Universal Library of Multidisciplinary Research Article

ISSN: 3064-9986 | Volume 3, Issue 1

Open Access | PP: 93-99

DOI: https://doi.org/10.70315/uloap.ulmdi.2026.0301009

Adaptation of Distributed Microservice System Patterns for In-Process Interaction in Monolithic Applications

I. D. Butorin

Colibri Scientific & Engineering Organization LLC, C/C++ Senior Software Engineer, Saint Petersburg, Russia.

Abstract

This article continues a series of studies on the thread-independent architecture of multithreaded C++ systems. It extends the previously proposed "Node Manager" model to support the architectural adaptation of microservice patterns within a single process. The paper examines an engineering problem typical of large monolithic applications, where direct inter-module calls, pervasive thread coupling, and the growing number of synchronization dependencies complicate system extensibility, failure localization, and reproducible debugging. The study aims to identify the elements of microservice discipline that retain practical value after the network boundary is removed and to interpret them through the mechanisms already introduced: "Node," "Node Manager," "Queue Segregation," "Distributed Tracing," and "Failure Handling." The empirical basis of the study consists of research on microservice patterns, event-driven architectures, non-blocking synchronization, observability, and operational metrics of software systems. The methods employed include comparative architectural analysis, conceptual comparison of distributed and in-process execution, and engineering reconstruction of patterns through the "Node Manager" core. It is shown that mechanisms such as the event bus, broker-mediated interaction, service discovery, distributed tracing, and overload-control techniques take on their own form within a monolith and remain suitable for the system's evolutionary growth without an early transition to network-level decomposition.

Keywords: *Thread Independence, "Node Manager," Modular Monolith, Microservice Patterns, C++, Event-Driven Architecture, Distributed Tracing, Lock-Free Queue, Publish/Subscribe, Backpressure.*

INTRODUCTION

Large-scale C++ systems increasingly retain a monolithic deployment model even though, in their internal structure, they are already approaching distributed platforms in terms of the number of modules, the intensity of event exchange, and the complexity of coordination. In such an environment, direct method invocation between components gradually ceases to be a convenient default model of interaction. It preserves a low local cost while transferring to application code knowledge about the execution thread, lock ordering, reentrancy constraints, and the actual lifecycle of dependent modules. As the system grows, such an arrangement strengthens coupling along two lines simultaneously: through interfaces and through the execution context.

In the first article of the series, a paradigm of thread independence was proposed in which the unit of business logic is a "Node." At the same time, inter-module coordination is delegated to the "Node Manager." That paper described the foundation of the solution: a "hub-and-spoke" topology, centralized lock-free queues, a publish/subscribe mechanism, deterministic event handling, the "Single Serialized Queue" and "Multi-Queue" modes, the three-stage cycle "Initialization – Registration – Execution," together with engineering constraints associated with backpressure and loss of causality. The present article continues that line of inquiry and examines the architectural productivity of adapting distributed microservice patterns inside a monolithic C++ application.

Interest in such a transfer stems from the practical need for an intermediate form between a tightly coupled monolith and an early transition to a network of services. For many production systems, full microservice decomposition at an early stage proves excessively costly because operational overhead, observability requirements, network fault tolerance demands, and data consistency concerns begin to outpace the actual benefits. At the same time, the discipline of microservice architecture itself contains a set of engineering techniques that remain useful even without a network boundary: event decomposition, indirect addressing, handler registration, thematic channel separation, tracing of causal chains, and localization of coordination failures.

Citation: I. D. Butorin, "Adaptation of Distributed Microservice System Patterns for In-Process Interaction in Monolithic Applications", Universal Library of Multidisciplinary, 2026; 3(1): 93-99. DOI: https://doi.org/10.70315/uloap.ulmdi.2026.0301009.

