Global Journal of Computer Science and Technology: H Information & Technology
Volume 23 Issue 2 Version 1.0 Year 2023
Type: Double Blind Peer Reviewed International Research Journal
Publisher: Global Journals
Online ISSN: 0975-4172 & Print ISSN: 0975-4350

Scalability and Performance of Microservices Architectures

By Amirali Kerimovs

Annotation- The inevitability of continuous evolution and seamless integration of dynamic alterations remains a paramount consideration in the realm of software engineering. This concern is particularly pronounced within the context of contemporary microservices architectures embedded in heterogeneous and decentralized systems composed of numerous interdependent components. A pivotal focal point within such a software design paradigm is to sustain optimal performance quality by ensuring harmonious collaboration among autonomous facets within an intricate framework. The challenge of microservices evolution has predominantly revolved around upholding the harmonization of diverse microservices versions during updates, all while curbing the computational overhead associated with such validation. This study leverages previous research outcomes and tackles the evolution predicament by introducing an innovative formal model, coupled with a fresh exposition of microservices RESTful APIs. The amalgamation of Formal Concept Analysis and the Liskov Substitution Principle plays a pivotal role in this proposed solution. A series of compatibility constraints is delineated and subjected to validation through a controlled experiment employing a representative microservices system. The suggested approach is poised to enhance the development of more sustainable microservices applications and elevate the efficacy of DevOps practices engaged in the creation and upkeep of microservices architectures.

Keywords: *software engineering, project management, microservices; API compatibility; service evolution; concept lattices; substitution principle.*

GJCST-H Classification: *ACM Code: D.2.11*

Strictly as per the compliance and regulations of:

Scalability and Performance of Microservices Architectures

Amirali Kerimovs

Annotation- The inevitability of continuous evolution and seamless integration of dynamic alterations remains a paramount consideration in the realm of software engineering. This concern is particularly pronounced within the context of contemporary microservices architectures embedded in heterogeneous and decentralized systems composed of numerous interdependent components. A pivotal focal point within such a software design paradigm is to sustain optimal performance quality by ensuring harmonious collaboration among autonomous facets within an intricate framework. The challenge of microservices evolution has predominantly revolved around upholding the harmonization of diverse microservices versions during updates, all while curbing the computational overhead associated with such validation. This study leverages previous research outcomes and tackles the evolution predicament by introducing an innovative formal model, coupled with a fresh exposition of microservices RESTful APIs. The amalgamation of Formal Concept Analysis and the Liskov Substitution Principle plays a pivotal role in this proposed solution. A series of compatibility constraints is delineated and subjected to validation through a controlled experiment employing a representative microservices system. The suggested approach is poised to enhance the development of more sustainable microservices applications and elevate the efficacy of DevOps practices engaged in the creation and upkeep of microservices architectures.

Keywords: software engineering, project management, microservices; API compatibility; service evolution; concept lattices; substitution principle.

I. Introduction

In an era where the intricacies and ingenuity of software systems' business logic continue to evolve, microservices have garnered a heightened degree of contemporary attention. This architectural paradigm configures systems as constellations of autonomously deployable services, each strategically aligned to fulfill specific business requisites within well-defined bounds of a domain model. While the merits of this approach abound – allowing for the development, implementation, and deployment of individualized microservices with novel functionalities and bug fixes sans the need to overhaul the entire system – it is not without its inherent constraints.

Amidst the array of benefits, complexities emerge, exemplified by the elevated overall intricacy of the system and the amplified level of inter-service communication. Furthermore, the introduction of each successive iteration of a deployed microservice holds the potential to influence its dependent counterparts, engendering challenges of compatibility both retroactively and proactively, unless meticulous design is observed.

The preceding research encompassed an analysis aimed at juxtaposing two distinct software architectural paradigms – microservices and monolithic architectures. The scope of this analysis encompasses the arrangement of architectural components, the dynamics of their interactions, and the constraints governing these intricate interplays [1]. The findings underscored the advantages inherent to microservices. Among these merits, heightened module isolation emerges as a pivotal facet, facilitating autonomous deployment tracks, embracing heterogeneous technological selections, and fostering augmented productivity [5]. Furthermore, the prevalence of loosely-coupled constructs, exemplified by microservices, engenders an instantaneous impact upon deploying new iterations, thus catering seamlessly to the demands of swiftly-evolving systems. This concurrent flexibility is further underscored by the absence of constraints on the quantity of versions amenable to deployment – an essential feature, considering the perpetual need to incorporate novel functionalities or enhance existing ones.

