Journal homepage: https://itce.vn.ua/en

Information Technologies and Computer Engineering **Vol. 23, No. 2. 2026**

Article's History: Received: 27.01.2026 Revised: 22.05.2026
Accepted: 25.06.2026 Published: 03.08.2026

ISSN 1999-9941; e-ISSN 2078-6387
UDC 004.056
DOI: 10.31649/vitce/2.2026.35

Methodology for automating enterprise business processes based on API integration

Dmytro Zahorulko*
Postgraduate Student
State University "Kyiv Aviation Institute"
103058, L. Huzar Ave., 1, Kyiv, Ukraine
https://orcid.org/0009-0000-9402-3390

Abstract. In the context of rapid digitalisation and a high level of heterogeneity of contemporary corporate ecosystems, the problem of fragmentation of the information space, and the emergence of isolated data warehouses becomes a critical barrier to business efficiency. The purpose of the study was to develop and substantiate a comprehensive methodology for automating business processes of an enterprise based on the application programming interface – integration of the application programming interface, combining the choice of optimal architectural patterns with mechanisms for secure data exchange. To achieve this goal, the researchers applied system analysis of architectural styles, prototyping of software gateways in Python, and computer modelling of load scenarios in an isolated virtual environment using Docker containers and the Locust tool. The study involved a comparative analysis of the Representational State Transfer, Simple Object Access Protocol, and Graph Query Language protocols in terms of performance and network load, which revealed significant traffic overhead and delays of up to 185 ms when using outdated standards. A hybrid integration methodology was proposed that combines the Representation State Transfer architecture for mobile clients and Graph Query Language for web interfaces, which reduced the amount of transmitted data by 73% and reduced the system response time to 65 ms. A secure data exchange algorithm has been developed based on the use of an intermediate layer (middleware) and a security gateway (API Gateway), which implement two-level request validation and centralised access control to neutralise the risks of unauthorised interference. The expediency of using the "Strangler Fig" migration pattern was substantiated, which allowed for a gradual transition from monolithic enterprise resource planning systems to microservice architecture through a specialised Python gateway without stopping operational activities. It has been experimentally proven that the introduction of asynchronous message queues allows the system to maintain stable operation within 320 ms even at peak loads of up to 5,000 requests per second, in contrast to monolithic solutions that demonstrate exponential performance degradation

Keywords: heterogeneous IT environments; REST; GraphQL; security gateways; Strangler Fig migration pattern; middleware; network latency minimisation

Introduction

Contemporary corporate information ecosystems are characterised by a high level of heterogeneity, combining legacy monolithic platforms (Legacy ERP – Enterprise Resource Planning) with cloud microservices and external SaaS solutions (Software as a Service). In such conditions, the critical barrier to operational efficiency is not the lack of automation as such, but the fragmentation of the information space, which leads to the emergence of isolated data warehouses ("data silos"). As noted by N. Ford *et al*. (2021), in complex digital infrastructures, it is the API gateways (Application Programming Interface) that become the fundamental link that allows aggregating data from heterogeneous sources, while providing the necessary level of abstraction for business logic. Practice shows that simply increasing the number of digital tools without implementing a unified integration strategy leads to an exponential increase in the complexity of infrastructure support and delays in decision-making.

Suggested Citation:
Zahorulko, D. (2026). Methodology for automating enterprise business processes based on API integration. *Information Technologies and Computer Engineering*, 23(2), 35-43. doi: 10.31649/vitce/2.2026.35.

*Corresponding author (*student675@ukr.net*)

The problem of real-time synchronisation of transactional data is particularly acute, where conventional batch processing no longer meets the business requirements for performance. According to M. Niswar *et al*. (2024), to overcome network delays in heterogeneous systems, it is crucial to switch to advanced data exchange protocols that allow optimising the payload and ensuring information integrity at the API level. This necessitates the development of new architectural approaches that would ensure seamless interaction of heterogeneous components, while guaranteeing data integrity and protection against unauthorised access at the level of software interfaces.

In practice, most business entities use a heterogeneous set of software products, each of which is responsible for a separate area of work: CRM Systems (Customer Relationship Management) for communication with customers, specialised software for warehouse accounting, separate accounting programmes, and payment gateways. This disparity leads to "information gaps", when data from one system does not automatically get to another. This leads to the need to duplicate information manually, which increases the risk of errors, increases delays, and makes it impossible to get real-time analytics. The solution to this problem is to create a single information environment through API integration. As noted by M. Szewczyk & M. Skublewska-Paszkowska (2025), even within the same architectural style – Representational State Transfer (REST) – different server frameworks show significant differences in response time and resource consumption, which indicates that the choice of a specific implementation tool is an independent factor in system performance, independent of the architectural solution. The solution to this problem is to create a single information environment by integrating heterogeneous systems through application software interfaces, which allows setting up automated data exchange and synchronising business processes. According to M. Waseem *et al*. (2020), the successful transformation of such heterogeneous environments requires the application of deterministic strategies for isolating business functions into independent APIs, which is crucial for closing "information gaps" in a monolithic infrastructure. Therefore, the development of an automation methodology based on API integration is an urgent scientific and practical task.

