability to scale under increasing load. An analysis of the current state of charging infrastructure, architectural approaches, communication protocols (in particular, OCPP with WebSocket), and reliability-enhancing technologies such as containerization and message brokers is provided. The implemented functional modules of the system are described. The results of the experimental study confirm the system's stability and efficiency under high load conditions and demonstrate its readiness for deployment in a real-world environment.

Keywords: electric vehicles, charging stations, software architecture, reliability

MODUŁOWA ARCHITEKTURA MONOLITYCZNA DLA OPROGRAMOWANIA ZARZĄDZAJĄCEGO STACJAMI ŁADOWANIA POJAZDÓW ELEKTRYCZNYCH O WYSOKIEJ DOSTĘPNOŚCI

Streszczenie. W artykule przedstawiono wyniki badań oraz opracowanie wysokodostępного, odpornego na awarie oprogramowania do zarządzania stacjami ładowania pojazdów elektrycznych. Zaproponowane rozwiązanie opiera się na architekturze modułowego monolitu, która łączy prostotę wdrożenia z możliwością skalowania systemu w warunkach rosnącego obciążenia. Przeprowadzono analizę aktualnego stanu infrastruktury stacji ładowania, podejść architektonicznych, protokołów komunikacyjnych (w szczególności OCPP z wykorzystaniem WebSocket) oraz środków zwiększających niezawodność, takich jak konteneryzacja i brokery komunikatów. Opisano zaimplementowane moduły funkcjonalne systemu. Wyniki badań eksperymentalnych potwierdzają stabilność i efektywność systemu w warunkach wysokiego obciążenia oraz wykazują jego gotowość do wdrożenia w rzeczywistym środowisku.

Słowa kluczowe: pojazdy elektryczne, stacje ładowania, architektura oprogramowania, niezawodność

Introduction

The growth in the number of electric vehicles requires the expansion of charging networks and the development of reliable software systems for their management. Stability, high availability, and fault tolerance of the infrastructure are key conditions for the advancement of electromobility, which imposes increased demands on the scalability and performance of solutions [11].

Most existing systems are based on either monolithic or microservice architectures. Monolithic solutions are simpler to develop but scale poorly [15], while microservices provide flexibility yet require significant resources to maintain inter-service communication [14]. An alternative is the modular monolith architecture, which combines the simplicity of a monolith with the separation of responsibilities and flexibility typical of microservices. This approach enables scalability without substantial system redesign, making it an optimal solution for managing charging stations under high-load conditions [6].

The aim of this study is the experimental evaluation of a developed highly available, fault-tolerant software system for managing charging stations based on a modular monolith architecture. The objectives include:

- Analyzing existing approaches and technologies;
- Implementing a system that ensures key parameters of high availability, scalability, and reliability;
- Evaluating the effectiveness of the proposed solution under real-world load conditions.

The approach involves the use of modern communication protocols (OCPP with WebSocket) [7], asynchronous interaction between components via message brokers (e.g., Apache Kafka or RabbitMQ), and containerization to enhance fault tolerance [22].

The results of the study have practical significance for developers and operators of charging infrastructure, contributing to the further adoption and improvement of electric vehicle technologies.

1. Current state of the problem

Recent research on electric vehicle charging station management focuses on architectures that ensure high availability, scalability, and fault tolerance. The main approaches are monolithic, microservice, and modular monolith architectures.

Monolithic systems are simple and suitable for early development but have scalability limitations, as higher loads demand significant hardware resources [15].

Microservice architecture enables flexible scaling, parallel development, and easier maintenance but complicates communication and data consistency [14].

The modular monolith, actively studied in recent years, combines the advantages of both previous approaches. It is a single deployable application internally divided into independent modules, each encapsulating its own business logic. This ensures separation of concerns, better maintainability, and flexibility while simplifying development and deployment compared to microservices, without sacrificing scalability or reliability [1, 6].

Among modern platforms, OpenEMS stands out as an open-source modular system for energy microgrids, integrating charging stations via protocols such as OCPP, Modbus, and MQTT. Implemented in Java, it operates on both edge devices and in the cloud [8, 16].

Other notable solutions include ChargeLab, an open platform (cloud and on-premises) supporting OCPP, REST API, WebSocket, analytics, and billing [2], and eMobility Cloud, a microservice-based system integrating OCPP and OPC UA with APIs for payment service interaction [5].

Virta is one of Europe's leading EV charging platforms, supporting OCPP and OCPI while providing billing, analytics, and mobile applications [21]. Greenlots (now Shell Recharge) specializes in cloud-based fast-charging solutions with a focus on analytics, cybersecurity, energy management, and ISO 15118 compliance [20]. EVBox targets large charging networks, supports OCPP and OCPI, integrates with ERP systems and mobile apps, and is built on a microservice, cloud-based architecture [4].

Modern charging platforms rely on key technologies and standards such as OCPP for station communication, OCPI for roaming, and ISO 15118 for automatic identification and charging [12, 17]. Open APIs, particularly REST and WebSocket, enable real-time interaction and integration with payment and external systems [13, 19].

Scalability is achieved through cloud infrastructure, containerization (Docker, Kubernetes), and microservice architectures. Cybersecurity remains critical, encompassing authentication, authorization, encryption, key management, and activity auditing. Analytics and big data processing are essential for optimizing energy consumption and forecasting load demand. Systems must also support flexible integration with diverse hardware and seamless scaling as the number of users and devices grows [9, 10].