The purpose of this article is to provide an engineering interpretation of which microservice patterns preserve architectural value after being moved into a single process and how they are expressed through the mechanisms of the already-introduced "Node Manager." In accordance with this purpose, three objectives are addressed. The first objective is to identify the transferable semantics of microservice patterns after the network boundary has been removed. The second objective focuses on describing their in-process implementation through "Node," registration, queues, and type-based routing. The third objective concerns the engineering consequences of such a transfer for C++ architecture, debugging, tracing, and failure handling.

The scientific novelty of the study lies in treating microservice patterns as a set of architectural disciplines that can be implemented within a single process using a thread-independent core. Unlike general literature reviews of microservices, the present work focuses on directly mapping these patterns to elements of the previously proposed architecture: "Node Manager," "Queue Segregation," "Failure Handling," publish/subscribe, "Distributed Tracing," and "Event Bus."

MATERIALS AND METHODS

The literature search was conducted using publications from 2020 to 2026 indexed in international academic databases and specialized publishing collections in software engineering and software architecture. At the initial selection stage, studies were considered if they addressed microservice patterns, event-based interaction models, non-blocking synchronization, observability, tracing, and the engineering implications of architectural choices. During the screening stage, publications with excessively general review content, works lacking an explicit architectural component, and materials that did not support a comparison between distributed and in-process execution were excluded.

The empirical basis consisted of publications covering six interrelated areas. The first group comprised studies on microservice architecture, its advantages, limitations, and strategies for pattern selection [1–3]. The second group included works on engineering solutions for microservice systems, addressing design decisions at the API, performance, and architectural tooling levels [4, 5]. The third group concerned actor-like coordination and event-based organization of interaction [6]. The fourth group covered non-blocking synchronization and MPSC/SPSC structures relevant to queue implementation in the "Node Manager" [7, 8]. The fifth group focused on observability, tracing, and the analysis of debugging episodes [9, 10]. The sixth group addressed operational metrics of development teams, enabling assessment of the influence of architectural form on integration and process reproducibility [11]. Review publications served as a background frame, while the analytical foundation relied on works that described patterns and mechanisms at the level of engineering application.

The methodological part was based on comparative architectural analysis and on a conceptual comparison of two execution contours: distributed inter-service exchange and in-process coordination. A separate step was devoted to aligning the terminology of the present article with that of the first publication so that the interpretation of patterns would proceed through the already introduced entities "Node," "Node Manager," "Single Serialized Queue," "Multi-Queue," "record-and-replay," "backpressure," and "loss of causality." Next, engineering reconstruction was applied: each pattern was examined through its target function in a microservice system, after which the semantics that retained practical meaning within a single process were identified. The final stage consisted of an analytical generalization of the architectural implications for C++ implementation, observability, failure management, and the trajectory of the modular monolith's further development.

RESULTS

The first article in the series established thread independence as an architectural invariant: business logic within a "Node" is not bound to the thread that originated the request. At the same time, inter-module coordination is entirely delegated to the "Node Manager." For the present article, this thesis serves as the point of departure. It defines the condition under which the internal architecture of a monolith can adopt the discipline of distributed systems without imposing the network as a mandatory execution boundary. In other words, the transfer of microservice patterns into a single process becomes possible only when direct invocation has ceased to function as a universal mode of interaction and when application code no longer depends on thread topology.

A recurring idea in the microservices literature is that architectural usefulness stems from manageable boundaries, independence in team-driven evolution, explicit exchange formats, and reduced hidden coupling [1–3]. In a single-process environment, the network layer disappears. Yet, the need for the same disciplines remains: formalized messages, indirect addressing, separation between sender and receiver, and limits on the impact of another module's lifecycle on one's own business logic. It is precisely here that the "Node Manager" architecture acquires a second dimension. The first article addressed the problem of concurrency. In the present study, it emerges as a carrier of microservice discipline within a modular monolith.

