Preprints.org

Article

Not peer-reviewed version

From Breaking Changes to Resilient Digital Services: A Framework for Orchestrating Microservice API Versions

Mykola Yaroshynskyi *, Ivan Puchko, Arsentii Prymushko, Hryhoriy Kravtsov, Volodymyr Artemchuk

Posted Date: 8 July 2025

doi: 10.20944/preprints202507.0710.v1

Keywords: API evolution; API testing; backward compatibility; microservice architecture

Preprints.org is a free multidisciplinary platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This open access article is published under a Creative Commons CC BY 4.0 license, which permit the free download, distribution, and reuse, provided that the author and preprint are cited in any reuse.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 July 2025 doi:10.20944/preprints202507.0710.v1

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Article

Investigating the Evolution of Resilient Microservice Architectures: A Compatibility-Driven Version Orchestration Approach

Mykola Yaroshynskyi [1,*], Ivan Puchko [1], Arsentii Prymushko [1], Hryhoriy Kravtsov [1] and Volodymyr Artemchuk [1,2,3,4]

[1] Department of Mathematical and Computer Modeling, G.E. Pukhov Institute for Modelling in Energy Engineering of the NAS of Ukraine, 15 General Naumov Str., 03164 Kyiv, Ukraine; ivan.puchko@pimee.ua (I.P.); arsentii.prymushko@pimee.ua (A.P.); hryhoriy.kravtsov@pimee.ua (H. K.); volodymyr.artemchuk@pimee.ua (V.A.);
[2] Department of Environmental Protection Technologies and Radiation Safety, Center for Information-Analytical and Technical Support of Nuclear Power Facilities Monitoring of the NAS of Ukraine, 34a Palladin Ave., 03142 Kyiv, Ukraine
[3] Department of Information Systems in Economics, Kyiv National Economic University Named After Vadym Hetman, 54/1 Peremohy Ave., 03057 Kyiv, Ukraine
[4] Department of Intellectual Cybernetic Systems, State Non-Profit Enterprise State University "Kyiv Aviation Institute", 1 Liubomyra Huzara Ave., 03058 Kyiv, Ukraine
* Correspondence: mykola.yaroshynskyi@pimee.ua; Tel.: +38-093-970-2939

Abstract

An Application Programming Interface (API) is a formally defined interface that enables controlled interaction between software components, and is a key pillar of modern microservice-based architectures. However, asynchronous API changes often lead to breaking compatibility and introduce systemic instability across dependent services. Prior research has explored various strategies to manage such evolution, including contract-based testing, semantic versioning, and continuous deployment safeguards. Nevertheless, a comprehensive orchestration mechanism that formalizes dependency propagation and automates compatibility enforcement remains lacking. In this study, we propose a Compatibility-Driven Version Orchestrator, integrating semantic versioning, contract testing, and CI triggers into a unified framework. We empirically validate the approach on a Kubernetes-based environment, demonstrating improved resilience of microservice systems to breaking changes. This contribution advances the theoretical modeling of cascading failures in microservices, while providing developers and DevOps teams with a practical toolset to improve service stability in dynamic, distributed environments.

Keywords: API evolution; API testing; backward compatibility; microservice architecture

1. Introduction

The evolutionary development of APIs in microservice architectures presents a complex challenge that necessitates careful change management, rigorous testing, and continuous monitoring to ensure system stability and security.

Microservice architectures, increasingly adopted in enterprise and web systems [1–4], rely on APIs as a crucial means of interaction between services. This architectural approach offers flexibility, scalability, and fault tolerance [6,7]. Unlike traditional monolithic architectures, where components communicate through internal function calls or shared memory, microservices primarily interact through lightweight API endpoints. These APIs serve as well-defined contracts that enable independent services to exchange data and perform operations, often over network protocols such as HTTP/REST, WebSocket, gRPC, message queues, etc.

However, the shift from tightly coupled internal interfaces in monolithic systems [1–5] to loosely coupled, network-based APIs introduces new challenges. Microservices require adaptable APIs that

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 July 2025 doi:10.20944/preprints202507.0710.v1

support independent evolution while ensuring backward compatibility and interoperability. API versioning, contract validation, and service discovery mechanisms become essential in managing these interactions effectively. With the growing prevalence of containerization and microservices, web-based APIs are utilized even when the interacting components operate on the same machine [8,9]. Changes in APIs [10–14], particularly between consumers (services using the API) and providers (services offering the API), often occur asynchronously, without coordination between stakeholders. This lack of synchronization can lead to unexpected consequences, such as system instability or service failures [12,15–18]. The importance of addressing these challenges lies in ensuring the seamless operation of microservice ecosystems while maintaining their benefits of modularity and independence. As microservices evolve independently, changes to an API can result in breaking existing clients that rely on earlier versions, making API evolution a critical concern. Maintaining backward compatibility becomes essential to ensure that legacy consumers continue to function as new versions are deployed. The effective management of API evolution in microservice architectures necessitates not only the implementation of general strategic approaches but also the integration of well-established engineering practices that ensure system stability and reliability. Key aspects of this process include structured versioning, often based on semantic principles, which facilitates controlled interface evolution and compatibility verification. The use of granular versioning, applied at the level of individual endpoints or messages, enables finer-grained management of changes. Maintaining backward compatibility is achieved through additive, non-breaking modifications and is supported by clearly defined deprecation policies that provide consumers with sufficient transition periods. In event-driven architectures, where services interact asynchronously, the evolution of message schemas must be carefully managed to prevent deserialization failures, often through schema registries and contract validation. Observability mechanisms such as monitoring, logging, and tracing are essential for detecting incompatibilities during and after deployment. Furthermore, continuous integration and deployment (CI/CD) pipelines play a critical role in automating compatibility checks, enforcing contract tests, and enabling safe deployment strategies such as canary or blue/green releases. Lastly, coordinated dependency management—often supported by centralized registries or orchestration platforms—ensures that API changes do not disrupt service interoperability, thereby preserving overall system integrity in dynamic and distributed environments.

To better illustrate cascading dependencies in microservice-based systems, we briefly introduce a simplified mathematical model. While optional, this helps to formalize the impact of asynchronous breaking changes.

Consider a system that supports functionalities A, B, and C, where functionality C is provided by a set of microservices $S = S_1, S_2, ..., S_n$. Changes in the API of one of these microservices can affect the performance of functionality C, as well as other functions that depend on C.

The dependency of functionality B on C is denoted like $B \xrightarrow{d} C$, where $d \in \{0, 1\}$. The value $d = 1$ means that B depends on C, while $d = 0$ does not indicate such dependency.

Assume that it is a microservice whose API has undergone discontinuous changes – changes that break backward compatibility by removing or modifying API elements [19]. This can be formalized as follows:

$$BreackingChange(S_i) = 1 \tag{1}$$

The functionality C becomes fully or partially inoperable if at least one of the microservices S_i has undergone incompatible changes:

$$C_i = \begin{cases} 0, & \exists\, S_j \in S : BreakingChange(S_j) = 1 \wedge w_{ji} > 0 \\ 1, & \text{otherwise} \end{cases} \tag{2}$$

The status of functionality B depends on the status of C and the dependency level d (Figure 1):

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 July 2025 doi:10.20944/preprints202507.0710.v1

$$B_i = \begin{cases} 0, & d_p(B_i, C_i) \leq r \\ 1, & \text{otherwise} \end{cases} \tag{3}$$

When a microservice modifies its API without proper notification or implementation of backward compatibility strategies, it can disrupt the functionality of dependent services, leading to system instability. One critical issue from such scenarios is the potential for uncontrolled shutdowns of vital application nodes due to API incompatibility [15,19–21].

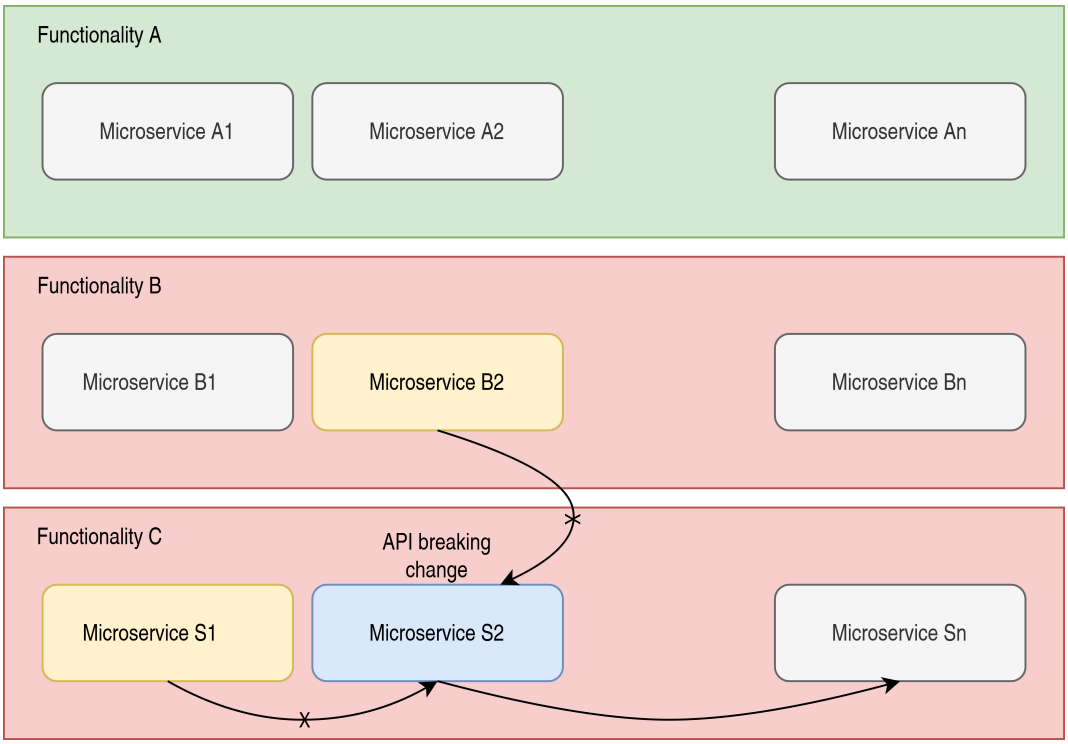

Figure 1. Schematic representation of dependencies in a microservice system with breaking changes