In academic literature, significant attention is given to communication protocols and fault-tolerance mechanisms. The OCPP standard combined with WebSocket remains the most widely adopted solution for stable, bidirectional communication [18]. Asynchronous interaction using message brokers such as RabbitMQ or Apache Kafka, along with containerization and orchestration technologies, further enhances the resilience of distributed systems [3, 22].

Overall, these findings underscore the importance of integrating modern architectural solutions and technologies to develop highly available and fault-tolerant EV charging infrastructure management systems.

2. Problem statement

The main objective of this study is the development of a highly available and fault-tolerant software system for managing electric vehicle charging stations that meets the requirements of modern e-mobility infrastructure.

The object of the study is a software system for managing the infrastructure of electric vehicle charging stations. This system includes the Central System Management System (CSMS), charging stations (Charge Points), and the mechanisms of their interaction.

The subject of the study is the design and analysis of a modular monolith architecture, its implementation, and experimental evaluation in the context of developing highly available and fault-tolerant software for charging station management.

2.1. System requirements

To ensure effective management of charging infrastructure and meet end-user needs, the system must fulfill both functional and non-functional requirements (Figure 1).

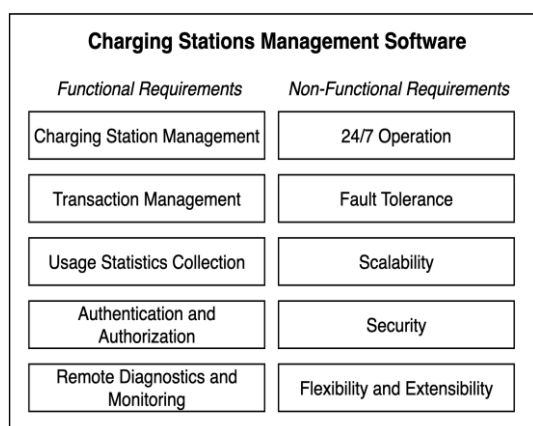


Fig. 1. Requirements for the developed charging station management system

The charging station management system must provide station registration, status monitoring, charging session control (start/stop), software updates, as well as tools for authentication, authorization, remote diagnostics, and monitoring. A critical requirement is the support of internal transactions between components to ensure reliability, fault tolerance, and business process consistency, along with the collection of statistical data for analytics.

The non-functional requirements include uninterrupted operation 24/7, resilience to individual component failures, and horizontal scalability to support a growing number of users and stations. The system must also guarantee data protection through encrypted communication and remain adaptable for integrating new equipment and extending functionality in compliance with industry standards.

2.2. Performance evaluation criteria

The effectiveness of the developed system will be evaluated according to the following criteria:

- Performance: system response time to key requests under varying load levels.
- Fault tolerance: the system's ability to recover after simulated failures in key components and the duration of downtime.
- Scalability: system behavior and performance when increasing the number of parallel sessions, including analysis of performance changes, resource utilization, and service response stability.
- Resource efficiency: assessment of computational resource consumption (CPU, RAM) depending on the load.
- Availability: maintaining system operability during degradation, analyzed on the basis of continuous request execution during failure simulation, with logging of service recovery time and API responses.

3. Architectural approach and system design

3.1. Justification for the choice of architecture

The choice of architecture is a critical factor in developing a reliable and scalable charging station management system. While monolithic systems are simple to implement, they quickly lose flexibility and efficiency as the system scales [15]. Microservice architecture, on the other hand, offers high scalability and performance but increases system complexity and demand greater management resources [14].

Given these trade-offs, the modular monolith was selected as the optimal approach. It combines the strengths of both architectures by maintaining a single deployable application divided into independent modules with clear responsibilities [6].

This structure simplifies development and deployment while enabling flexible modification and gradual scaling of components. Moreover, it provides a stable foundation for future evolution into a full microservice architecture if system growth requires it [1].

As a result, the modular monolith ensures high availability and fault tolerance without unnecessary architectural complexity in the early development stages.

In contrast to fully distributed microservice-based EV charging platforms, which introduce additional operational overhead due to inter-service communication, latency, and complex DevOps management, the modular monolith reduces architectural complexity while maintaining clear separation of concerns. For medium-scale and rapidly growing charging networks, this trade-off ensures operational reliability and simplified deployment without the immediate need for distributed transaction management or multi-service orchestration. Therefore, the proposed architecture supports controlled scalability with lower infrastructure complexity.

3.2. Structural model of the system

The proposed modular monolith architecture for the charging station management system is based on the principle of separation of concerns, where each component is responsible for a specific business function. This ensures high cohesion within modules and minimal coupling between them. The structural model of the system includes the following key modules:

- Authorization Module ensures secure access for users and devices by implementing authentication through identifiers or RFID tags.
- User Management Module is responsible for processing and storing user data as well as managing profiles.
- Configuration Module stores the parameters of charging stations and ports, enabling flexible configuration.
- Station Management Module processes information about stations, ports, and their associations with users.
- Transaction Module manages charging processes, including port reservations, deferred operations, and standard sessions.
- Diagnostics Module collects data on port status and key metrics during charging.
- Statistics Module gathers and analyzes information on energy consumption, costs, and tariff plans (Figure 2).