The "Broker/mediator" pattern in a distributed system supports indirect interaction between participants and moves coordination into a separate layer. In an in-process architecture, this function is assumed directly by the "Node Manager." Its meaning is not limited to that of a convenient call dispatcher. It forms a unified coordination contour in which a module publishes a message without knowing the actual consumer, the number of subscribers, or the thread of subsequent processing. In the microservices literature, precisely such separation between sender and receiver is associated with greater evolutionary resilience of the

system [1, 2]. For a C++ monolith, the practical implication is that changing the internal implementation of a single "Node" does not require cascading modifications to direct inter-module calls. As long as the message contract and the handler registration rules are preserved, neighboring modules continue to operate without dependency rewrites and without reconfiguration of thread interaction.

The "Event Bus" pattern in a distributed environment supports asynchronous propagation of events between independent participants. Once the network boundary is removed, its architectural semantics do not disappear; the carrier simply changes. In the previously proposed model, "Single Serialized Queue" and "Multi-Queue" function as in-process channels organized thematically. In the single-queue mode, the system receives a global event sequence with a deterministic processing order. Such a configuration is particularly appropriate when reproducibility and strict adherence to temporal order are valued. In the "Multi-Queue" mode, chronology is preserved within each channel, whereas partial divergence between channels is allowed. This mode is closer to the microservice practice of thematic flow segregation, where independent domain lines are processed in parallel and do not block one another [2–4]. For the "Node Manager," such a transfer is especially productive because it does not require a separate network bus. Yet, it already compels the architecture to think in terms of channels, message types, and rules of local load isolation.

From an engineering standpoint, this adaptation produces two concrete effects. First, a module that publishes a message no longer needs to know whether the recipient has already been initialized, which thread it operates on, or whether it is ready to handle the call synchronously. Second, concurrency is moved from application code into the infrastructure layer: processing order is determined by queues, subscription rules, and channel-draining policies. Research on non-blocking synchronization and MPSC structures confirms the practical value of shifting such complexity from business logic into a single infrastructural layer [7, 8]. For C++, this direction is especially significant because the direct transfer of concurrency concerns into user code rapidly increases the surface area for errors related to memory visibility, publication order, and spontaneous lock-based coupling.

The next transferable pattern is associated with "asynchronous dispatch." In a microservice environment, it separates the acceptance of a request from its subsequent processing. In a thread-independent architecture, the same principle is realized through the publish/subscribe mechanism and the controlled execution context already embedded in the "Node Manager." A command is published in one thread, whereas its actual processing is performed in the context assigned by the dispatching rules. The practical meaning of such a solution is that a module is no longer obliged to share the fate of the calling thread synchronously. It no longer inherits its locks, latency budget, or reentrancy constraints. Studies on microservice patterns and event-driven architectures show that precisely this weakening of immediate causal coupling usually opens the way to independent component evolution [1–3, 6]. Within a single process, it produces a familiar effect: application code is designed as message handling.

The "Service Discovery" pattern, once the network boundary has been removed, requires a more careful interpretation. In a microservice system, it serves to locate a recipient among dynamically existing services. Inside a single process, the need for network address resolution disappears, yet the idea of late binding remains relevant. In the proposed architecture, the "Registration" stage performs its function, and the "subscribe" method is invoked. A module does not receive a direct reference to its neighbor. It merely declares which message types it can handle. The "Node Manager" accumulates this correspondence map and, during the Execution phase, resolves routing according to message type and subscription rules. In terms of distributed patterns, this scheme reproduces the meaning of service discovery without the network layer: the recipient is determined by the registered capability to handle a particular contract [2, 5]. For a modular monolith, such a form of binding is especially useful during gradual restructuring of the system, when domain modules change in composition and boundaries. At the same time, direct dependency injection becomes difficult to manage.

For clarity, the architectural correspondences between microservice patterns and the elements of the "Node Manager" were presented in a separate diagram (see Fig. 1). It shows how late binding, handler registration, queue segregation, and tracing of causal chains are implemented within a single process without a network layer.