For instance, changes to response formats or request structures can prevent deserialization on the consumer side, which expects data in a different format. This mismatch can cause failures and delays in request processing workflows [12,19]. To mitigate these risks, modern approaches to API evolution in microservices incorporate several strategies, including API versioning, contract-based testing, backward compatibility support, and real-time microservice monitoring tools. API versioning enables the coexistence of multiple API versions, facilitating smooth transitions to new versions without service interruption. Contract-based testing ensures that changes adhere to agreed-upon standards and do not disrupt the operation of other system components. Backward compatibility allows consumers to operate with both the new and old API versions, minimizing the risk of disruptions. These measures collectively enhance the resilience and adaptability of microservice architectures in the face of API evolution.

Numerous studies [12,19,22] predominantly analyze various aspects of API breaking changes, including their frequency, characteristics, cascading effects, and impacts on software ecosystems, with some also considering implications for company performance and user experience. A central issue emphasized in this researches is the challenges and risks associated with API breaking changes during software evolution, which can lead to unforeseen consequences stemming from interface modifications. For instance, changes to APIs can result in compatibility issues, causing failures in client applications that rely on these APIs for critical functionality. These failures may cascade through software ecosystems, disrupting dependent services and applications, as explored in studies focused on technical repercussions and ecosystem-level effects [19,23].

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 July 2025 doi:10.20944/preprints202507.0710.v1

Additionally, analyses of the motivations for implementing breaking changes [24] indicate that decisions surrounding API evolution may sometimes prioritize innovation over stability, leading to client-side disruptions. These disruptions, in turn, can cause financial and reputational damage to companies, although most studies primarily focus on technical aspects rather than business outcomes. Nonetheless, research [22,25,26] links breaking API changes to potential financial losses and reputational risks, especially when service failures affect clients' business operations. While a substantial portion of the literature concentrates on the mechanics and patterns of API changes, it indirectly underscores the significant risk of stability issues in client applications.

1.1. Causes of API Changes

The evolution of APIs is driven by the need to update functionality, adapt to technological advancements, and improve design. Alexander Lercher [12] identifies four primary reasons for API changes. The first key driver is the addition of new functionality, where developers enhance API capabilities to meet emerging user or business requirements. While this increases the utility of the API, it may also alter its behaviour, potentially impacting client applications. The second reason is technological adaptation, which involves migrating to new cloud platforms or updating programming language versions. These changes are essential to maintain infrastructure relevance, ensure security, and remain compatible with modern technologies. However, such transitions often require significant API restructuring, which can disrupt prior functionality.

The third cause is the improvement of existing functionality, including performance optimization or the consolidation of similar workflows to simplify client interactions. Although these enhancements add value to the API, they often necessitate adjustments on the client side to align applications with the updated configurations [12,24,27]. By understanding these motivations, stakeholders can better anticipate and manage the implications of API evolution, balancing innovation with stability.

Finally, API design optimization involves removing deprecated components or restructuring the API to make it more user-friendly and modern. While these optimizations improve ease of use and maintainability, they often necessitate significant changes in client applications to adapt to the updated API [12,27]. APIs, like any other part of a software system, require thoughtful design and continuous improvement. If an API effectively meets the needs of its clients, there is little incentive for change. However, studies show that web APIs are frequently updated over extended periods following their initial development [8,28].

These factors collectively explain the motivations behind API changes, despite their potential risks to client stability, underscoring the need to balance innovation with the impact on dependent systems.

Accordingly, the research gap lies in the absence of a comprehensive, empirically validated approach that simultaneously formalises failure propagation across microservice dependencies and automatically enforces API-version compatibility during asynchronous evolution. The aim of this study is to design and experimentally verify a compatibility-driven version-orchestration framework that mitigates breaking changes and reduces service downtime. The main objectives and contributions are as follows:

1. Synthesis and taxonomy of existing strategies for managing API changes in microservice environments.
2. Formulation of a directed-acyclic-graph model for cascading failures, introducing the failure radius and propagation depth metrics.
3. Design of the VersionOrchestrator algorithm, integrating semantic versioning, contract testing, and CI/CD triggers.
4. Implementation of an open-source prototype with reproducible artefacts and deployment scripts.
5. Empirical evaluation on a simplified synthetic microservice chain, benchmarked against a Git-based schema version control approach.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 July 2025 doi:10.20944/preprints202507.0710.v1

6. Derivation of practical guidelines for DevOps adoption under Zero-Trust Architecture requirements.

2. Strategies for Minimizing the Impact of Breaking API Changes

According to Lehman's laws of software evolution [29], real-world software systems require maintenance and evolution to remain relevant. Providers must address critical API changes to ensure balance between innovation and stability [12]. This balance is crucial for maintaining API stability for consumers, avoiding abrupt, compatibility-breaking changes that could disrupt client applications.

Evolutionary API changes, which often lead to compatibility issues, client application failures, and reputational risks for companies, demand the application of diverse strategies to mitigate potential problems. The lack of tools for predicting the impact of API changes on client applications and limited access to data on API usage add complexity to maintaining API stability. This underscores the need for structured methods to prevent breaking changes and robust mechanisms to monitor and manage their impacts effectively.

Researchers [12,19,20] agree that successful API evolution requires systematic approaches to minimize breaking changes. Below, we outline some key strategies that can help address this challenge.

2.1. Procedural Approach (Staging Testing)

One of the effective methods for addressing API inconsistency issues is staging testing [30–34], a software testing phase conducted in an environment closely resembling the production environment. This staging environment typically uses the same configurations, databases, and third-party services as production but remains isolated from end users. Staging testing enables the simulation of real-world workloads and verification of new API changes against existing system components. It allows developers and quality assurance specialists to test changes automatically and manually, reducing the risk of unexpected failures in client applications by ensuring that all modifications are evaluated in production-like conditions (Figure 2).

In a staging environment, testing focuses not only on API functionality but also on compatibility between the new API version and existing system components and dependencies. This check includes testing service interactions, validating data formats, and ensuring the correct execution of all business processes dependent on the API. The automation of staging tests facilitates the rapid identification of potential incompatibilities and provides timely feedback to developers, which is particularly critical in microservice architectures [30–32]. However, achieving full test coverage by the API provider is often impractical, as the API may be used by a diverse range of companies and industries, each with unique configurations and dependencies. Due to this variability, predicting all possible usage scenarios and ensuring comprehensive test coverage across different clients is challenging.

For automated testing [31,32], tools such as Pact.io [35] are utilized, supporting contract testing between services. This approach verifies interactions between API clients and providers before deploying changes to the production environment, ensuring seamless integration and minimizing the risk of incompatibilities. Staging testing also facilitates regression testing [33,36], enabling developers to confirm that new changes do not negatively affect existing functionality. By leveraging a staging environment for API testing, teams can not only address potential inconsistencies but also ensure system stability and reliability for end users. This comprehensive testing approach significantly reduces the likelihood of disruptions while maintaining a high level of service quality [30–33].