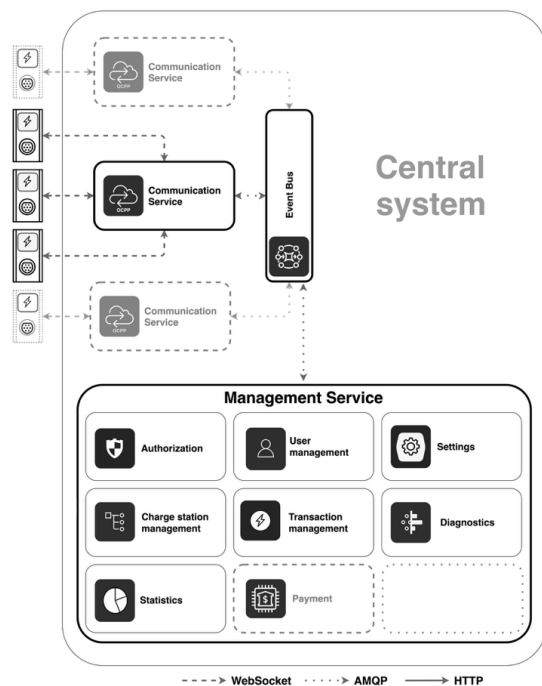


Fig. 2. Modules of the charging station management system under development

Such a modular structure ensures logical separation of the system into independent components, making it easier to develop, maintain, and scale further.

3.3. Modeling constrained data contexts

Modeling bounded contexts is a key principle in modular monolith architecture that helps to avoid the "Big Ball of Mud" and maintain clear separation of concerns. In the context of a charging station management system, each module is treated as a distinct bounded context with its own data model and business rules relevant only within that module.

Each module owns its data and is responsible for maintaining its consistency. For example, in a charging station management system, the Station Management module operates with the ChargePoint and Connector entities, while the Authorization module deals with User and Token data.

This approach guarantees module independence, allowing them to evolve without affecting other parts of the system, since

communication occurs through stable, well-defined interfaces. It simplifies maintenance, scaling, and the overall evolution of the system.

3.4. Communication between components

A key element in ensuring fault tolerance and high availability is communication with charging stations, which requires processing large volumes of messages in real-time. To address this, a dedicated Communication Service was introduced, functioning as a WebSocket server and supporting interaction via the OCPP protocol [18]. This approach increases system availability and simplifies the scaling of individual modules.

Interaction between components is coordinated using the Advanced Message Queuing Protocol (AMQP) implemented through a message broker. This enables asynchronous data exchange, reduces the load on individual subsystems, and improves the overall efficiency of the system.

3.5. Ensuring fault tolerance and scalability

Fault tolerance is a critical non-functional requirement, implemented through containerization (Docker, Kubernetes). This ensures automatic scaling, module recovery without system downtime, and simplified updates. The modular structure with orchestration support also provides a foundation for potential migration to a microservice architecture.

Asynchronous interaction via message brokers enhances resilience, prevents cascading failures, and guarantees operation execution even during temporary unavailability of individual components.

Data security is ensured through centralized secret management in Kubernetes, which protects passwords, tokens, and access keys at the infrastructure level and allows their updates without interrupting service operation.

Security mechanisms also contribute to system availability. Token-based authentication and certificate management reduce the risk of unauthorized access and resource exhaustion, while centralized secret management in Kubernetes enables credential rotation without service interruption. Thus, security controls function not only as protective measures but also as components supporting operational reliability.

3.6. Technological implementation and infrastructure

A modern technology stack was applied for the development of the software system, focusing on flexibility, scalability, and high availability. The server side was implemented in C# using ASP.NET, which enables the creation of secure REST APIs with asynchronous processing. For storing structured data (transactions, users, charging station status), Microsoft SQL Server was chosen as a reliable relational DBMS.

Communication with charging stations is implemented through OCPP 1.6 over WebSocket, providing real-time bidirectional data exchange. Asynchronous interaction between modules and fault tolerance are supported by the RabbitMQ message broker, which allows components to operate independently.

The system is deployed in a containerized environment (Docker, Kubernetes) (Figure 3), enabling automatic scaling of key components according to the load.

The use of Kubernetes ensures automatic module restarts in case of failures, load balancing, and updates without system downtime. Data security is guaranteed through centralized secret management, which protects keys and access tokens at the infrastructure level. User and service authorization is implemented through tokens with automatic renewal, ensuring continuous and secure data exchange. Thanks to the modular architecture, the system remains flexible and can be gradually migrated to a microservice architecture if required.

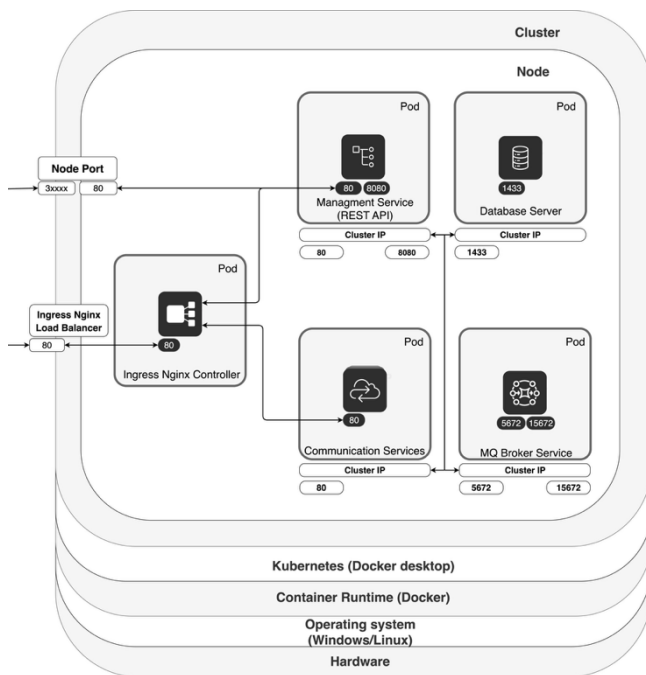


Fig. 3. Schematic representation of server infrastructure

3.7. Client-side implementation

The client application, developed with .NET MAUI (.NET 8), is a cross-platform solution for both mobile and desktop devices. Its architecture follows the MVVM pattern using the Prism framework, which ensures clear separation of concerns.