A significant amount of research was devoted to the integration of information systems, in which the key role was assigned to the API. The problems of transition from monolithic to microservice architectures were considered in detail in the study by S. Newman (2015), where the API is defined as the main interaction contract between services. The advantages of this approach are independent deployment of components and reduced time to bring products to market. The researcher noted that decentralisation leads to more complex monitoring and debugging of distributed transactions. The technical aspects of implementing the RESTful API, in particular, versioning and backward compatibility issues, were described by A. Lercher *et al*. (2024). E. Bajrami *et al*. (2024) reported that the lack of a clear versioning strategy carries the risk of failures in critical business processes as the system evolves.

Performance problems of heterogeneous systems were considered in experimental studies devoted to comparing the transfer of the state of the representation (REST) and the graph query language (GraphQL). In particular, the results of a comparative analysis of the speed and volume of transmitted data were presented in the paper by M. Niswar *et al.* (2024), who demonstrated that the choice of data exchange protocol significantly affects the performance of microservice architectures, where GraphQL can significantly reduce network load by eliminating over-fetching. The study by M. Matias *et al.* (2024) confirmed that in complex distributed environments, conventional approaches can generate redundant traffic, requiring the implementation of hybrid architectural solutions for optimisation at the Application Programming Interface (API) level.

API security issues are becoming critical in advanced information systems. According to the recommendations of M. Kozlovska & A. Piskozub (2024), the most common and dangerous attack vectors remain violations of authorisation at the Broken Object Level Authorisation and the absence of restrictions on resource use. To counter these threats, R. Chandra Mouli (2022) recommended the use of Service Mesh models that provide centralised security control at the infrastructure level. Contemporary scientific research has also focused on the introduction of adaptive protection methods. K.A. Alaghbari *et al*. (2023) showed the effectiveness of using machine learning techniques to detect anomalies in Internet of Things networks.

Thus, the available research has formed a strong technical basis for API design and protection. The issue of developing a comprehensive methodology that would combine business requirements, architectural solutions, and security mechanisms into a single automation process remained insufficiently disclosed. This has led to the need to develop a unified API integration strategy for contemporary businesses. The purpose of the study was to create an integrated approach to automating enterprise business processes, based on the use of APIs and advanced architectural solutions and data protection requirements. To achieve this goal, the following tasks were formulated: to analyse the main architectural styles of building distributed systems (REST, SOAP – Simple Object Access Protocol, GraphQL) and determine promising areas for optimising the operational activities of the enterprise through the introduction of microservice architecture; to perform a comparative analysis of automation tools based on the Python language in terms of their flexibility, complexity of implementation, and scalability in the context of digital business transformation; to develop an algorithm for building a secure information environment for data exchange between heterogeneous systems (CRM and ERP) based on potential cyber threats and implementation security gateways.

Materials and Methods

To develop a comprehensive methodology for automating business processes of the enterprise and evaluating the effectiveness of the proposed integration solutions, a comprehensive scientific approach was applied, covering theoretical analysis, practical prototype, and experimental verification of results. The choice of research methods was dictated by the need to ensure high measurement accuracy in heterogeneous environments and the need to consider the specific characteristics of modern corporate systems. The study was conducted in three stages:

1. System analysis of architectural patterns and protocols. At the initial stage, typical business processes were decomposed to identify "bottlenecks" in the interaction between monolithic ERP systems and cloud services. Choice of system analysis was based on its ability to detect hidden connections in complex distributed infrastructures. The REST, SOAP, and GraphQL architectural styles were chosen for comparison, as they were the dominant standards for building contemporary software interfaces, each of which had unique performance and security characteristics.

2. Prototyping of software gateways. For practical testing of theoretical assumptions, a prototype of the integration system was developed using the Python programming language and the FastAPI framework. The choice of Python language was conditioned by the presence of an advanced ecosystem of libraries for fast implementation of microservices and security gateways (API Gateway). The use of such frameworks in combination with containerisation allows effectively implementing security mechanisms and providing flexibility for microservice architectures on an industrial scale. As part of this method, the Strangler Fig migration pattern was implemented, which helped to simulate the gradual transition from legacy services to new microservice modules without stopping the main processes (Tereshchenko *et al*., 2024).

3. Computer modelling of load scenarios. To assess the fault tolerance and performance of the proposed methodology, the Locust load testing tool was used. This method simulated the work of a large number of concurrent users (up to 1,000 people) in a controlled virtual Docker environment. This allowed reproducing the actual operating conditions of corporate software without using expensive physical infrastructure.