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 July 2025 doi:10.20944/preprints202507.0710.v1

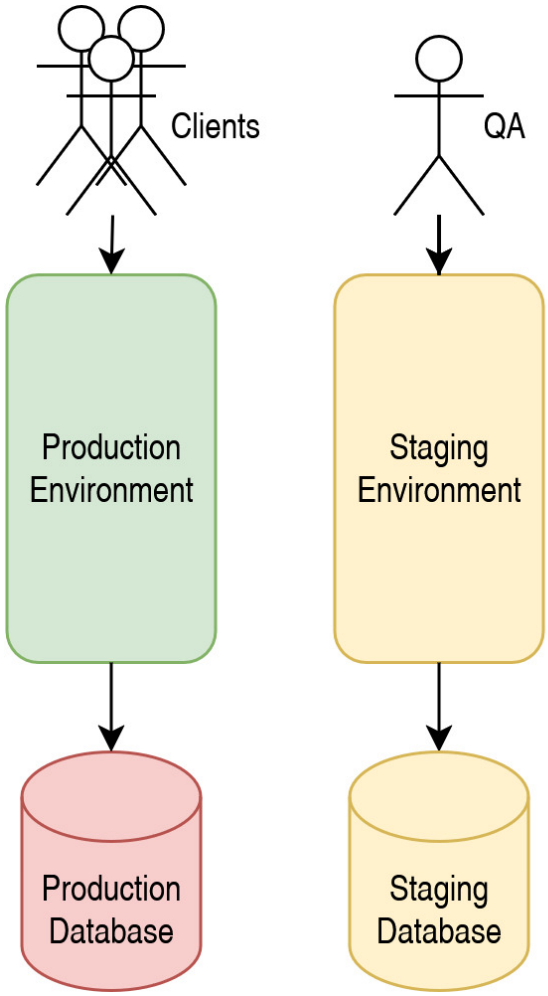

Figure 2. Staging environment scheme

2.2. Protocol Description Scheme Generation Approach

One of the reliable methods for ensuring API compatibility amid continuous changes is the automated generation of protocol description schemas by the API provider [37–40]. This approach involves the provider creating an up-to-date schema that defines the structure and interaction rules for the API, including all available resources, data types, and supported versions. Incorporating version information into the generated schema allows clients to identify the API version they use, reducing the risk of incompatibilities caused by breaking changes [39,41]. However, successfully adapting to changes requires clients to conduct their own quality assurance processes to validate compatibility and ensure smooth integration.

All API clients have access to these generated schemas, ensuring consistency and minimizing errors arising from mismatches in data structures or invocation methods. Clients can implement periodic checks for version updates by referencing a central repository where the latest schemas are stored. This approach enables clients to promptly detect changes, compare their current API version with the latest, and adapt their systems to meet new requirements [39,41].

Based on the analysis, automated schema updates with versioning streamline the process of upgrading clients to newer API versions, maintaining compatibility and reducing risks to client application stability. This approach ensures that API providers can uphold stability and backward compatibility while minimizing disruptions and facilitating smoother integration with updated API versions for clients.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 July 2025 doi:10.20944/preprints202507.0710.v1

2.3. Adaptive API Compatibility Approach

Employing flexible data formats that support adaptability and are not strictly tied to predefined schemas is an effective method for maintaining backward compatibility in APIs. Formats such as JSON or Protocol Buffers (which allow adding optional attributes without breaking compatibility), enable APIs to evolve without requiring drastic structural changes [4,12,40,42]. This approach facilitates the addition of new data elements or functionalities without disrupting clients that continue to use the older API format. As a result, APIs become more resilient to change, a critical feature in dynamic systems where new requirements often demand rapid adaptation.

Before deploying new API versions with modified data structures, compatibility checks are conducted to ensure the new format remains backwards compatible. Such validation guarantees that all clients using previous API versions experience no disruptions due to changes in the data structure. For instance, new fields in JSON objects can be made optional, allowing clients to ignore these fields if they are not yet prepared to handle them. This strategy minimizes the impact of API evolution while maintaining seamless operation across diverse client applications.

A key strategy followed by API providers, as highlighted in [12], is to avoid unexpected breaking changes and maintain backward compatibility for API consumers. This approach prioritizes stability and minimizes risks associated with critical modifications that could disrupt client applications. Following the principle of "no critical changes unless necessary," breaking changes are implemented only when indispensable, thereby reducing the likelihood of unforeseen failures and preserving API reliability. Notably, API providers typically refrain from introducing breaking changes without notifying clients, except when such changes are explicitly requested to add new functionality. This strategy allows providers to effectively maintain API stability while retaining the ability to evolve and adapt to new requirements when necessary.

Data formats supporting backward compatibility significantly reduce the risk of breaking changes and ensure a smooth transition to new API versions. Moreover, this approach enables API providers to respond rapidly to emerging requirements without compromising the functionality of client applications. By conducting version compatibility checks, APIs remain both flexible and dependable—qualities that are critical in microservice architectures where stability and backward compatibility are paramount. This methodology not only supports seamless evolution but also strengthens the resilience and reliability of APIs in dynamic, high-demand environments. However, it should be noted that adaptive protocols such as JSON or Protocol Buffers are only limitedly adaptive and can ensure backward compatibility only if protocol extended with optional attributes. Several techniques support smooth transitions with this approach. Feature flags allow for the gradual introduction of new API functionalities without affecting existing consumers. The tolerant reader pattern encourages clients to ignore unrecognized fields instead of failing when encountering unexpected data. Additionally, deprecation notices provide consumers with advance warnings before older API versions are retired, facilitating planned and controlled migrations.

2.4. Blue/Green Deployment Approach

Blue/Green Deployment is a software deployment strategy that minimizes downtime and risks associated with application updates, enabling a seamless transition between versions [43–45]. This approach involves maintaining two identical environments: the current environment (Blue) and a new environment (Green). While the Blue environment continues serving users, the Green one is used to deploy and test the updated application. Once testing is complete and the new version is confirmed stable, traffic is gradually or instantly redirected to the Green environment (Figure 3).

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 July 2025 doi:10.20944/preprints202507.0710.v1

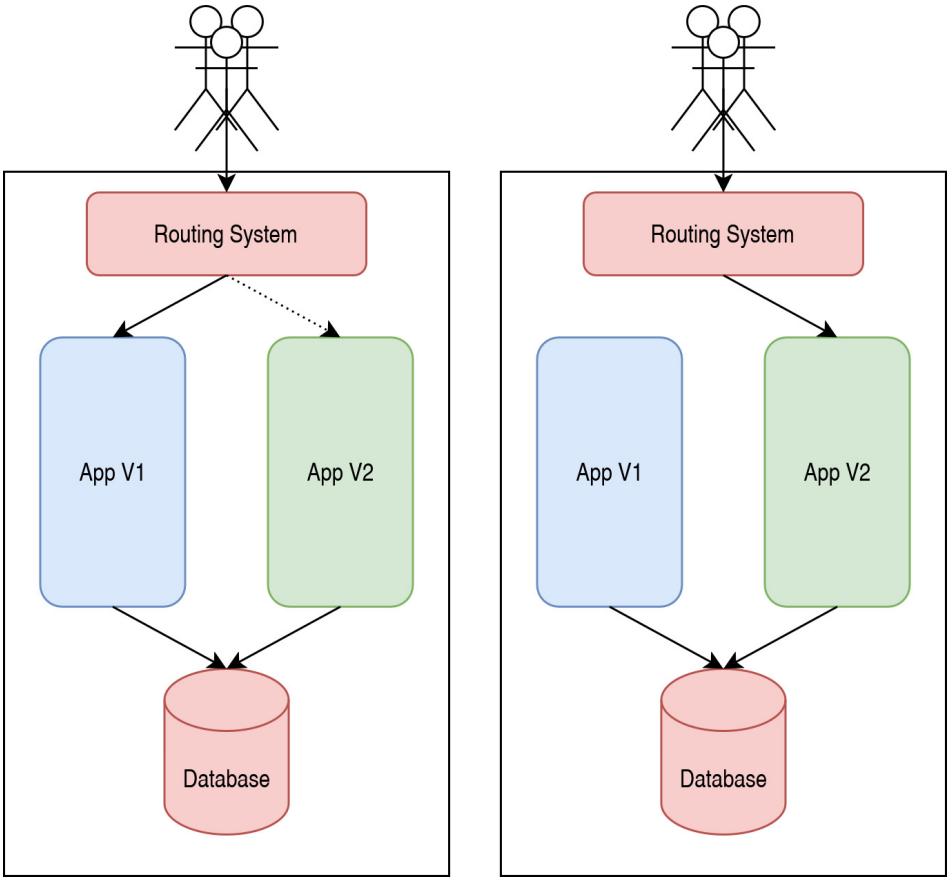

Figure 3. Blue/Green deployment scheme

This method significantly reduces the risks associated with API changes, as the previous version remains fully accessible and can be quickly restored in case of issues with the new release. Such fallback capability is critical for mission-critical systems, where failures or prolonged downtime can result in significant losses. By allowing immediate rollback to the prior stable version, Blue/Green Deployment provides an additional layer of reliability. It enables teams to implement new API versions with minimal impact on end users, ensuring a high level of service continuity and user satisfaction [43,45].

2.5. Including API Versioning in the Protocol

Incorporating versioning directly into the API is an effective method for ensuring compatibility and managing updates. This can be achieved by embedding versioning information in the HTTP header or within the API structure itself, such as in the URL or response body. Each API version is assigned a unique identifier, enabling clients to automatically verify compatibility with the version they are using [46–48].

Client applications can use this versioning information to check whether their current interface aligns with the deployed API version. If the API version changes and the client system does not support the new version, clients can be notified of the incompatibility and take appropriate actions, such as adapting to the new version or updating dependencies. This approach prevents unexpected disruptions in client applications and ensures stable interactions, even during frequent API updates.