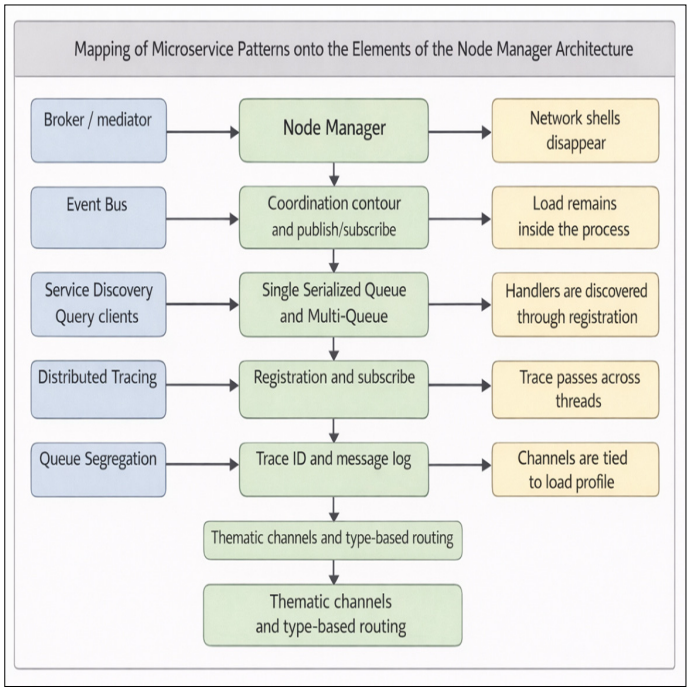

Figure 1. Mapping of microservice patterns onto the elements of the "Node Manager" architecture

Particular attention should be paid to "distributed tracing." In a network architecture, it is necessary to restore the causal chain of a request that has passed through several services and brokers. In the in-process architecture, the first article had already identified the problem of "loss of causality": the call stack breaks at the point where a message is placed into a queue, and the local debugger no longer provides an exhaustive picture of the event's origin. Here, the microservice discipline of observability is transferred with almost no semantic loss. An internal message requires a Trace ID that accompanies it during publication, routing, resubmission, and processing in another thread. Studies of debugging episodes and the monitoring of complex systems show that reconstructing message history is critical for reproducibility, defect localization, and the explainability of platform behavior [9, 10]. For the "Node Manager," such a transfer means that tracing ceases to be an optional convenience and becomes a mandatory part of the message protocol. Without it, thread independence remains functionally viable but loses engineering transparency.

The "Queue Segregation" pattern in microservice systems usually manifests as separate queues for commands, events, and service messages, and as the separation of load flows. Inside a single process, it corresponds to the choice between "Single Serialized Queue" and "Multi-Queue." In the first case, the entire system is subordinated to a common ordering. In the second, a controlled decomposition by channel types emerges. What matters is that this choice already influences the internal geometry of concurrency. If heavy commands and short control messages are combined in a single queue, the "Node Manager" itself becomes a source of false competition. If channels are separated by type, local sequence is preserved, while overall throughput rises and the probability of mutual slowdown between heterogeneous subsystems falls [3, 7, 8]. From this follows an engineering rule: queue segregation within a thread-independent architecture is appropriate for expressing domain boundaries and differences in load profile.

The problem of "Failure Handling" when transferring microservice patterns into a single process requires distinguishing between two distinct failure classes. In a distributed system, a substantial share consists of network and transport failures, message loss, and partial unavailability of a neighboring service. In a single process, the network disappears. Yet coordination failures persist: queue overflow, violations of registration rules, repeated delivery of a service command, degradation of a handler that delays channel draining, and the accumulation of backpressure. For that reason, in an in-process architecture, it is useful to distinguish between "business failure" and "coordination failure." The first arises at the domain logic level of the "Node" itself, for example, when a correctly delivered command cannot be processed. The coordination infrastructure generates the second and affects multiple modules simultaneously. Research on microservice strategies and architectural resilience emphasizes that the operational value of a pattern depends precisely on how early the architecture distinguishes between such types of disruption [2, 3, 11]. For the "Node Manager," this distinction leads to separate engineering measures: queue-depth limits, retry policies, separation of "poison messages" into a dedicated channel, and recording overload as an infrastructural event.