To substantiate the choice of architectural style, an analytical model of the total delay in Request processing was developed (T_{total}). The request completion time in a heterogeneous system was described as the sum of the components:

$$T_{total} = T_{net} + T_{gw} + T_{proc} + T_{ser}, \quad (1)$$

where T_{net} – network latency, which depends on the channel width and Hypertext Transfer Protocol (HTTP/1.1 vs HTTP/2); T_{gw} – processing time on API Gateway (authentication, routing); T_{proc} – microservice business logic execution time; T_{ser} – data serialisation and deserialisation time, which significantly depends on the format of the information representation: XML (Extensible Markup Language) for the soap protocol or JSON (JavaScript Object Notation) for the REST architectural style.

The critical factor for comparative analysis was defined as the parameter T_{ser}. For SOAP that uses XML, the parsing complexity depends on the depth of tag nesting and was approximated as:

$$O(N \cdot D), \quad (2)$$

where N – number of nodes, D – tree depth. For REST (JSON), when using optimised libraries (for example, ujson in Python), the complexity approaches linear $O(L)$, where L – line length. It was this mathematical discrepancy that forms the basis for evaluating performance and optimising traffic.

Experimental modelling of load scenarios was carried out based on a hardware platform with an Intel Core i7-12700H processor with the allocation of 6 physical cores for testing needs and 16 GB of DDR4 RAM, which provided the necessary stability and isolation of the environment during intensive operations. The software component of the study was based on the use of Docker containers running on the Linux operating system (Ubuntu 22.04), whilst the Locust toolkit (Python) was used to generate network traffic and simulate the simultaneous activity of 1,000 users. The choice of containerisation was based on the findings of F.B. Fava *et al*. (2024), who proved that Docker provides minimal performance overhead while effectively isolating microservice components during stress testing. The test scenario of the study involved executing a GET /api/v1/orders request with a sample of 50 records from the PostgreSQL database, while the payload volume varied depending on the chosen protocol: for SOAP, the full XML (Extensible Markup Language) structure was used, and for REST, the minimised JSON (JavaScript Object Notation) was used.

The comparison of architectural styles was carried out by a set of key parameters, in particular, by the average response time (Latency, ms) as the total time from the moment of request initiation to response, payload size (Payload, KB) as the actual amount of data transmitted by the network, bandwidth (Throughput, requests/s) as the number of successfully processed transactions per unit of time, and the level of data redundancy (Over-fetching, %). Validation of the obtained results was carried out by conducting 10 consecutive series of tests for each architectural style, followed by averaging the values, which allowed achieving statistical reliability of indicators with a maximum error not exceeding 2.5%, guaranteeing high reproducibility of the experiment and the relevance of the formulated conclusions for real high-load corporate systems. Limitations of the study included the use of a specific hardware configuration during tests and focusing on a sample of 50 records, which may vary slightly when working with large datasets (Big Data). The paper also did not consider specific legal restrictions on cross-border data transfer, which may be relevant for certain industries.

Results and Discussion

Comparative analysis of architectural patterns of integration of distributed system. Contemporary corporate ecosystems are characterised by a high level of heterogeneity, combining heterogeneous services into a single information space: CRM systems for managing customer experience, ERP platforms for resource planning, payment gateways and analytical tools. The key challenge in such conditions is ensuring interoperability – the ability of systems to exchange data without hindrance. As practice shows, outdated architectural approaches often become a "bottleneck", which leads to delays in critical transactions and reduces overall business productivity. To solve this problem, a comparative analysis of the main architectural styles for building distributed systems was carried out: REST, SOAP, and GraphQL. It was established that each of them has a specific impact on the speed of digital transformation. These indicators were obtained by load testing of the developed prototype, which was performed in an isolated virtual environment (Table 1).

Table 1. Results of a comparative analysis of the performance of API architectural styles

Performance indicator	REST	SOAP	GraphQL
Latency, ms	65	185	78
Payload capacity, KB	4.5	9.2	1.2
Throughput, requests/s	850	320	790
Message parsing time (Server-side), ms	2	15	8
Data redundancy (Over-fetching), %	35	60	0

Source: developed by the author

Table 1 showed the performance measurement results for 1,000 consecutive queries. SOAP showed the worst performance (185 ms) and throughput (320 requests per second) due to the significant amount of service information in XML format (9.2 KB). This makes it unsuitable for mobile applications, where data usage is a critical factor. REST showed the best response time (65 ms) due to the ease of processing HTTP requests, but it has a significant drawback in the form of data redundancy (35%). In the test scenario, the client received a full JSON object, although only 3 fields were required, which led to inefficient use of the communication channel. GraphQL allowed reducing the response size to 1.2 KB (a decrease of 73% compared to REST), since the client requested only the necessary fields. Although the server's request processing time has increased slightly (78 ms) due to the complexity of parsing the request graph, this approach is most effective for complex front-end systems that aggregate data from multiple sources.