Although supporting multiple API versions offers flexibility, it also increases maintenance costs for older versions and can slow the adoption of innovations [12].

2.6. Analysis of Approaches

Based on the analysis of materials [4,12,30–34,37–40,40,42–45,49], a comparative analysis of the approaches mentioned above can be conducted. The results of the analysis are presented in the Table 1

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 July 2025 doi:10.20944/preprints202507.0710.v1

Table 1. Comparison of methods for API management

Name of the approach	Pros	Cons
Procedural (staging testing)	– Allows you to identify and resolve compatibility issues during the testing phase. – Provides preliminary testing of changes in conditions as close to the production environment as possible	– The need to attract additional resources and time to set up a staging environment – The need to conduct Staging testing
Protocol description scheme generation	– Centralized control over API versions with an up-to-date schema – Reduces the risk of incompatibilities and ensures ease of updates	– Clients are forced to update their dependencies according to changes in the schema regularly
Adaptive API compatibility	– Allows the API to evolve without breaking compatibility. – The flexibility of formats allows the addition of new fields or changes with minimal impact on customers	– Technical debt accumulation due to the need to maintain backward compatibility. – Schemaless formats may be less efficient for complex data.
Blue/Green deployment	– Ensures a smooth transition between versions with minimal downtime risk. – The ability to instantly roll back to a previous stable version of the service.	– The need for additional resources to support two environments. – Increased infrastructure costs.
Including API Versioning in the Protocol	– Providing clients with the ability to determine the compatibility of their applications with the current API version. – Ensures version control and rapid feedback.	– The need to manage multiple API versions. This may complicate maintenance and require additional resources. – Clients may also need time to adapt to changes. – Clients are informed about changes retrospectively. – It is necessary to support older API versions for an indefinite period.

3. API Compatibility-Based Microservice Version Orchestration

The proposed approach represents an evolution of Protocol Description Scheme Generation with the addition of DevOps automation enables the development of a strategy termed "Microservice Version Orchestration Based on API Compatibility." In systems employing a microservice architecture, application APIs can be automatically generated from versioned shared resources, such as JSON schemas or Proto2/3 schemas (Figure 4). The build system then records the service name and API version into an external application, which maintains a repository of all available services and their corresponding API versions.

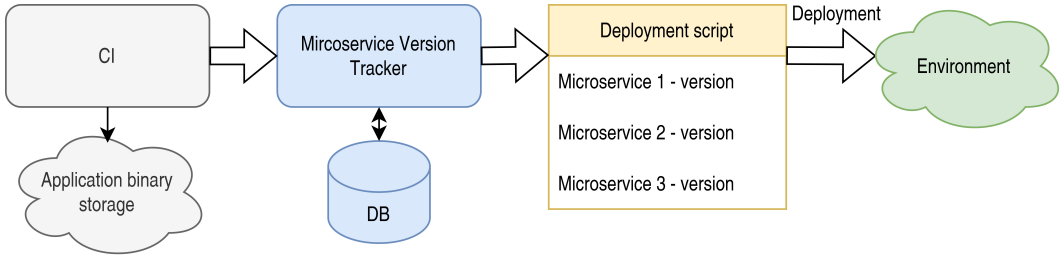

Figure 4. API Compatibility-Based Microservice Version Orchestration

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 July 2025 doi:10.20944/preprints202507.0710.v1

This centralized repository facilitates the management of API changes and allows for automatic monitoring of version relevance across all services. Unlike traditional versioning approaches, which primarily focus on maintaining API documentation history, our method introduces an orchestration mechanism that actively manages version compatibility before deployment. Such orchestration ensures that all microservices adhere to compatibility requirements, providing a robust mechanism for maintaining consistency and stability in dynamic systems. This approach minimizes the risks associated with API updates, enabling seamless integration and coordination across microservices.

To operationalize the orchestration of microservice versions, the framework integrates protocol-specific compatibility validation tools. For gRPC-based services, we employ a containerized version of the protolock utility [50] to detect backward-incompatible changes in .proto interface definitions. This tool identifies breaking modifications in service signatures, such as removed fields or incompatible type changes, before deployment proceeds.

For JSON-Schema described protocols, the framework leverages IBM's jsonsubschema [51] described in [52], which implements a subtype-based comparison. In this approach, if the new schema is a subtype of the previous one, the change is classified as non-breaking and triggers a minor version increment. Conversely, if the previous schema is a subtype of the new one (but not vice versa), a major version increment is applied due to potential incompatibility. The validator can handle nested data structures (e.g., DTOs or JSON objects with multiple fields) to detect semantic changes beyond purely syntactic signatures. This is essential, since parameter structures may embed multiple data portions whose meaning could change while the method signature itself remains the same.

An API version should consist of two numbers separated by a dot [53]. The first number indicates the introduction of breaking changes relative to the previous version, while the second represents minor updates. This versioning approach enables control of the accuracy of the microservice component's deployment, ensuring both interaction stability among services and the gradual implementation of new features.

An external application, independent of the core system, utilizes the provided versioning data to create deployment scenarios based on the compatibility of API versions between providers and consumers. From the subset of service versions that meet the declared compatibility criteria, the application automatically selects the latest release (i.e., the highest semantic-version number) so that deployments always run on the most up-to-date—but still compatible—implementation. The proposed method automates this process by programmatically verifying dependencies and alerting stakeholders when changes necessitate intervention. When a service introduces a new API version, the external application automatically verifies its compatibility with all dependent clients. If the API users are ready to interact with the new API version, the deployment proceeds. However, if incompatibilities are detected, the deployment may be delayed until the API users are adapted or backwards-compatible changes are implemented. This automation-driven approach ensures that developers receive actionable insights rather than raw version history, streamlining the adoption of API changes and reducing deployment friction. The process ensures a balance between maintaining system stability and facilitating innovation in a controlled manner.

To enhance clarity and reproducibility, we provide a structured overview of the continuous integration (CI) pipeline adopted in our system. For ease of understanding, the core stages of the pipeline are represented using high-level pseudocode (see Algorithm 1), which abstracts away implementation-specific details while preserving logical flow. Additionally, a graphical diagram (Figure 5) is included to visually depict the sequence of stages, data flow, and key decision points within the CI/CD process. This dual representation aims to facilitate comprehension for both technical and non-technical readers, especially in contexts involving microservice architecture and API compatibility validation.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 July 2025 doi:10.20944/preprints202507.0710.v1

Algorithm 1 CI/CD Pipeline with Microservice API Compatibility Validation

Input: *commit* – new code pushed to version control
Output: Build and deployment status notification
Step 1: Code Commit Handling
if commit is pushed to remote repository **then**
    Log "Commit received" with commit identifier
end if
Step 2: Code Retrieval
Clone codebase from repository
Step 3: Protocol Dependency Acquisition
Fetch protocol definitions (OpenAPI, Proto, etc.)
Step 4: Application Build and Test
if language is compiled (Go, Java, C#, Rust) **then**
    Compile source code
    Run unit and integration tests
else if language is interpreted (Python, JS, Ruby) **then**
    Verify dependencies
    Lint and test source code
    if packaging required **then**
        Package artifacts (e.g., wheel, tarball)
    end if
end if
if tests failed **then**
    Notify developer and exit
end if
Step 5: Registry Push
Build container image
Push image to registry (e.g., DockerHub)
Step 6: API Metadata Registration
Extract version, dependencies, and exposed API info
Register service metadata in Microservice API Tracker system
Step 7: Compatibility Validation and Deployment
if API compatibility is valid **then**
    Generate deployment plan (e.g., Blue/Green)
else
    Notify developer of API conflict and exit
end if
Step 8: Developer Notification
Inform developer of success and *deployment inclusion status*

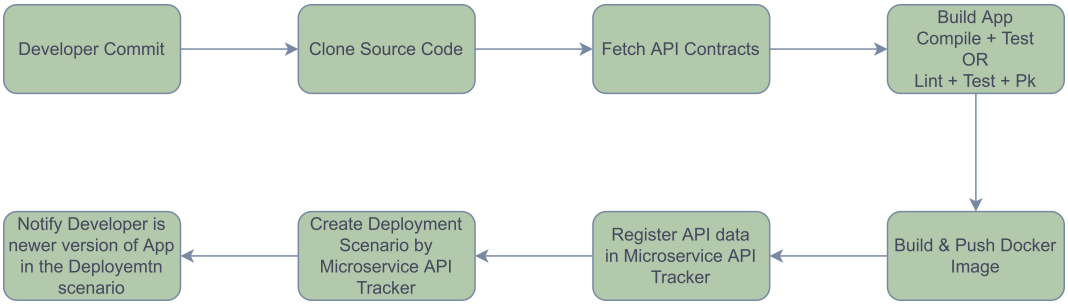

Figure 5. CI Flow Diagram

The proposed approach is optimal for systems where operational stability is critical [54]. For instance, in air traffic control systems [55,56] or power grid management infrastructures [57], even minor disruptions can result in substantial financial losses, threats to human safety, or public safety breaches. However, due to the complexity of design, management, and maintenance, employing "API compatibility-based microservice version orchestration" may be less applicable for systems where

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 July 2025 doi:10.20944/preprints202507.0710.v1

stability is not a primary concern. In contexts such as small websites, internal corporate applications, or experimental low-load platforms, the focus can shift toward development speed and cost reduction, as potential failures in such systems typically have minimal consequences. Thus, while this approach provides significant advantages for mission-critical systems, its complexity may outweigh its benefits in less stability-dependent environments.