For server communication, the application employs Refit (declarative REST requests) and SignalR (real-time data exchange). Additional features include local caching (MonkeyCache), charting (SkiaSharp, LiveChartsCore), and logging (Serilog).

Authentication and authorization are implemented via HttpClientFactory with token handlers, simplifying access management. The application includes pages for authorization, registration, station overview, statistics, and settings. The interaction scheme of the application's pages and services is shown in Figure 4.

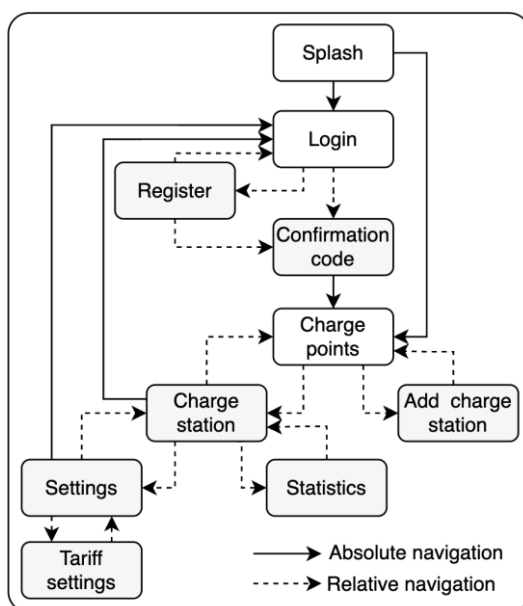


Fig. 4. Schematic representation of application pages and navigation

4. Experimental study

4.1. Experimental testing methodology and criteria

To assess the reliability and performance of the charging station management software system, a methodology was developed that takes into account its operational characteristics and allows testing the system under realistic load conditions. It includes a description of the test environment, tools for load simulation, methods for measuring key performance indicators (response time, throughput), analysis of behavior during failures, and verification of service availability in the event of partial disruptions. The tests involved all critical components: the Management Service, Communication Service, database, mobile client, and charging station simulator. The main focus was placed on load generation, system monitoring, and subsequent analysis of logs and metrics.

Preparation for experimental testing began with the deployment of the server infrastructure in a local isolated environment based on Docker Desktop with Kubernetes enabled, ensuring independent and scalable execution of each component. Docker images were built for all key services (including the Management Service and Communication Service) using configured Dockerfiles, while recommended public images were used for SQL Server and RabbitMQ. The resulting images were uploaded to registries for further use in the cluster.

Subsequently, Kubernetes objects were created via kubectl: Deployments for managing the number of replicas (supporting horizontal scaling) and Services for routing and load balancing. The architecture provided internal access to the database via ClusterIP, while the management and communication services were exposed externally through NodePort, offering flexibility for testing and simulating client requests. This configuration ensured a controlled and reproducible environment for further load testing, failure analysis, and system stability evaluation.

To ensure resilience and high availability, the services were configured for automatic restarts, and Kubernetes employed Liveness and Readiness Probes to detect and replace faulty instances. Special attention was paid to data persistence during restarts, particularly for SQL Server: a PersistentVolume was configured on the host along with a corresponding PersistentVolumeClaim in the Deployment configuration, ensuring state preservation and enabling future data migration between nodes or to backup storage.

Infrastructure monitoring was performed using Docker Desktop tools, allowing evaluation of CPU, memory, and disk usage for performance analysis under load. To emulate public deployment, the NGINX IP address was locally bound to the domain chargingexample.com via the hosts file modification; NGINX acted as a reverse proxy, routing requests to the appropriate services in the Kubernetes cluster, thus making test requests as close to real client scenarios as possible.

The test environment was complemented by emulated clients: a mobile application running in iOS and Android emulators based on .NET MAUI, with various device characteristics configured (resolution, network conditions, etc.), and a charging station simulator. The latter used the domain ws://chargingexample.com as the WebSocket server address and transmitted basic OCPP information to establish a connection, enabling the reproduction of client-server interaction scenarios and validation of message exchange correctness.

Component interaction followed predefined architectural principles and was verified through key scenarios. Initially, a user registered via the mobile app, with authentication handled by the Management Service through its REST API. After registration, the user received a unique access token for subsequent requests. This process (Figure 5) confirms correct authorization, user management, and integration between the service and client side.

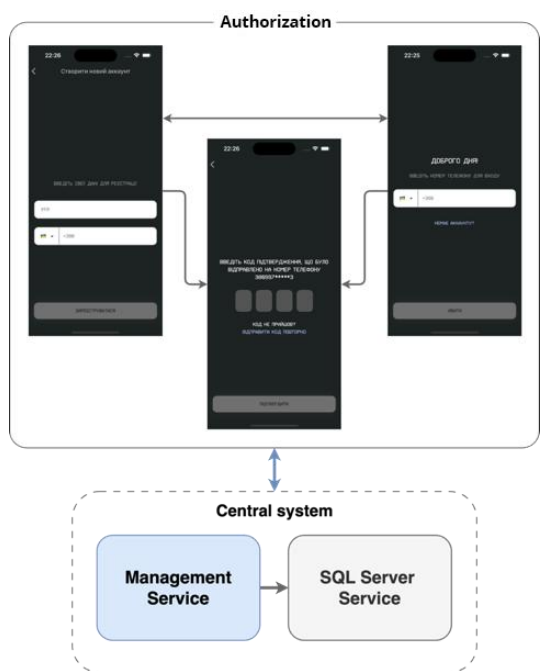


Fig. 5. Schematic representation of the interaction between the application and the central charging station management system

Next, a test charging station was added to the system via the Swagger interface of the Management Service: at the endpoint `http://chargingexample.com/api/ChargePoint/insert_or_update`, data for the station EVB-P10001797 with a single port was submitted. The Management Service verified and stored this information in the SQL Server database and ensured its synchronization with other system components.