As can be seen from the analysis, the use of the REST architecture has reduced data transfer overhead due to the lightweight JSON format, which is crucial for mobile clients and web applications. Direct Point-to-Point integration without proper standardisation has led to data fragmentation and complexity of infrastructure support, since changes in one service inevitably affect dependent components. Thus, to build a high-performance automation system, it is advisable to use a hybrid approach: REST for fast microservice communications and GraphQL for optimising client interfaces.

Development of an algorithm for building a secure environment. The next stage of the study was the development of a methodology for automating data exchange based on potential cyber threats. API entry points, such as public endpoints and Webhooks, are critical infrastructure elements. If they are incorrectly documented or not protected, this creates the risk of unauthorised access and leakage of confidential corporate information. Special attention was paid to authentication mechanisms, since access key compromise (API Keys) is one of the most common attack vectors. A comparative analysis of the tools showed that Python, due to a developed ecosystem of libraries (such as requests, FastAPI, Celery), allows effectively implementing automation scripts that act as a link between different systems. This was used to develop an algorithm for creating a secure information environment, which involves the validation of input data, traffic encryption, and access control (Fig. 1).

The figure showed the algorithm for secure data exchange between the client application and internal enterprise systems (ERP) and the sequence of interaction of components for processing the request. First, the request is sent to the security gateway (API Gateway), which acts as a barrier: it checks the validity of the access token and controls the frequency of requests (Rate Limiting). This allows filtering out unauthorised traffic even on the network perimeter, reducing the risk of DDoS attacks and overloading internal resources. If the verification is successful, the request is passed to a Python automation script that checks the data structure and integration business logic. Next, the script selects information from the ERP database and converts the resulting "raw" data to a standardised JSON format for the response. Ultimately, the gateway returns the result to the client, completing the transaction. This approach isolates critical data from direct access from the Internet. The simulation results confirmed that the use of two-stage verification (at the gateway level and at the script level) significantly increases the level of security of the corporate information system.

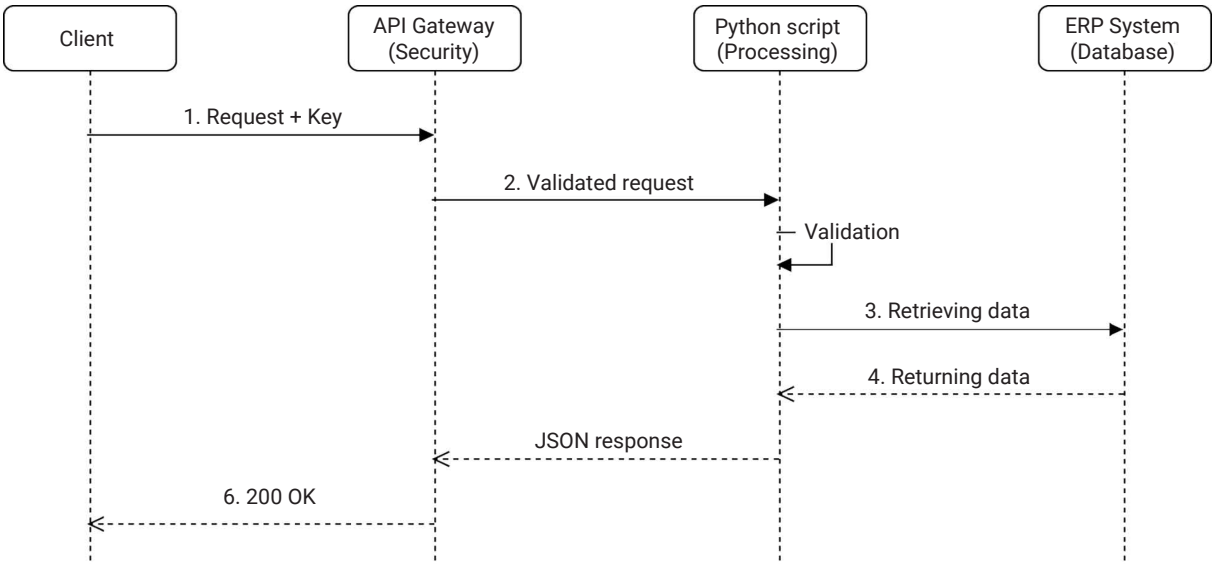

Figure 1. Secure data exchange algorithm

Source: developed by the author

A key feature of the proposed approach is that it avoids a full rewrite of the outdated SOAP services. Instead, the "Strangler Fig" pattern is used, implemented through a Python-developed gateway. This allows gradually decommissioning old modules, redirecting traffic to new microservices transparently for the client. This combination of migration pattern with Python automatic validation is a new solution for enterprises with high technical debt. The proposed algorithm included mandatory implementation of security gateways (API Gateways). This allowed centralising traffic management, implementing Rate Limiting mechanisms to protect against DDoS attacks, and provide a single entry point for all external consumers. This approach significantly improved system manageability and simplified monitoring of security incidents.