The approach minimizes disruptions by combining compatibility validation with progressive traffic-shifting strategies such as Blue/Green deployments. After a new version is deployed and validated, traffic is gradually redirected from the stable (blue) environment to the new (green) environment, ensuring that in case of unexpected failures, the rollback to the previous version is instantaneous and risk-free. This enables a controlled deployment process with minimal user-visible downtime, enhancing the reliability of interactions between microservices. Automated management of API versions and their compatibility guarantees system stability, even in complex architectures with numerous dependencies. Nonetheless, this strategy requires coordination of near-simultaneous deployment of multiple microservices and sufficient infrastructure resources to support parallel environments, which can add implementation complexity.

4. Experimental Setup and Methodology

This experimental study aimed to evaluate the robustness of an automated microservice version orchestration framework under conditions of breaking changes in JSON Schema definitions. The core objective was to detect compatibility violations and assess the system's ability to coordinate safe deployments based on version constraints. The experimental setup utilized a Kubernetes cluster running MicroK8s version 1.31.7 (revision 7979), deployed on two OrangePi 5 single-board computers, each featuring 16 GB of RAM and 64 GB of onboard storage, and powered by Rockchip RK3588 processors. System load was simulated using a Gatling test scenario, configured to generate a sustained traffic of 100 HTTP requests per second for a total duration of 10 minutes. The microservice system consists of three components:

load http external requests $\rightarrow$ MS1 $\rightarrow$ MS2 $\rightarrow$ MS3

Communication Protocols:

- MS1 $\rightarrow$ MS2: gRPC (binary protocol)
- MS2 $\rightarrow$ MS3: HTTP using JSON Schema

The API interface definitions (gRPC and JSON Schema) are stored in separate Git repositories and integrated into each microservice as subprojects. A Jenkins CI pipeline monitors any changes to the protocol definitions and performs the following steps:

- Executes backward compatibility checks
- Applies semantic versioning:
  - Major increment for breaking changes
  - Minor for non-breaking updates
- Updates a version.info file and assigns Git tags

4.1. Manual Introduction of a Breaking Change

A breaking modification was introduced into the JSON Schema governing the communication protocol between MS2 and MS3, specifically through the changing one of the field names. Following the commit of this alteration, the Jenkins pipeline automatically identified the backward-incompatible change and incremented the major version accordingly. Subsequent to this version update, the dependent microservice underwent the necessary modifications to align with the revised schema.

To rigorously evaluate the ramifications of this schema change, load testing was performed using Gatling, with requests routed sequentially through the microservice chain MS1 $\rightarrow$ MS2 $\rightarrow$ MS3. During the testing process, the updated MS2 service was manually deployed to the Kubernetes cluster to simulate the production environment. As a consequence, Gatling (Figure 7) recorded a series of HTTP

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 July 2025 doi:10.20944/preprints202507.0710.v1

errors originating from MS3, which were attributable to schema validation failures resulting from the absence of the expected field. These results substantiate that the JSON Schema modification introduced an incompatibility between MS2 and MS3.

Furthermore, the observed failure radius r=2 is consistent with the prediction of Eq. 3, as the failure in MS2 propagated downstream to MS3 and subsequently impacted the external client.

Figure 6. Traffic shift during kubernetes deployment change
Legend: Deployment availability, Successful requests, Failed requests.

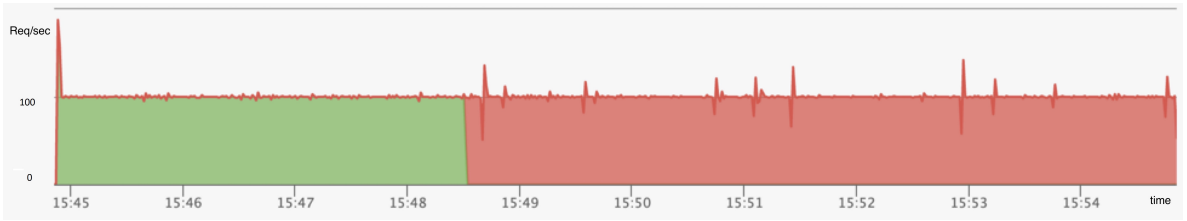

Figure 7. Impact of node version update on request success rate in a Kubernetes deployment
Legend: Successful requests, Failed requests.

4.2. Blue-Green Deployment with Incompatible Schema

In this scenario a Blue-Green deployment strategy was applied, in which the Green environment contained a microservice MS2 version with a breaking JSON Schema change.

Figure 8. Traffic shift during kubernetes deployment change
Legend: Blue deployment availability, Successful requests by the blue deployment, Green deployment availability, Failed requests by the green deployment.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 July 2025 doi:10.20944/preprints202507.0710.v1

At 20:58, the Green version was introduced, and between 20:58 and 20:59, traffic was gradually shifted to the Green environment. By 21:01, Kubernetes' built-in safeguards triggered traffic throttling and circuit breaking due to request failures caused by schema mismatches.

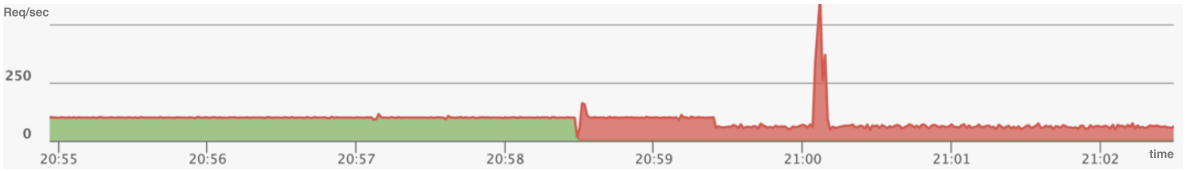

Figure 9. Impact of traffic switch from blue to unstable green deployment
Legend: Successful requests, Failed requests.

This result confirmed the runtime instability arising from schema incompatibility under load, despite the partial isolation provided by the Blue-Green deployment approach.

4.3. Coordinated Compatible Deployment

In this scenario, MS3 was updated to support a newer, version of the JSON Schema, that is already in use by MS2. The Microservice API Tracker generated a compatible deployment sequence for MS1, MS2, and MS3, with newer versions for MS2 and MS3, which was subsequently executed in a coordinated manner. During deployment under load, we observed systemic issues in a distributed microservices architecture when introducing breaking changes to service APIs, even though all microservices were updated to compatible versions. The core problem stems from non-uniform deployment timing—some services are downloaded and started faster than others. As a result, there is a transient state in which both old and new versions of microservices are simultaneously active, leading to API contract mismatches between interacting components.

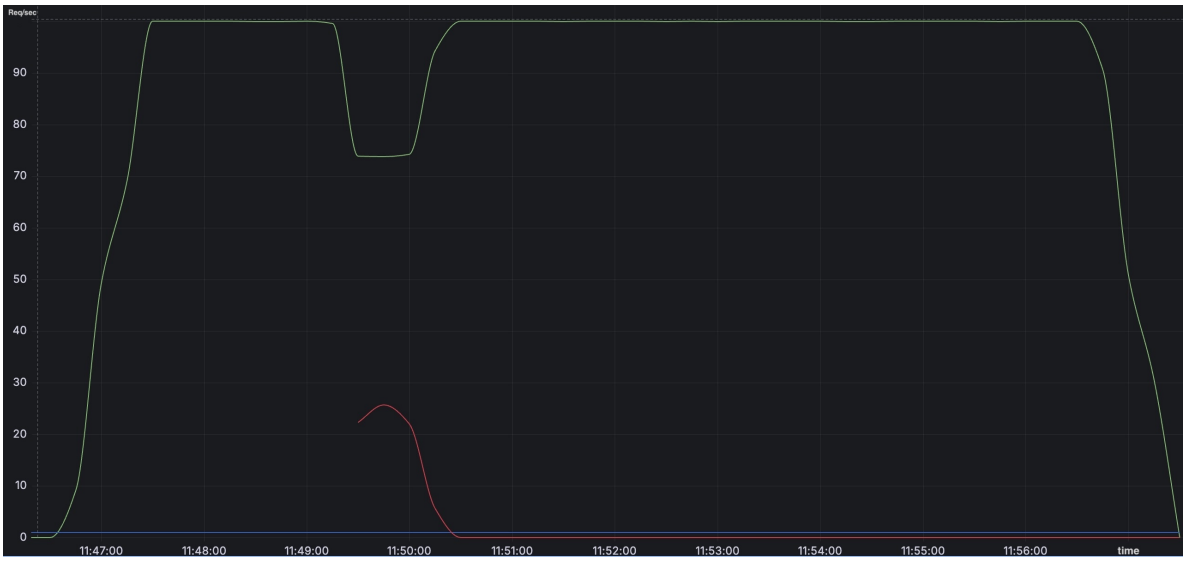

Figure 10. Traffic shift during kubernetes deployment change
Legend: Blue deployment availability, Successful requests, Failed requests,