At the level of C++ implementation, the transfer of microservice discipline relies on the elements introduced in the first article, which now acquire an additional function. "Type erasure" and "std::function" support type-based routing without rigidly embedding handlers into the "Node Manager" code. Lock-free queues with correctly chosen memory barriers make the publish operation suitable for concurrent message publication with controlled synchronization cost [7, 8]. As a result, the architecture is built on two coordinated levels. The upper level defines the model of messages, subscriptions, and channels. The lower level ensures safe data visibility between producer and consumer without spreading locks into application code. Such a two-layered organization brings a thread-independent modular monolith closer to microservice discipline: outwardly, the system is still deployed as a single process, but internally it already follows the principles of exchange isolation, handler registration, and controlled asynchrony.

The "record-and-replay" mechanism introduced in the first article as a means of deterministic testing retains particular value. In microservice practice, reproducibility is complicated by distribution, network latency, and incomplete observability. In an in-process architecture, recording and replaying the message stream turns into a real instrument of operational control. If each message carries a trace identifier, a typed contract, and is recorded in the common channel sequence, then the system acquires a reproducible form of integration testing without the need to raise a complex network environment [9, 11]. At this point, thread independence ceases to be merely a way of combating race conditions. It becomes the foundation for controlled architectural evolution.

Table 1 shows a fundamental point: when microservice patterns are translated into a single process, their network shells disappear, but their architectural functions do not. On the contrary, in this form their constructive meaning becomes even more visible. The "Node Manager" does not imitate an external distributed environment. It extracts from it those disciplines that give the monolith a more manageable geometry of interaction: late binding, typed exchange, load channeling, instrumented asynchrony, and controlled observability.

Table 1. Mapping of Microservice Patterns onto the Elements of the "Node Manager" Architecture

Microservice Pattern	Preserved Semantics	Element of the Architecture from the First Article	Change After Transfer into a Single Process
"Broker/mediator"	indirect interaction between modules	"Node Manager"	The network disappears, while a unified coordination contour is preserved
"Event Bus"	event publication without knowledge of the consumer	publish, subscriptions, queues	The event remains in process memory, and the transport cost is reduced
"Service Discovery"	late binding between sender and handler	Registration, subscribe	The address is resolved through the subscription map
"Distributed Tracing"	restoration of the causal chain	Trace ID, message log, record-and-replay	The trace passes across threads without network transitions
"Queue Segregation"	separation of load flows	"Single Serialized Queue", "Multi-Queue"	The choice is made according to the domain and load profile of channels
"Failure Handling"	localization and classification of failures	backpressure, service commands, queue control	Coordination failure is separated from domain error

The set of correspondences considered above allows one more conclusion. A microservice pattern inside a single process is useful only if its transfer preserves structural economy. This defines the limit of adaptation. There is no reason to reproduce the entire set of network techniques within a monolith if a simple direct call has already solved the target problem and does not increase thread coupling. Only those elements are meaningful that change the quality of the internal architecture: indirect interaction, a queue as a coordination boundary, registration instead of a direct reference, a trace instead of a broken stack, and backpressure as a controlled reaction to overload.

DISCUSSION

The proposed adaptation is valuable primarily because it creates an intermediate form of architectural maturity. Direct invocation is convenient in a small system where the lifecycle of dependencies is transparent, thread competition is limited, and the causal chain is easily read in a single stack. Early decomposition into microservices, by contrast, is justified when organizational and operational boundaries have already taken shape and when the cost of independent deployment is offset by the benefits, despite network complexity. Between these poles lies a broad class of C++ systems for which it is more productive to first master the internal discipline of event-based interaction and only then decide whether network decomposition is warranted. In such a case, the "Node Manager" functions as a full-fledged phase of architectural development.

The practical justification of a brokerized modular monolith appears in systems with several independent domain lines, active event exchange, high demands for reproducibility, and a tangible cost of concurrency errors. These include client platforms with dense business logic, trading and settlement systems, server applications with intensive inter-module exchange, game engines, and application platforms for device management. In all these cases, thread independence and the "Node Manager" relieve application code of the obligation to maintain thread context consistency. At the same time, the limit of usefulness is reached when the unified coordination core becomes a systemic bottleneck, channels become excessively numerous and dynamic, and modular boundaries begin to coincide with organizational and operational team boundaries. At that stage, it becomes more reasonable to discuss the next step in architectural evolution.