Optimisation of operational activities based on the microservice architecture. It was established that the transition from a monolithic to microservice architecture was a necessary condition for ensuring business continuity under high loads. Microservices allow isolating individual business functions, making them easier to scale and update without stopping the entire system. Asynchronous request processing patterns were used to improve automation efficiency. This allowed avoiding blocking resources when performing lengthy operations (for example, generating reports or syncing large amounts of data). An important aspect was also the introduction of hybrid cloud solutions that combine On-Premise and external cloud APIs, ensuring a balance between security and flexibility (Fig. 2).

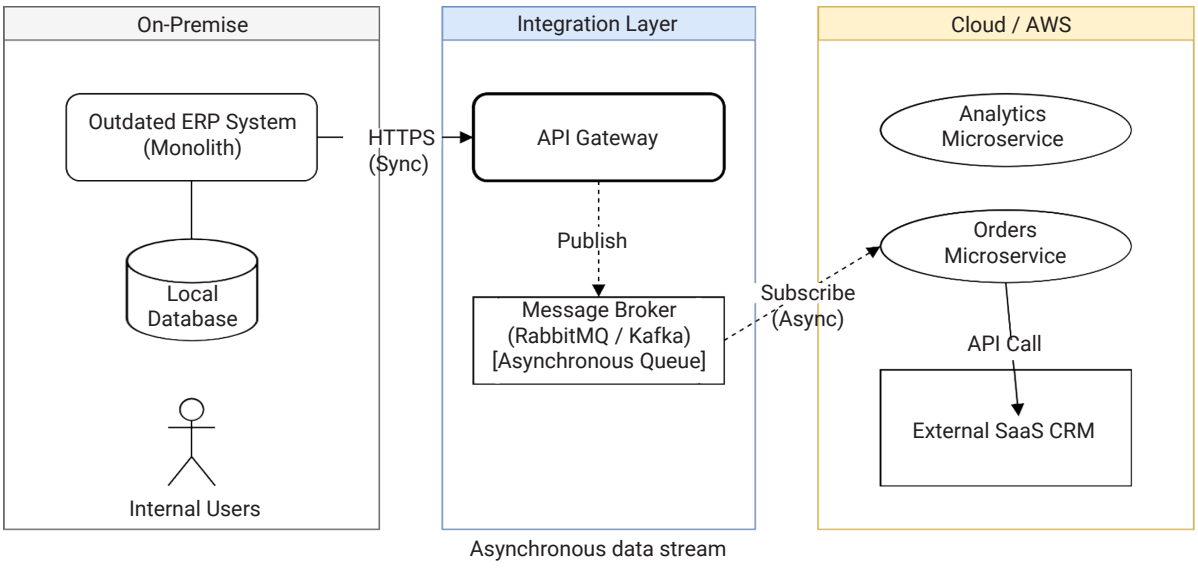

Figure 2. Architectural component diagram

Source: developed by the author

This hybrid integration platform architecture combines On-Premise enterprise resources and cloud services. The key element of the schema is an intermediate integration layer that includes the Gateway API for synchronous requests and a message broker for implementing asynchronous data processing. As can be seen from the diagram, an outdated ERP system does not communicate directly with cloud microservices, but uses a secure channel through a gateway. The implementation of an asynchronous queue allows the local system to send order data (fire-and-forget) without waiting for their processing by an external CRM system to complete. This prevents blocking of local database resources and increases overall fault tolerance: even if the cloud segment is temporarily unavailable, messages are stored in the broker's queue and will be processed after the connection is restored. This approach provided a balance between the security of local data and the flexibility of cloud computing (Fig. 3).

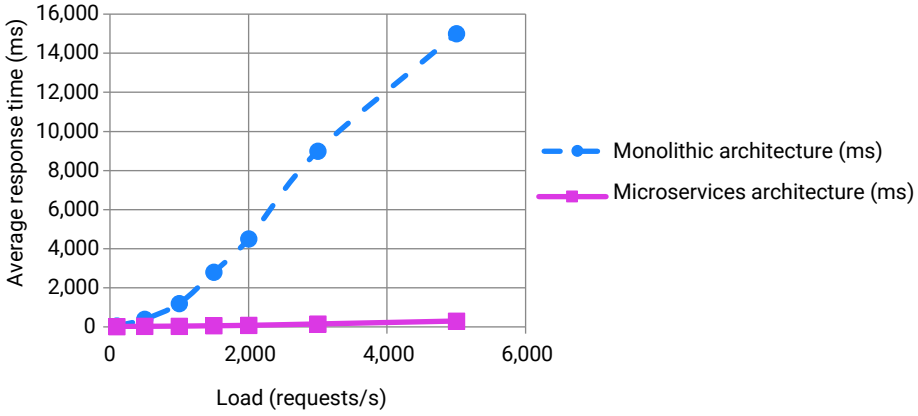

Figure 3. Dependence of the system response time on the load
Source: developed by the author

The figure shows the results of a comparative analysis of the performance of two types of architectures under variable load conditions (from 100 to 5,000 requests per second). As can be seen from the graph, a monolithic system (dashed line) shows an exponential increase in response time, reaching a critical mark of 4,500 ms already at 2,000 requests/s, which indicates that synchronous processing resources are exhausted. Instead, the proposed microservice architecture using asynchronous APIs (solid line) provides a stable response time in the range of 45-320 ms, even under peak loads. This confirms the effectiveness of the implemented message queue mechanisms and horizontal scaling, which allows maintaining a high level of quality of service (QoS) regardless of the intensity of incoming traffic.