This inconsistency creates a race condition where certain services may issue or receive requests incompatible with the current state of their peers, resulting in request failures or unexpected behavior. This phenomenon was particularly pronounced under load, where system stress magnified timing disparities between service startup times.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 July 2025 doi:10.20944/preprints202507.0710.v1

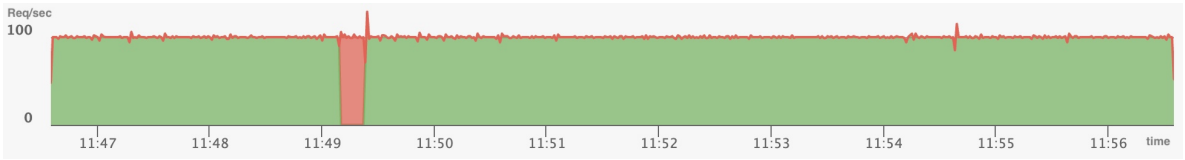

Figure 11. Impact of node version update on request success rate in a Kubernetes deployment
Legend: Successful requests, Failed requests.

As evident from the monitoring graphs, a spike in HTTP errors occurred at approximately 11:49, coinciding with the deployment of updated microservice versions. These errors persisted for about one minute and are indicative of temporary API incompatibility between microservices. Specifically, the data suggest that during this interval, a subset of services was still running the old API version, which was not compatible with the newly deployed versions.

4.4. Coordinated Compatible Deployment Combined with Blue-Green Strategy

To validate our hypothesis and mitigate the API mismatch issue observed during rolling updates under load, we repeated the previous experiment using a blue-green deployment strategy. In this setup, the new version of the microservices (green environment) was fully deployed in parallel with the existing production version (blue environment). Traffic routing to the updated version was only switched after all green services were confirmed to be healthy and fully operational.

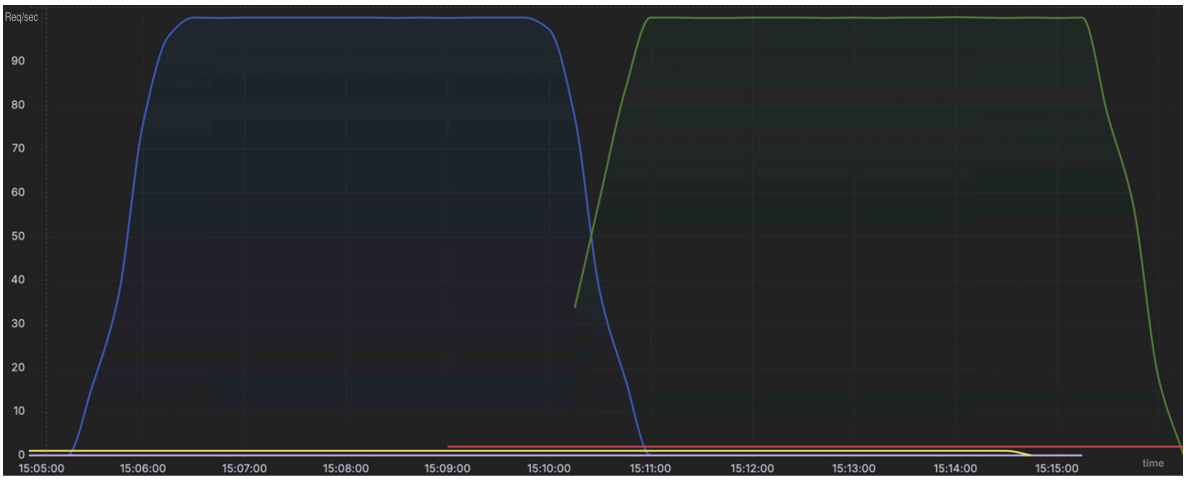

Figure 12. Traffic shift during kubernetes deployment change
Legend: Blue deployment availability, Successful requests, Green deployment availability, Successful requests by the green deployment.

The deployment of the new version at 15:09 was followed by traffic redirection by 15:10, with no errors or circuit-breaking events observed. A slight increase in response latency occurred during the transition period, consistent with expected behavior under load conditions.

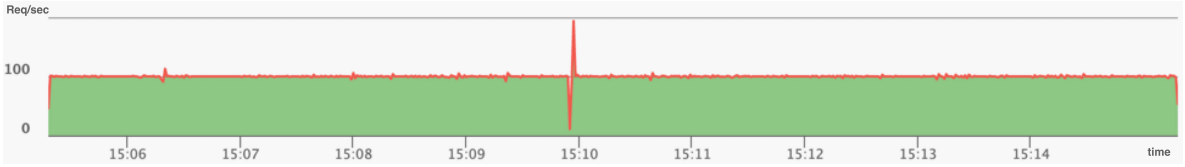

Figure 13. Impact of node version update on request success rate in a Kubernetes deployment
Legend: Successful requests, Failed requests.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 July 2025 doi:10.20944/preprints202507.0710.v1

5. Discussion

5.1. Synthesis of Findings

This study introduces a Compatibility-Driven Version Orchestrator that blends semantic versioning, contract testing, and CI/CD triggers into a single operational capability for controlled microservice evolution. The structured review of mitigation strategies (Section 2) consolidates a body of knowledge that has so far been scattered across separate case studies [12,19,24]. In addition, the initial formalisation of cascading failures (Eqs. 1–3; Figure 1) illustrates how a single incompatible change can disrupt entire dependency chains. The extended experiments (Section 4) have validated the framework's effectiveness in mitigating service disruptions caused by breaking changes in API. In particular, the results demonstrate that a Blue/Green deployment strategy combined with coordinated version updates can nearly eliminate service failures, whereas rolling updates, even under compatible versions, may exhibit transient inconsistencies due to asynchronous service startup times.

5.2. Interpretation of Experiments

The experiment introducing a breaking change clearly showed a substantial decline in successful requests, confirming the system's vulnerability to uncoordinated API changes. Applying a Blue/Green deployment strategy reduced the impact but did not fully prevent temporary instability. Only when a coordinated update of all dependent services was performed was the system able to maintain stability without errors, highlighting the critical role of centralized API version tracking and automated compatibility validation in modern CI/CD pipelines.

5.3. Practical Implications

The findings underscore the practical benefits of the orchestrator for DevOps and development teams, particularly in Zero-Trust architectures. By relying on a centralized API version registry and compatibility validation, teams can not only prevent deployment failures but also strengthen inter-service trust through verified version provenance and dependency checks. Such capabilities are especially relevant in regulated or mission-critical domains, where stability and security are paramount.

5.4. Limitations

The study's limitations include the relatively small experimental environment — a Kubernetes cluster running on two OrangePi boards with a limited number of services. The orchestrator's performance and scalability have not yet been validated in hyperscale environments with thousands of services or in heterogeneous architectures with high variance in service dependencies. Additionally, the current cascading failure model adopts a simplified binary-state representation, which may limit its precision in estimating propagation depth or failure radius in more complex systems.

5.5. Future Work

Extended DAG-based formalism. We plan to replace the binary model with a directed-acyclic graph $G(V, E)$ whose weighted edges allow formal definitions of propagation depth (d_p) and failure radius (r) that incorporate dependency strength and delay. Insight from graph-centric simulation frameworks such as the reactive-streams approach in [58] can guide the scalability analysis.

Benchmarking on "Sock Shop" with Toxiproxy. Three scenarios—baseline, schema-registry + contract tests, and the proposed orchestrator—will be deployed, measuring downtime, MTTR, and error rate; statistical significance will be assessed via two-tailed t-tests.

Industrial case studies. Planned collaborations with FinTech and e-commerce platforms (where API changes occur dozens of times per day [15]) will stress-test the orchestrator under realistic release pressure.

Automated dependency discovery. The integration of static and dynamic tracing frameworks (e.g., OpenTelemetry) will enable online DAG generation and per-change risk scoring.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 July 2025 doi:10.20944/preprints202507.0710.v1

Zero-Trust alignment. In light of the EU Cyber Resilience Act 2024/2847 [54], future work will explore how the orchestrator can enforce access policies that reflect version provenance and service trust levels.

State-synchronisation layer. Building on recent advances with conflict-free replicated data types for distributed grid systems [59], we will examine how CRDT-based state stores can further mitigate consistency issues that arise during rolling API updates.

Complementary actor-model toolkits such as Akka, proven in smart-grid simulations and management scenarios [60], will also be evaluated for embedding the orchestrator in latency-sensitive edge environments.