After that, the charging station simulator was connected, which established a WebSocket connection with the Communication Service, transmitting the identifier EVB-P10001797. The Communication Service checked the station's existence by interacting with the Management Service, and after confirming its registration, initiated communication.

The connection process and the message exchange between the charging station and the central system are shown in Figure 6, followed by a basic validation of functionality in accordance with the OCPP specification.

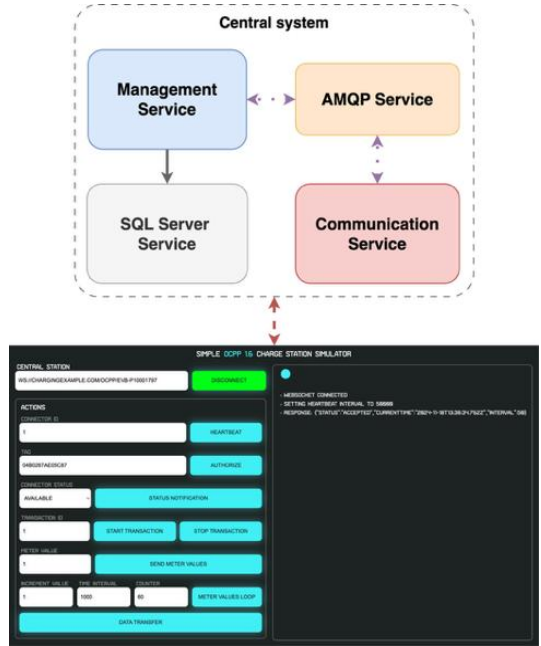


Fig. 6. Interaction of the charging station with the central control system

The next step in comprehensive testing involved linking the charging station to a test user via the client interface. The user entered the station identifier in a form, which triggered a request to the Management Service. The service searched for the station, established a link between it and the user in the database, and, after successful association, made the station available in the user interface.

This made it possible to verify the functionality of managing a charging station directly from the mobile application. The process is illustrated in Figure 7.

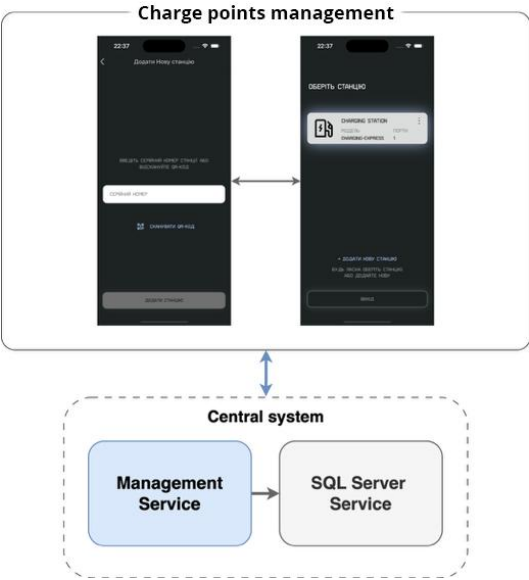


Fig. 7. Charging station management

For the performance evaluation, a set of tools was prepared, with the Postman Collection Runner serving as the core component, enabling the automation of HTTP requests that simulate real-world system usage scenarios. A Postman environment was configured with request parameters – URLs, authorization tokens, and specific data – as well as load metrics such as the number of parallel virtual sessions and test duration. Each request included checks for HTTP status compliance and validation of key fields in response bodies to monitor response time and correctness of system behavior.

Initially, a collection of requests was created for isolated backend testing without involving the Communication Service or charging stations. It consisted of a sequence of nine GET requests targeting different modules of the Management Service, allowing verification of database interactions and assessment of load distribution under conditions without external influences (Figure 8). This approach made it possible to isolate internal service performance from the behavior of integrated components.

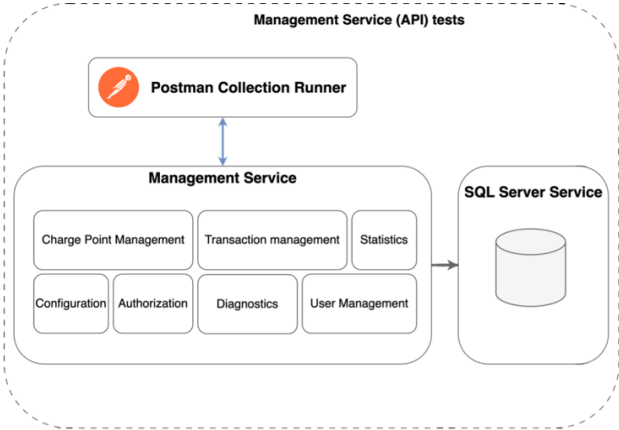


Fig. 8. Schematic representation of REST API load scenario

The second collection of requests simulated interaction with charging stations through the Communication Service, specifically the transmission of commands from the Management Service to the station using the OCPP TriggerMessage. Through GET requests to the Management Service, different variations of the parameter triggerMessageType (for example, status update or diagnostic data request) were initiated, transmitted to the Communication Service, and subsequently delivered to the charging station; in response, the station generated the expected messages.