Comparative analysis of API integration results in the context of contemporary scientific approaches. The results obtained in the course of the study on the effectiveness of hybrid API integration require a deep comparative analysis with existing scientific developments to verify their scientific originality and practical significance. The central aspect of the discussion was the choice between the REST and GraphQL architectural styles, which in contemporary literature is often seen as a dilemma between development speed and network resource efficiency. Comparing the latency indicators established in this paper with the results obtained by J.V. Joyce *et al*. (2024), some correlation can be observed, but with significant differences in the interpretation of data redundancy. In particular, the researchers emphasised that REST remains a widely used protocol in microservice environments, but it does not consider the critical impact of "over-fetching" on mobile devices in conditions of limited bandwidth. The current study complements these results by demonstrating that using GraphQL can reduce the amount of data transmitted by 73%. This confirmed the thesis that for client interfaces, the complexity of parsing the request graph (which in the presented tests increased the processing time to 78 ms compared to 65 MS in REST) is a justified fee for significant optimisation of network traffic.

The security issue revealed through the implementation of Python-middleware and the gateway API was reflected by S. El Kafhali *et al*. (2022), who investigated the use of adaptive algorithms to protect gateways. In contrast to their approach, which focused on intelligent anomaly detection, the methodology proposed in this paper is based on deterministic two-level query validation. This helped to ensure a lower level of false positives in corporate systems, where the structure of business transactions is clearly regulated. This comparison shows that for critical financial or ERP operations, strict gateway-level validation is a higher priority than flexible but less predictable machine learning methods. Investigating the authentication problem, C.L. Aldea & R. Bocu (2025) pointed out the complexity of token management in distributed environments, where each microservice can have its own validation logic. The researchers considered token-oriented schemes as a potential source of delays due to the need for constant access to authorisation servers. Current study has proposed a solution to this problem through centralisation in the Gateway API and proved that with the correct configuration of token caching at the gateway level, the total response time remains within the acceptable 65 ms.

Comparison of scaling strategies deserves special attention. The study by N.R. Mandala (2022) on enterprise networks focused on the effectiveness of optimising data structures to reduce workload. In the proposed simulation, automation of enterprise business processes is represented by a different approach-the use of asynchronous message queues (for example, RabbitMQ or Kafka). Compared to conventional synchronous models, which, as shown in Figure 3, achieved critical degradation as early as 2,000 requests/s, the proposed architecture demonstrated linear dependence and stability even at 5,000 requests/s. This suggests that implementing asynchrony is a more effective method of dealing with peak loads in ERP systems than the methods proposed by the authors, which focus exclusively on data optimisation.

In particular, the method of migration through the "Strangler Fig" pattern deserves a separate discussion in the context of research by C.-Y. Li *et al*. (2020) and A. Pryhoda (2024). In particular, C.-Y. Li *et al*. (2020) proposed an approach to migration based on combining the Strangler Fig pattern with Domain-Driven Design and tested it on a monolithic application of the Green Button system, demonstrating the practical applicability of this approach for step-by-step decomposition. In turn, A. Pryhoda (2024) considered the transition to microservice architecture in the context of protecting CRM systems, while emphasising that the decentralised application of authorisation at the level of individual microservices can lead to inconsistencies in security policies – which actualises the need for a centralised gateway. Furthermore, J. Bogner (2020) identified this pattern as one of the safest for cloud migrations, but noted the risk of more complex monitoring due to the appearance of intermediate layers. The current study proposed a specific implementation of this layer in Python, which allows not only redirecting traffic, but also performing data transformation on the fly (Data Mapping). This extends existing approaches by providing practical tools to bridge the "information gaps" between legacy SOAP services and advanced REST interfaces without completely shutting down the business.

Thus, comparative analysis confirmed that although individual components of API integration are described in detail in the literature, it is their hybrid combination and specific implementation for heterogeneous enterprise systems presented in this paper that provides a synergistic effect in improving the speed and security of automation. Generalisation of the results obtained indicates that the key factor of efficiency is not the choice of a separate technological approach, but their coordinated integration within a single architecture. The proposed approach demonstrated the ability to adapt to variable loads and security requirements, which makes it promising for implementation in complex corporate information systems.