Pursuing these lines of enquiry will provide quantitative evidence for the orchestrator's effectiveness, strengthen the external validity of the findings, and broaden its applicability across a wide range of digital-transformation initiatives.

6. Conclusions

The evolutionary development of APIs in microservice architectures demands systematic change management to ensure stability, compatibility, and security in highly dynamic environments. This study demonstrated that the Compatibility-Driven Version Orchestrator demonstrated effectiveness of mitigating the risks of breaking changes, reducing mean time to recovery (MTTR), and improving the resilience of microservice-based systems.

Experimental results confirmed that combining coordinated dependency updates with Blue/Green deployment strategies can practically eliminate the transient failures commonly seen with rolling updates, even when versions are otherwise compatible. This positions the orchestrator as a particularly valuable approach for mission-critical domains, including energy, aviation, and financial services, where downtime is unacceptable.

Nevertheless, this method requires investment in supporting infrastructure, including a centralized API registry, automated compatibility verification, and sophisticated monitoring capabilities. Its complexity may be excessive for small-scale or experimental projects with lower risk profiles.

Future research should focus on scaling the framework and validating its effectiveness under industrial-grade workloads. In addition, aligning its security model with Zero-Trust requirements would help strengthen external validity and industry adoption.

Author Contributions: M.Y.: Conceptualization, methodology, formal analysis, writing—original draft preparation, visualization, conducting experiments. I.P.: Methodology, software, formal analysis, writing—original draft preparation, conducting experiments. A.P.: Investigation, resources and editing, visualization. H.K.: Supervision, writing—review and editing. V.A.: Supervision, visualization, writing—review and editing. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding

Institutional Review Board Statement: Not applicable

Informed Consent Statement: Not applicable

Data Availability Statement: The source code used to conduct the experiments described in this study is publicly available at https://gitlab.com/frameork-orchestartor-api-versions. This repository contains all relevant scripts, configuration files, and documentation necessary to reproduce the results.

Acknowledgments: This work represents a harmonious blend of independent research pursuits and a series of scientifically rigorous endeavours underpinned by various grants and funding sources. The authors would like to extend their heartfelt gratitude to all the institutions and organisations, reviewers, and our editor who have contributed to the successful completion of this study. The research was conducted as part of the projects "Development of methods and means of increasing the efficiency and resilience of local decentralized electric power systems in Ukraine" and "Development of Distributed Energy in the Context of the Ukrainian Electricity Market Using Digitalization Technologies and Systems", implemented under the state budget program "Support for Priority Scientific Research and Scientific-Technical (Experimental) Developments of National Importance"

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 July 2025 doi:10.20944/preprints202507.0710.v1

(CPCEL 6541230) at the National Academy of Sciences of Ukraine. Specially, the authors would like to express our sincere gratitude to Volodymyr Mokhor, the Director of the G.E. Pukhov Institute for Modelling in Energy Engineering of NAS of Ukraine for his invaluable support and assistance in facilitating this research.

During the preparation of this manuscript, the authors used ChatGPT-4o (OpenAI) and Grammarly for grammar and spelling checks, sentence refinement, and improving overall clarity. The authors have reviewed and edited the output and take full responsibility for the content of this publication.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

API	Application Programming Interface
CI	Continuous Integration
CD	Continuous Deployment (or Delivery)
JSON	JavaScript Object Notation
gRPC	gRPC Remote Procedure Call
HTTP	HyperText Transfer Protocol
REST	Representational State Transfer

References

1. Woods, E. Software architecture in a changing world. *IEEE Softw.* **2016**, *33*, 94–97. https://doi.org/10.1109/MS.2016.149.
2. Richards, M.; Ford, N. *Fundamentals of Software Architecture: An Engineering Approach*, 1st ed.; O'Reilly Media: Beijing, Boston, Farnham, Sebastopol, Tokyo, 2020.
3. Fowler, M; Lewis, J. Microservices. Available online: https://martinfowler.com/articles/microservices.html (accessed on 19 May 2025).
4. Söylemez, M.; Tekinerdogan, B.; Tarhan, A.K. Microservice reference architecture design: A multi-case study. *Softw. Pract. Exp.* **2024**, *54*, 58–84.
5. Choi, H.-J.; Kim, J.-H.; Lee, J.-H.; Han, J.-Y.; Kim, W.-S. Adaptive Microservice Architecture and Service Orchestration Considering Resource Balance to Support Multi-User Cloud VR. *Electronics* **2025**, *14*, 1249. https://doi.org/10.3390/electronics14071249.
6. Faustino, D.; Gonçalves, N.; Portela, M.; Silva, A.R. Stepwise migration of a monolith to a microservice architecture: Performance and migration effort evaluation. *Perform. Eval.* **2024**, *164*, 102411.
7. Narváez, D.; Battaglia, N.; Fernández, A.; Rossi, G. Designing Microservices Using AI: A Systematic Literature Review. *Software* **2025**, *4*, 6. https://doi.org/10.3390/software4010006.
8. Ye, S.; Saukh, V.; Puchko, T.V. Sustainable web API evolution: Forecasting software development effort. *IOP Conf. Ser. Earth Environ. Sci.* **2024**, *1415*, 012077.
9. Bonorden, L.; Riebisch, M. API deprecation: A systematic mapping study. In *Proceedings of the 2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2022; pp. 451–458.
10. Zdun, U.; Wittern, E.; Leitner, P. Emerging trends, challenges, and experiences in DevOps and microservice APIs. *IEEE Softw.* **2020**, *37*, 87–91.
11. Zhou, X.; Peng, X.; Xie, T.; Sun, J.; Ji, C.; Li, W.; Ding, D. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Trans. Softw. Eng.* **2021**, *47*, 243–260.
12. Lercher, A.; Glock, J.; Macho, C.; Pinzger, M. Microservice API evolution in practice: A study on strategies and challenges. *J. Syst. Softw.* **2024**, *215*, 112110. https://doi.org/10.1016/j.jss.2024.112110
13. Cerny, T.; Donahoo, M.J.; Trnka, M. Contextual understanding of microservice architecture: Current and future directions. *SIGAPP Appl. Comput. Rev.* **2018**, *17*, 29–45. https://doi.org/10.1145/3183628.3183631
14. Lercher, A.; Bauer, C.; Macho, C.; Pinzger, M. AutoGuard: Reporting Breaking Changes of REST APIs from Java Spring Boot Source Code (Tool-Demo Paper, SANER 2025). 2024. Available online: https://pinzger.github.io/papers/Lercher2025-AutoGuard.pdf (accessed on 19 May 2025).
15. Waseem, M.; Liang, P.; Shahin, M.; Ahmad, A.; Nassab, A.R. On the nature of issues in five open source microservices systems: An empirical study. In *Proceedings of the 25th International Confer-*

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 July 2025 doi:10.20944/preprints202507.0710.v1

ence on Evaluation and Assessment in Software Engineering, New York, NY, USA, 2021; pp. 201–210. https://doi.org/10.1145/3463274.3463337.