Each parallel virtual session executed a sequence of four such requests with different message types, allowing the simulation and loading of the communication channel through the server. The interaction scheme is shown in Figure 9.

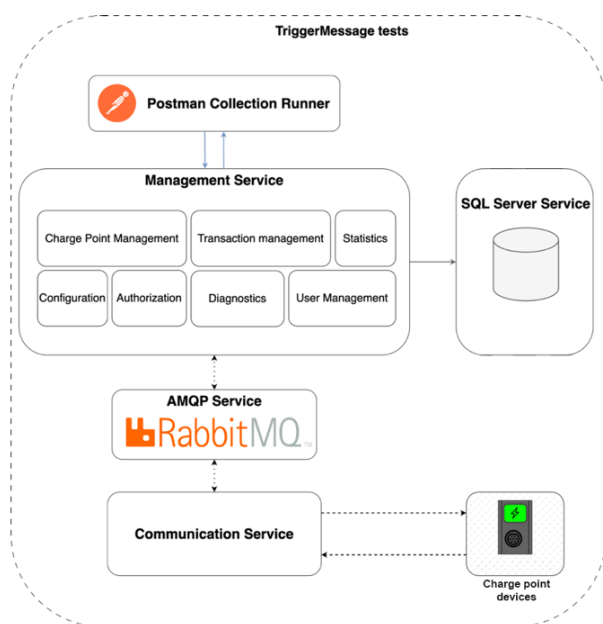


Fig. 9. Schematic representation of the load scenario of the central charging station management system

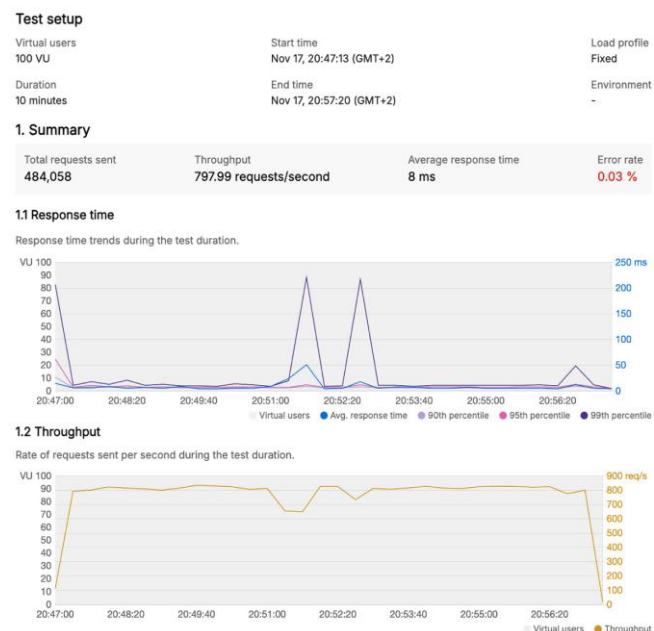


Fig. 10. Example of a load testing report in postman

During the experiments, Docker Desktop tools were used to monitor the state of the server infrastructure in real time. These tools displayed the status of containers, their CPU, memory, and disk usage, as well as logs, providing the necessary visibility for diagnostics under load and during anomalies.

Fault tolerance assessment was carried out by simulating infrastructure-level failures. Forced restarts of deployments were performed using the command `kubectl rollout restart deployment`, while critical failure simulation was achieved by deleting pods (`kubectl delete pod`). Recovery was tracked via Kubernetes monitoring tools (including `kubectl get pods`) and Docker Desktop / Kubernetes Dashboard, recording the time to create a new instance, the availability of APIs during and after recovery, and the correctness of request handling.

Particular attention was paid to data preservation: the SQL Server container had a bound storage (Persistent Volume / Persistent Volume Claim), ensuring database integrity and availability after failures.

The high availability evaluation methodology included measuring the interval from failure occurrence to service readiness, analyzing the number of successful requests during recovery, and assessing system throughput during degradation. For this purpose, continuous requests were executed through Postman, recording response statuses and times (Figure 10).

The obtained metrics made it possible to quantitatively characterize the system's resilience, performance, and readiness to serve clients under unpredictable failure conditions.

4.2. Experimental study results

The evaluation presented in this study examines the performance characteristics of the proposed modular monolith architecture under realistic workload conditions. The experimental design was focused on assessing the feasibility, stability, and scalability of the developed solution within a production-oriented environment. Comparative benchmarking against alternative architectural implementations was considered beyond the scope of this work.

In the process of analyzing the experimental study results, attention was focused on three key aspects: performance under load, fault tolerance, and recovery time after failures.

At the initial stage, the basic functionality of the system was verified through the interaction between the client application and the charging station simulator, including station control, real-time port status updates via SignalR, modification of configuration parameters, and correct generation and display of statistical data. This ensured the verification of core operational scenarios and confirmed the functional integrity of the system prior to subsequent load and stress testing.

The analysis of availability testing results for the central charging station management system demonstrated high efficiency under moderate load and an approach toward the throughput limit as the load increased (Table 1).

Table 1. Result of REST API load tests

Parallel sessions	Average number of requests/sec	Average API response time	Capacity	Error rate	Processor usage
20	162.99	6 ms	stable	not detected	up to 70%
50	415.85	4 ms	stable	not detected	~ 83%
100	797.99	8 ms	decrease	0.03%	~180%

The system operates efficiently under a load of up to 50 concurrent sessions, maintaining a low response time (4–6 ms) and stable throughput without errors. At 100 parallel sessions, an increase in response time, a decrease in throughput, and the occurrence of a small error rate (0.03%) were observed, indicating proximity to the performance limit.