To compare architectural gains and new limitations, it is useful to bring the main patterns together in a single table (see Table 2).

Table 2. Microservice Patterns and Their In-Process Interpretation Through the "Node Manager"

Pattern	In-Process Interpretation	Architectural Gain	New Limitation
"Broker/mediator"	unified coordination contour in the "Node Manager."	reduced direct coupling between modules	increased dependence on the quality of the core
"Event Bus"	publication of messages into shared or thematic queues	Weakened synchronous coupling between sender and receiver	more difficult causal reconstruction without tracing
"Service Discovery"	recipient resolution through Registration and subscription	late binding and flexible replacement of handlers	need for strict registration discipline
"Distributed Tracing"	"Single Serialized Queue" or "Multi-Queue"	management of load profile and determinism	risk of incorrect channel separation and hidden inter-channel competition

"Queue Segregation"	Trace ID inside the message and routing log	restoration of the event propagation chain	increased service overhead in the message protocol
"Failure Handling"	separation of coordination failures and overload policies	early localization of infrastructural disruptions	A separate service model of errors is required

The table shows that in-process pattern adaptation does not literally duplicate the microservice architecture. It extracts managerial and constructive disciplines from it. The greatest benefit comes from moving away from direct object coupling toward contract-based exchange through channels. The greatest risk is that any decline in the engineering quality of the "Node Manager" immediately affects the entire system. From this follows a strict practical requirement: the coordination core needs to be designed as a separate infrastructural product with its own testing, profiling, and instrumentation procedures.

A separate circle of questions concerns observability. In a network system, services are required by default to log interactions because diagnostics would otherwise be nearly impossible. Within a single process, this obligation is often underestimated because developers rely on an ordinary debugger and the local call stack. A thread-independent architecture removes this illusion. As soon as a message crosses the boundary of a queue, the traditional local, step-by-step debugging model becomes incomplete. For that reason, an internal architecture based on the "Node Manager" requires the same level of tracing attention as a distributed platform.

This thesis can be developed more conveniently through a set of engineering measures that strengthen the architecture at the operational stage (see Table 3).

Table 3. Engineering Measures for Strengthening the "Node Manager" Architecture

Direction of Strengthening	Practical Measure	Target Effect
Tracing	mandatory Trace ID in every message; publication and processing log	restoration of the causal chain and faster diagnostics
Queue control	depth limits, overload counters, and a separate channel for service signals	early detection of backpressure and prevention of degradation
Registration rules	Completion of registration before entering the Execution phase; checking for duplicate subscriptions	routing stability and reduction of hidden conflicts
Message deduplication	command identifiers and repeated-processing policy	protection against repeated delivery of service and transactional messages
Type-based routing	strict correspondence between the message contract and the processing channel	reduction in erroneous deliveries and simpler flow auditing
Reproducibility	recording of message sequences and record-and-replay mode	reproducible integration testing
Core profiling	measurement of channel-draining time and handler occupancy share	identification of bottlenecks and incorrect queue separation

It becomes clear that a coordinated set of disciplines forms architectural resilience. Thread independence provides the foundation, queues define the geometry of load, registration supports late binding, tracing restores causality, and record-and-replay returns reproducibility to the system. If even one of these elements is removed, the architecture remains functional, but its operational maturity is noticeably reduced.

From an engineering perspective, the most productive property of this model remains the controlled redistribution of complexity. Synchronization, routing, and observability cease to be scattered responsibilities of multiple modules. They are concentrated into a single layer, making them easier to test, profile, and develop consistently. The cost of such concentration is evident: a unified coordination core requires a higher quality of implementation than an ordinary application module. For that reason, when introducing the "Node Manager," it is reasonable to treat it as the system's architectural center.