16. Musavi, P.; Adams, B.; Khomh, F. Experience report: An empirical study of API failures in OpenStack cloud environments. In *Proceedings of the 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, Ottawa, ON, Canada, 23–27 October 2016; pp. 424–434. https://doi.org/10.1109/ISSRE.2016.42.
17. Lelovic, L.; Huizenga, A.; Goulis, G.; Kaur, A. Change Impact Analysis in Microservice Systems: A Systematic Literature Review. *J. Syst. Softw.* **2024**, *219*, 112241. https://doi.org/10.1016/j.jss.2024.112241.
18. Cerny, T.; Goulis, G.; Abdelfattah, A.S. Towards Change Impact Analysis in Microservices-Based System Evolution. *arXiv* **2025**, preprint. https://doi.org/10.48550/arXiv.2501.11778.
19. Xavier, L.; Brito, A.; Hora, A.; Valente, M.T. Historical and impact analysis of API breaking changes: A large-scale study. In *Proceedings of the 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Klagenfurt, Austria, 20–24 February 2017; pp. 138–147. https://doi.org/10.1109/SANER.2017.7884616.
20. Chen, F.; Zhang, L.; Lian, X. A systematic gray literature review: The technologies and concerns of microservice application programming interfaces. *Softw. Pract. Exp.* **2021**, *51*, 1483–1508. https://doi.org/10.1002/spe.2967.
21. Morris, B. REST APIs don't need a versioning strategy—they need a change strategy. Available online: https://www.ben-morris.com/rest-apis-dont-need-a-versioning-strategy-they-need-a-change-strategy/ (accessed on 19 May 2025).
22. Dackebro, E. *An Empirical Investigation into Problems Caused by Breaking Changes in API Evolution*; Master's Thesis, School of Electrical Engineering and Computer Science, KTH Royal Institute of Technology: Stockholm, Sweden, 2019. Available online: https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-255015 (accessed on 19 May 2025).
23. Robbes, R.; Lungu, M. A study of ripple effects in software ecosystems: (NIER track). In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, Honolulu, HI, USA, 21–28 May 2011; pp. 904–907. https://doi.org/10.1145/1985793.1985940.
24. Brito, A.; Valente, M.T.; Xavier, L.; Hora, A. You broke my code: Understanding the motivations for breaking changes in APIs. *Empir. Softw. Eng.* **2020**, *25*, 1458–1492. https://doi.org/10.1007/s10664-019-09756-z.
25. Musavi Mirkalaei, S.P. API Failures in OpenStack Cloud Environments; Master's Thesis, École Polytechnique de Montréal: Montreal, QC, Canada, 2017. Available online: https://publications.polymtl.ca/2715/ (accessed on 19 May 2025).
26. Brito, A.; Xavier, L.; Hora, A.; Valente, M.T. APIDiff: Detecting API breaking changes. In *Proceedings of the 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Campobasso, Italy, 20–23 March 2018; pp. 507–511. https://doi.org/10.1109/SANER.2018.8330249.
27. Granli, W.; Burchell, J.; Hammouda, I.; Knauss, E. The driving forces of API evolution. In *Proceedings of the 14th International Workshop on Principles of Software Evolution*, New York, NY, USA, 31 August 2015; pp. 28–37. https://doi.org/10.1145/2804360.2804364.
28. Di Lauro, F.; Serbout, S.; Pautasso, C. Towards large-scale empirical assessment of Web APIs evolution. In *Web Engineering*; Brambilla, M., Chbeir, R., Frasincar, F., Manolescu, I., Eds.; Springer International Publishing: Cham, Switzerland, 2021; pp. 124–138. https://doi.org/10.1007/978-3-030-74296-6_10.
29. Lehman, M.M. On understanding laws, evolution, and conservation in the large-program life cycle. *J. Syst. Softw.* **1979**, *1*, 213–221. https://doi.org/10.1016/0164-1212(79)90022-0.
30. Teivonen, P. Staging Environment Implementation in a Software Delivery Automation Pipeline; Bachelor's Thesis, Oulu University of Applied Sciences: Oulu, Finland, 2024. Available online: http://www.theseus.fi/handle/10024/862900 (accessed on 19 May 2025).
31. Everett, G.D.; McLeod, R., Jr. *Software Testing: Testing Across the Entire Software Development Life Cycle*; John Wiley & Sons: Hoboken, NJ, USA, 2007; ISBN 978-0-470-14634-7.
32. Fowler, S.J. *Production-Ready Microservices: Building Standardized Systems Across an Engineering Organization*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2016; ISBN 978-1-4919-6592-4.
33. Desikan, S.; Ramesh, G. *Software Testing: Principles and Practice*; Pearson Education India: Delhi, India, 2006; ISBN 978-81-7758-121-8.
34. Beilharz, J.; Wiesner, P.; Boockmeyer, A.; Brokhausen, F.; Behnke, I.; Schmid, R.; Pirl, L.; Thamsen, L. Towards a Staging Environment for the Internet of Things. In *Proceedings of the 2021 IEEE International Conference on Pervasive Computing and Communications Workshops and Other Affiliated Events (PerCom Workshops)*, Kassel, Germany, 22–26 March 2021; pp. 312–315. https://doi.org/10.1109/PerComWorkshops51409.2021.9431087.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 July 2025 doi:10.20944/preprints202507.0710.v1

35. Pact Docs. Introduction. Available online: https://docs.pact.io/ (accessed on 19 May 2025).
36. Carver, J.C.; Chue Hong, N.P.; Thiruvathukal, G.K. *Software Engineering for Science*; CRC Press: Boca Raton, FL, USA, 2016; ISBN 978-1-315-35192-6.
37. Viotti, J.C.; Kinderkhedia, M. A Survey of JSON-Compatible Binary Serialization Specifications. *arXiv* **2022**, arXiv:2201.02089. https://doi.org/10.48550/arXiv.2201.02089.
38. Loechel, L.; et al. Hook-in Privacy Techniques for gRPC-Based Microservice Communication. In *Web Engineering*; Springer Nature: Cham, Switzerland, 2024; pp. 215–229. https://doi.org/10.1007/978-3-031-62362-2_15.
39. Babal, H. *gRPC Microservices in Go*; Simon and Schuster: New York, NY, USA, 2023.
40. Friesen, J. *Java XML and JSON: Document Processing for Java SE*; Apress: Berkeley, CA, USA, 2019.
41. Richardson, C. *Microservices Patterns: With Examples in Java*; Manning Publications: Shelter Island, NY, USA, 2018.
42. Tulach, J. Practical API Design; Apress: Berkeley, CA, USA, 2008. https://doi.org/10.1007/978-1-4302-0974-4.
43. Humble, J.; Farley, D. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*; Pearson Education: Boston, MA, USA, 2010.
44. Servile, V. *Continuous Deployment*; O'Reilly Media: Sebastopol, CA, USA, 2024.
45. Bhuyan, A.P. *Microservices Design Patterns: Best Practices for Cloud-Native Architectures*; Self-Published: India, 2024.
46. Serbout, S.; Pautasso, C. An Empirical Study of Web API Versioning Practices. In *Web Engineering*; Garrigós, I., Murillo Rodríguez, J.M., Wimmer, M., Eds.; Springer Nature: Cham, Switzerland, 2024. (DOI pending if available).
47. Knoche, H.; Hasselbring, W. Continuous API Evolution in Heterogeneous Enterprise Software Systems. In *Proceedings of the 2021 IEEE 18th International Conference on Software Architecture (ICSA)*, Stuttgart, Germany, 22–26 March 2021; pp. 58–68. https://doi.org/10.1109/ICSA51549.2021.00014.
48. Koçi, R.; Franch, X.; Jovanovic, P.; Abelló, A. Classification of Changes in API Evolution. In *Proceedings of the 2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*, Paris, France, 28–31 October 2019; pp. 243–249. https://doi.org/10.1109/EDOC.2019.00037.
49. Singhal, V. What Is Staging Environment & Why You Should Have One. 2024. Available online: https://instawp.com/what-is-staging-environment/ (accessed on 19 May 2025).
50. Nilslice. Protolock: Protocol Buffer Lockfile Utility. Available online: https://github.com/nilslice/protolock (accessed on 6 July 2025).
51. IBM. jsonsubschema: Subtype-Based Comparison Library for JSON Schemas. Available online: https://github.com/IBM/jsonsubschema (accessed on 6 July 2025).
52. Habib, A.; Shinnar, A.; Hirzel, M.; Pradel, M. Finding Data Compatibility Bugs with JSON Subschema Checking. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Virtual Event, 11–17 July 2021; pp. 620–632. https://doi.org/10.1145/3460319.3464796.
53. Lam, P.; Dietrich, J.; Pearce, D.J. Putting the Semantics into Semantic Versioning. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Chicago, IL, USA, 16–21 November 2020; pp. 157–179. https://doi.org/10.1145/3426428.3426922.
54. European Parliament and Council of the European Union. Regulation (EU) 2024/2847 of 23 October 2024 on Horizontal Cybersecurity Requirements for Products with Digital Elements (Cyber Resilience Act). *EUR-Lex*, 2024. Available online: https://eur-lex.europa.eu/eli/reg/2024/2847/oj/eng (accessed on 19 May 2025).
55. DO-178C - Software Considerations in Airborne Systems and Equipment Certification. RTCA. 2024. Available online: https://my.rtca.org/productdetails?id=a1B36000001IcmqEAC (accessed on 19 May 2025).
56. Sun, R.; Zhong, D.; Li, W.; Lu, M.; Ding, Y.; Xu, Z.; Gong, H.; Zha, Y. A Safety Analysis Method of Airborne Software Based on ARP4761. *J. Phys. Conf. Ser.* **2021**, *1654*, 012022. https://doi.org/10.1088/1742-6596/1673/1/012045
57. Weedy, B.M.; Cory, B.J.; Jenkins, N.; Ekanayake, J.B.; Strbac, G. *Electric Power Systems*, 5th ed.; John Wiley & Sons: Hoboken, NJ, USA, 2012; ISBN 978-1-118-36108-5.
58. Sirotkin, O., Prymushko, A., Puchko, I., Kravtsov, H., Yaroshynskyi, M., & Artemchuk, V. (2025). Parallel Simulation Using Reactive Streams: Graph-Based Approach for Dynamic Modeling and Optimization. Computation, 13(5), 103. https://doi.org/10.3390/computation13050103

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 8 July 2025 doi:10.20944/preprints202507.0710.v1

59. Prymushko, A., Puchko, I., Yaroshynskyi, M., Sinko, D., Kravtsov, H., & Artemchuk, V. (2025). Efficient State Synchronization in Distributed Electrical Grid Systems Using Conflict-Free Replicated Data Types. IoT, 6(1), 6. https://doi.org/10.3390/iot6010006
60. Yaroshynskyi, M., Prymushko, A., Puchko, I., Sirotkin, O. and Sinko, D., 2025. Akka as a tool for modelling and managing a smart grid system. Journal of Edge Computing , 4(1), pp.105–115. Available from: https://doi.org/10.55056/jec.822

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.