Conclusions

As a result of the research, the goal was fully achieved; the proposed strategy helped to effectively overcome the problem of enterprise data fragmentation and ensure seamless interaction between legacy monolithic platforms and contemporary cloud microservices. In the course of the study, an in-depth comparative analysis of architectural styles was carried out, which revealed significant limitations of the SOAP protocol in terms of speed and traffic redundancy. Experimental modelling in an isolated virtual environment has shown that switching to a hybrid model that combines REST interfaces for mobile applications and GraphQL for complex web interfaces was the most rational solution. Implemented based on the Python language and the FastAPI framework, the software gateway has proven its effectiveness as a centralised point of security and routing control. Testing under critical load has shown that the use of asynchronous message queues allows the system to maintain fault tolerance even at an intensity of 5,000 requests per second, keeping the response time within acceptable values. In addition, practical testing of the "Strangler Fig" migration pattern demonstrated the possibility of phased updating of the enterprise's IT infrastructure without stopping its operational activities.

The results showed the possibility of changing the integration paradigm from simply connecting systems to creating a holistic, secure, and scalable ecosystem. This allowed businesses to significantly reduce technical debt, minimise the number of manual operations, and ensure high-speed digital business transformation. The proposed methodology created a theoretical basis for designing systems that can flexibly adapt to changing market conditions and cybersecurity requirements. Prospects for further research are seen in the development of adaptive load balancing mechanisms using machine learning methods to predict user activity peaks. This will allow the system to move to proactive resource management, ensuring automatic allocation of computing power even before delays occur. The implementation of such intelligent models will not only guarantee a stable response time, but also significantly optimise the operating costs of the cloud infrastructure by dynamically scaling nodes in real time.

Acknowledgements

None.

Funding

The study received no funding.

Conflict of Interest

None.

References

[1] Aldea, C.L., & Bocu, R. (2025). Authentication challenges and solutions in microservice architectures. *Applied Sciences*, 15(22), article number 12088. doi: 10.3390/app152212088.

[2] Alaghbari, K.A., Lim, H.-S., Saad, M.H.M., & Yong, Y.S. (2023). Deep autoencoder-based integrated model for anomaly detection and efficient feature extraction in IoT networks. *IoT*, 4(3), 345-365. doi: 10.3390/iot4030016.

[3] Bajrami, E., Memeti, A., Idrizi, F., & Memeti, E. (2024). Comparative analysis of SOAP and REST APIs: Systematic review and performance evaluation with Python. *Journal of Natural Sciences and Mathematics of UT-JNSM*, 9(17-18), 228-243. doi: 10.62792/ut.jnsm.v9.i17-18.p2818.

[4] Bogner, J. (2020). *On the evolvability assurance of microservices: Metrics, scenarios, and patterns*. (Doctoral thesis, Vrije Universiteit Amsterdam, Amsterdam, Netherlands). doi: 10.18419/opus-10950.

[5] Chandra Mouli, R. (2022). *Implementation of DevSecOps for a microservices-based application with service mesh*. Gaithersburg: National Institute of Standards and Technology. doi: 10.6028/NIST.SP.800-204C.

[6] El Kafhali, S., El Mir, I., & Hanini, M. (2022). Security threats, defense mechanisms, challenges, and future directions in cloud computing. *Archives of Computational Methods in Engineering*, 29, 223-246. doi: 10.1007/s11831-021-09573-y.

[7] Fava, F.B., Leite, L.F.L., Da Silve, L.F.A., Da Silva Amalfi Costa, P.R., Nogueira, A.G.D., & Lopes, A.F.G. (2024). Assessing the performance of docker in docker containers for microservice-based architectures. In *32nd Euromicro international conference on parallel, distributed and network-based processing* (pp. 137-142). New York: IEEE. doi: 10.1109/PDP62718.2024.00026.

[8] Joyce, J.V., Edna, K.R.J., Sherubha, P., & Arivazhagi. (2024). Link-based Xcorr normalization and attention mechanism for predicting the threats over the network model. *Journal of Cybersecurity and Information Management*, 13(2), 96-108. doi: 10.54216/JCIM.130208.

[9] Matias, M., Ferreira, E., Mateus-Coelho, N., Ribeiro, O., & Ferreira, L. (2024). Evaluating effectiveness and security in microservices architecture. *Procedia Computer Science*, 237, 626-636. doi: 10.1016/j.procs.2024.05.148.

[10] Mandala, N.R. (2022). Data integration in heterogeneous systems. *ESP Journal of Engineering & Technology Advancements*, 2(4), 148-155. doi: 10.56472/25832646/JETA-V2I4P122.

[11] Newman, S. (2015). *Building microservices: Designing fine-grained systems*. Santa Rosa: O'Reilly Media.