CONCLUSION

It has been established that, once the network boundary is removed, those microservice patterns that express the discipline of interaction retain architectural value. These include broker-mediated communication, event bus, service discovery, queue segregation, distributed tracing, and overload-control mechanisms. Their migration to a single process is justified when the monolith is already under pressure from inter-module coupling and thread-related complexity.

It has been shown that these patterns acquire a direct in-process form through the mechanisms introduced in the first article: "Node," "Node Manager," publish/subscribe, the "Registration" and "Execution" stages, the "Single Serialized

Queue" and "Multi-Queue" modes, as well as type-based routing built on "type erasure" and "std::function." As a result, microservice discipline is expressed through the structuring of the monolith itself.

The engineering implications for C++ architecture, debugging, tracing, and failure management have been identified. Thread independence and deterministic dispatch reduce the pressure on application code from concurrency, yet they simultaneously increase the requirements for observability, registration rules, queue control, and reproducibility. As a result, the earlier architectural paradigm continues in this article. From a solution to the problem of multithreaded interaction, it evolves into an instrument for adapting microservice patterns within monolithic C++ systems.

REFERENCES

1. Wang, Y., Kadiyala, H., & Rubin, J. (2021). Promises and challenges of microservices: An exploratory study. *Empirical Software Engineering, 26*(4). https://doi.org/10.1007/s10664-020-09910-y
2. Waseem, M., Liang, P., Ahmad, A., Shahin, M., Khan, A. A., & Márquez, G. (2022). Decision models for selecting patterns and strategies in microservices systems and their evaluation by practitioners. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '22)* (pp. 135–144). Association for Computing Machinery. https://doi.org/10.1145/3510457.3513079
3. Söylemez, M., Tekinerdogan, B., & Kolukısa Tarhan, A. (2022). Feature-driven characterization of microservice architectures: A survey of the state of the practice. *Applied Sciences, 12*(9), Article 4424. https://doi.org/10.3390/app12094424
4. Cortellessa, V., Di Pompeo, D., Eramo, R., & Tucci, M. (2022). A model-driven approach for continuous performance engineering in microservice-based systems. *Journal of Systems and Software, 183*, Article 111084. https://doi.org/10.1016/j.jss.2021.111084
5. Singjai, A., & Zdun, U. (2022). Conformance assessment of architectural design decisions on API endpoint designs derived from domain models. *Journal of Systems and Software, 193*, Article 111433. https://doi.org/10.1016/j.jss.2022.111433
6. Spenger, J., Carbone, P., & Haller, P. (2024). A survey of actor-like programming models for serverless computing. In *Lecture Notes in Computer Science* (Vol. 14360, pp. 123–146). Springer. https://doi.org/10.1007/978-3-031-51060-1_5
7. Ruiz, A., Aldea-Rivas, M., & Harbour, M. (2020). Non-blocking synchronization between real-time and non-real-time applications. *IEEE Access*. https://doi.org/10.1109/ACCESS.2020.3015385
8. Adas, D., & Friedman, R. (2020). Jiffy: A fast, memory efficient, wait-free multi-producers single-consumer queue. *arXiv*. https://arxiv.org/abs/2010.14189
9. Alaboudi, A., & LaToza, T. D. (2023). What constitutes debugging? An exploratory study of debugging episodes. *Empirical Software Engineering, 28*(5). https://doi.org/10.1007/s10664-023-10352-5
10. Rossetto, A. G. d. M., Noetzold, D., Silva, L. A., & Leithardt, V. R. Q. (2024). Enhancing monitoring performance: A microservices approach to monitoring with spyware techniques and prediction models. *Sensors, 24*(13), Article 4212. https://doi.org/10.3390/s24134212
11. Rüegger, J., Kropp, M., Graf, S., & Anslow, C. (2024). Fully automated DORA metrics measurement for continuous improvement. In *Proceedings of the International Conference on Software and Systems Processes (ICSSP '24)*. Association for Computing Machinery. https://doi.org/10.1145/3666015.3666020