Resource utilization analysis revealed that the CPU is the system's bottleneck. At 50 sessions, CPU usage reached approximately 83%, and at 100 sessions – around 180% (across cores). This indicates the need for optimization or horizontal scaling to handle higher loads.

From a practical perspective, further scalability improvements may be achieved through horizontal pod autoscaling in Kubernetes based on CPU and request metrics, optimization of asynchronous processing pipelines, request batching strategies, and fine-tuning of thread pool configurations. Additionally, selective scaling of the Communication Service independently from the Management Service may reduce contention under high message throughput. These measures would allow extending the operational limits without fundamental architectural redesign.

Meanwhile, memory, disk, and internal network usage remained stable. This confirms that at such load levels, the CPU is the primary limiting factor, and further scaling should focus on horizontal expansion or request processing optimization.

During comprehensive testing of the central management system together with the charging station simulator, a slight decrease in throughput was observed compared to the previous scenario. However, overall performance remained acceptable, ensuring stable system operation (Table 2).

Table 2. Results of comprehensive testing with a simulator

Parallel sessions	Average number of requests/sec	Average API response time	Error rate	Processor usage (management service)	Processor usage (communication service)
20	77.45	~6 ms	not detected	25–40%	12–20%
50	194.54	~5 ms	not detected	42–50%	25–28%
100	390.36	~4 ms	not detected	60–70%	35–36%

Under this scenario, the system demonstrated high stability and performance even as the load increased. Across all tested scenarios (from 20 to 100 sessions), the system maintained a consistently low response time (4–6 ms) and a zero error rate, indicating efficient scalability and the absence of bottlenecks in the operation of key services.

Increasing the number of parallel sessions to 100 led to a moderate increase in CPU load for both the Management Service and Communication Service, yet both services successfully handled the workload. The results confirm that the system's architecture is reliable and fault-tolerant, capable of processing large volumes of requests without data loss or functional degradation.

The fault tolerance analysis included forced container restarts, service outages, and infrastructure overloads, followed by observation of automatic recovery. Sudden restarts of the Management Service, Communication Service, and SQL Server containers resulted in their restoration within 5–10 seconds on average, thanks to Kubernetes mechanisms, while data remained accessible through external storage.

Disabling individual services showed partial availability of other components, and after automatic relaunch, functionality was restored within a few seconds. Full recovery, including reconnection to dependent resources, did not exceed 15 seconds, and the system remained accessible to users even during the recovery process, meeting high availability criteria.

Table 3. Fault tolerance and recovery time

Failure type	Recovery time
Forced container restart	5–10 seconds
Complete system recovery after failure	No more than 15 seconds

The system demonstrates high fault tolerance thanks to Kubernetes mechanisms, ensuring user accessibility even during the recovery process.

Although the experiments were conducted in a controlled local Kubernetes environment, the architectural decisions remain portable to public cloud infrastructures such as AWS, Microsoft Azure, or Google Cloud Platform.

Kubernetes provides an abstraction layer that enables seamless migration from local clusters to managed cloud services, where NodePort exposure can be replaced with LoadBalancer or Ingress controllers, and PersistentVolumes can be mapped to cloud storage solutions.

Therefore, the obtained performance and resilience characteristics are expected to generalize to production-grade distributed deployments, subject to appropriate infrastructure provisioning.

5. Conclusions

This article has examined an approach to building highly available software for managing electric vehicle charging stations based on a modular monolith architecture. The proposed solution combines the advantages of the classical monolith – simplified deployment and management – with modularity, which ensures development flexibility, scalability, and functional isolation of system components.

The implementation of the system using this architectural approach has made it possible to achieve:

- high performance, due to efficient interaction of modules within a single process;
- increased fault tolerance, through monitoring mechanisms, self-recovery, and control of charging station status;
- simplified maintenance and testing, enabled by clear functional separation across modules;
- the potential for migration to a microservice architecture, if higher loads or scalability requirements arise.

Thus, the modular monolith architecture is a well-founded choice for developing mission-critical software that must ensure continuous availability and control over charging infrastructure.

Future research will address the integration of additional automatic scaling mechanisms and interoperability with power grids and intelligent load distribution systems. Particular attention will be devoted to conducting a structured comparative analysis between modular monolith and microservice-based implementations in terms of performance, development complexity, and operational costs. Further investigation of advanced scaling strategies, including Kubernetes Horizontal Pod Autoscaling and cloud-level autoscaling groups, is also planned. Additionally, integration with smart grid interaction standards such as OpenADR and demand-response mechanisms represents a promising direction for extending system interoperability with power distribution infrastructures and intelligent load management platforms.