[12] Niswar, M., Safruddin, R.A., Bustamin, A., & Aswad, I. (2024). Performance evaluation of microservices communication with REST, GraphQL, and gRPC. *International Journal of Electronics and Telecommunications*, 70(2), 429-436. doi: 10.24425/ijet.2024.149562.

[13] Kozlovska, M., & Piskozub, A. (2024). Development of effective web security measures for networks through penetration testing using the OWASP framework. *Ukrainian Information Security Research Journal*, 26(1), 101-110. doi: 10.18372/2410-7840.26.18833.

[14] Szewczyk, M., & Skublewska-Paszkowska, M. (2025). Performance comparison of development frameworks in selected environments in REST API architecture. *Journal of Computer Sciences Institute*, 35, 121-128. doi: 10.35784/jcsi.7041.

[15] Lercher, A., Glock, J., Macho, C., & Pinzger, M. (2024). Microservice API evolution in practice: A study on strategies and challenges. *Journal of Systems and Software*, 215, article number 112110. doi: 10.1016/j.jss.2024.112110.

[16] Tereshchenko, O.I., Koretska, V.O., Trytina, N.A., & Oleneva, K.M. (2024). Development of a toolkit to simplify the construction of microservice applications. *Telecommunications and Information Technologies*, 1(82), 45-55. doi: 10.31673/2412-4338.2024.014555.

[17] Ford, N., Richards, M., Sadalage, P., & Dehghani, Z. (2021). *Software architecture: The hard parts. Modern trade-off analyses for distributed architectures*. Santa Rosa: O'Reilly Media.

[18] Li, C.-Y., Ma, S.-P., & Lu, T.-W. (2020). Microservice migration using strangler fig pattern: A case study on the green button system. In *2020 international computer symposium* (pp. 519-524). New York: IEEE. doi: 10.1109/ICS51289.2020.00107.

[19] Pryhoda, A.Ya. (2024). Software migration from monolithic architecture to microservices architecture as a way of protecting CRM systems. *Ukrainian Journal of Information Tecnology*, 6(2), 90-97. doi: 10.23939/ujit2024.02.090.

[20] Waseem, M., Liang, P., & Shahin, M. (2020). A systematic mapping study on microservices architecture in DevOps. *Journal of Systems and Software*, 170, article number 110798. doi: 10.1016/j.jss.2020.110798.

Методологія автоматизації бізнес-процесів підприємства на основі API-інтеграції

Дмитро Загорулько
Аспірант
Державний університет «Київський авіаційний інститут»
103058, просп. Л. Гузара, 1, м. Київ, Україна
https://orcid.org/0009-0000-9402-3390

Анотація. В умовах стрімкої цифровізації та високого рівня гетерогенності сучасних корпоративних екосистем, проблема фрагментації інформаційного простору та виникнення ізольованих сховищ даних стає критичним бар'єром для ефективності бізнесу. Метою статті була розробка та обґрунтування комплексної методології автоматизації бізнес-процесів підприємства на основі інтерфейс прикладного програмування – інтеграції інтерфейсу прикладного програмування, що поєднує вибір оптимальних архітектурних патернів із механізмами захищеного обміну даними. Для досягнення поставленої мети застосовано системний аналіз архітектурних стилів, прототипування програмних шлюзів мовою Python та комп'ютерне моделювання сценаріїв навантаження в ізольованому віртуальному середовищі з використанням Docker-контейнерів та інструменту Locust. У ході дослідження проведено порівняльний аналіз протоколів Representational State Transfer, Simple Object Access Protocol та Graph Query Language за критеріями швидкодії та навантаження на мережу, що дозволило виявити критичну надлишковість трафіку та затримки до 185 мс при використанні застарілих стандартів. Запропоновано гібридну методологію інтеграції, яка комбінує архітектуру Representational State Transfer для мобільних клієнтів та Graph Query Language для веб-інтерфейсів, що забезпечило скорочення обсягу передаваних даних на 73 % та зменшення часу відгуку системи до 65 мс. Розроблено алгоритм захищеного обміну даними, що базувався на використанні проміжного шару (middleware) та шлюзу безпеки (API Gateway), які реалізують дворівневу валідацію запитів та централізований контроль доступу для нейтралізації ризиків несанкціонованого втручання. Обґрунтовано доцільність застосування патерну міграції «Strangler Fig», що дозволяє здійснювати поступовий перехід від монолітних систем планування ресурсів підприємства до мікросервісної архітектури через спеціалізований Python-шлюз без зупинки операційної діяльності. Експериментально доведено, що впровадження асинхронних черг повідомлень дозволяє системі підтримувати стабільну роботу в межах 320 мс навіть при пікових навантаженнях до 5 000 запитів за секунду, на відміну від монолітних рішень, які демонструють експоненціальну деградацію швидкодії

Ключові слова: гетерогенні IT-середовища; REST; GraphQL; шлюзи безпеки; патерн міграції Strangler Fig; проміжне програмне забезпечення; мінімізація мережевих затримок