References

- [1] Belge, E. (2025). *Clean architecture with modular monolith and vertical slice*. Medium. <https://medium.com/@eda.belge/clean-architecture-with-modular-monolith-and-vertical-slice-896b7ee22e3e>
- [2] ChargeLab Inc. (2025). *ChargeLab*. <https://chargelab.co/>
- [3] DAC.digital. *Software development for electric vehicle charging stations platform*. <https://dac.digital/case-studies/software-development-for-electric-vehicle-charging-stations-platform/>
- [4] EVBox. (2025). *Smart EV charging stations and software*. <https://evbox.com/en/>
- [5] Fink IT-Solutions GmbH & Co KG. (2025). *FITS/eMobility Cloud®*. <https://emobilitycloud.de>
- [6] Horshkov, V., Lishchina, N., & Sychuk, V. (2024). Modular monolith: An architectural approach for building a high-availability electric vehicle charging station management system. *Computer-Integrated Technologies: Education, Science, Production*, 57, 31–42. <https://doi.org/10.36910/6775-2524-0560-2024-57-05>
- [7] Ionity. *Shho take ocpp vidkrytyj protokol zaryadnoyi stancziyi i navishho vin potriben*. Retrieved August 15, 2025, from <https://ionity.ua/faq/shho-take-ocpp-vidkrytyj-protokol-zaryadnoyi-stancziyi-i-navishho-vin-potriben/>
- [8] OpenEMS. *Documentation*. Retrieved August 15, 2025, from <https://openems.github.io/openems.io/openems/latest/contribute/documentation.html>
- [9] ScienceDirect. *Energy Reports* (Vol. 9, Supplement C). Retrieved August 15, 2025, from <https://www.sciencedirect.com/journal/energy-reports/vol/9/suppl/C>
- [10] International Energy Agency. (2023). *Global EV outlook 2023*. <https://www.iea.org/reports/global-ev-outlook-2023>
- [11] International Energy Agency. (2023). *Global EV outlook 2023: Catching up with climate ambitions*. OECD Publishing. <https://doi.org/10.1787/cbe724e8-en>

- [12] International Organization for Standardization. (2025). *Road vehicles — Vehicle to grid communication interface — Part 20: 2nd generation network layer and application layer requirements — Amendment 1: AC DER service, MCS service, and improved security concept* (ISO/DAmD Standard No. 15118-20:2022/DAmD 1).
- [13] Khalid, M. R., Alam, M. S., Sarwar, A., & Asghar, M. S. J. (2019). A comprehensive review on electric vehicles charging infrastructures and their impacts on power-quality of the utility grid. *eTransportation*, 1, Article 100006. <https://doi.org/10.1016/j.etrans.2019.100006>
- [14] Lewis, J., & Fowler, M. (2014). *Microservices: A definition of this new architectural term*. MartinFowler.com. <https://martinfowler.com/articles/microservices.html>
- [15] Newman, S. (2019). *Monolith to microservices: Evolutionary patterns to transform your monolith* (1st ed.). O'Reilly Media.
- [16] Okoyomon, E., & Goebel, C. (2024). EnergyOS: Designing a multi-application EMS platform. In *Proceedings of the 15th ACM International Conference on Future and Sustainable Energy Systems (e-Energy'24)*, 237–255. Association for Computing Machinery. <https://doi.org/10.1145/3632775.3661961>
- [17] Open Charge Alliance. *Open charge point protocol*. <https://openchargealliance.org/protocols/open-charge-point-protocol/>
- [18] Open Charge Alliance. (2024). *What is OCPP and how to use it for smart charging: Troubleshooting and common issues during commissioning*. https://cdn.prod.website-files.com/617180d35a4f4fd9084e5396/6765e35eea01b5898949bb78_OCPP%20Report%202024.pdf
- [19] Elsevier. *Renewable and Sustainable Energy Reviews*. ScienceDirect. <https://www.sciencedirect.com/journal/renewable-and-sustainable-energy-reviews>
- [20] Shell EV Charging Solutions B.V. *Shell Recharge*. Shell. <https://shellrecharge.com/>
- [21] Virta. (2025). *Virta — EV charging software*. Virta Global. <https://www.virta.global/>
- [22] Waehner, K. (2025). *Virta's electric vehicle (EV) charging platform with real-time data streaming: Scalability for large charging businesses*. Kai Waehner Blog. <https://www.kai-waehner.de/blog/2025/04/22/virtas-electric-vehicle-ev-charging-platform-with-real-time-data-streaming-scalability-for-large-charging-businesses/>

M.Sc. Volodymyr Horshkov

e-mail: vladslid@gmail.com

Team Lead (Software Engineering) at Mint Innovations, Lutsk, Ukraine. Master's degree in software engineering, Lutsk National Technical University. Has over 5 years of experience in software development. Works on projects related to e-commerce solutions.

Research interests: software engineering, software architecture, cross-platform applications.

<https://orcid.org/0009-0008-3907-7212>**Ph.D. Nataliia Lishchyna**

e-mail: lishchyna@gmail.com

Associate professor, Head of the Department of Software Engineering, Lutsk National Technical University. Author of more than 70 scientific publications.

Research interests: software development, data mining, innovative technologies in education.

<https://orcid.org/0000-0002-5200-536X>**Ph.D. Andrii Yashchuk**

e-mail: xxxxyandyxxx@gmail.com

Associate professor of Software Engineering Department in Lutsk National Technical University. Scientific and practical interests are related to software engineering, automated information systems, internet of things, computer vision. Author of more than 40 scientific publications.

<https://orcid.org/0000-0003-4872-7949>**Ph.D. Olena Surynovych**

e-mail: sivom@ukr.net

Associate professor of Software Engineering Department in Lutsk National Technical University, specializes in software engineering and project management, author of 76 publications, participant in Erasmus+ and Horizon Europe projects, experienced in international training, and active in innovative entrepreneurship education.

<https://orcid.org/0000-0002-9300-0039>**Ph.D. Valerii Lishchyna**

e-mail: lvaleriy@gmail.com

Associate professor, Head of the Department of Computer Science, Lutsk National Technical University.

Author of more than 70 scientific publications. Scientific and practical interests are related to such areas as smart cities and modern mobility, software engineering.

<https://orcid.org/0000-0002-2371-3850>