However, this effusion of microservices, accompanied by their intricate interconnections, begets a palpable challenge concerning the management of contracts and seamless service integration. Consequently, the imperative to anticipate and adeptly manage such transformations at every stage of software development looms large. In essence, the predicament accentuates the exigency for a novel approach, one that can be seamlessly incorporated within the context of this emergent architectural paradigm known as micro-services. This approach ought to draw inspiration from prior research that addresses the crucial facet of ensuring service compatibility within the realm of service-oriented architectures.

II. Literature Review

The intricate challenge of ensuring seamless compatibility throughout the evolutionary journey of services has garnered notable attention in recent years, prompting diverse avenues of research exploration. In our pursuit of comprehensively addressing this intricacy,

Author: Independent Researcher, Riga, Latvia.
e-mail: kerimovsoftdev@gmail.com

we have undertaken a methodical systematic literature review [2], a scholarly endeavor geared towards dissecting the present landscape of this compelling issue.

Within the confines of this research endeavor, we have meticulously articulated a set of stringent prerequisites that guide the selection and evaluation of pertinent studies. These stipulations delineate a clear trajectory for research inclusion: the chosen studies must be firmly rooted within the expansive domain of services/microservices evolution; they must actively encapsulate a meticulously formulated perspective of the intricate conundrum surrounding services /microservices compatibility; and most notably, the studies must introduce novel methodologies that bear their own consequential implications, encapsulating the potential to unravel the complexities of this dynamic arena.

A pivotal cornerstone within our examination is the insightful contribution presented within the paper [10]. This scholarly exposition delves intently into the realm of microservices versioning, meticulously probing their compatibility while concurrently orchestrating a blueprint for the orchestrated deployment of harmonious configurations. At the heart of this endeavor lies the formulation of a version dependency model, an ingeniously crafted framework geared towards facilitating deployment management. The underpinning mechanics of this model are deeply rooted in the construct of a service dependency graph, augmented by a judiciously devised greedy-based optimization algorithm. Significantly, the discourse within [10] does not merely conclude with this innovative framework; rather, it resonates with the clarion call for the expansion and adaptation of existing algorithms to deftly accommodate the evolving landscape of user requisites. A compelling crescendo within this scholarly odyssey is the articulation of an overarching objective – the dual quest to minimize the average response time while concurrently curtailing the formidable evolution cost.

In the grand tapestry of research inquiries, our efforts converge at the crossroads of innovation, grappling with the multifaceted challenge of services evolution and compatibility. Through our systematic review and scholarly analysis, we endeavor to chart a course towards more streamlined and effective methodologies that illuminate the path forward in the intricate realm of services/microservices evolution.

The primary objective of the study documented in [7] revolved around the formulation of an intricate theoretical framework, one that orchestrates the management of service evolution with a nuanced focus encompassing the realms of structure, behavior, and Quality of Service (QoS) induced alterations. A cardinal aspiration was to engineer a paradigm that would seamlessly accommodate these changes while preserving the integrity of type-safe operations. To this end, the research introduced a methodical formalization of the notion of "service compatibility." This endeavor harnessed the potent construct of Abstract Service Descriptions (ASD), meticulously stratified into three distinct layers: structural, behavioral, and non-functional attributes. Notably, the subtyping relation within ASD records was harnessed as a linchpin mechanism for scrutinizing the compatibility interplay among different versions of services.

As the intellectual terrain was traversed, a panorama of diverse methodologies emerged within the purview of microservices evolution. These scholarly offerings unveiled a spectrum of approaches, ranging from those that demonstrated commendable efficacy within real-world contexts to those necessitating recalibration and adaptation to more practical projects. A preponderance of these endeavors pivoted around grappling with the evolutionary challenge at the nexus of software testing and deployment. However, a salient revelation surfaced, spotlighting the paucity of approaches attuned to tackling this predicament at the architectural echelon. Within the crucible of this review, a pivotal quandary crystallized – one that revolves around harnessing the intrinsic features of microservice architecture to fortify compatibility amidst the relentless march of microservices evolution.

Within the contours of this very paper, we embark upon an exploration that envisions an API as an intricate set of meticulously defined regulations that underpin seamless communication between disparate applications. Notably, our discerning focus zeroes in on the contours and intricacies of the RESTful API, an architectural facet meticulously aligned with the design tenets of the REST architectural style. Through we need to unravel the interplay of intricacies inherent within this realm, with the overarching aim of ushering forth a more harmonious coexistence and evolution of microservices.

a) The Purpose of the Article

In this work, the definition of application programming interfaces (APIs) of microservices is formalized, which highlights the specific features of this type of architecture that can be used to maintain compatibility within the framework of updating their versions.

b) Presentation of the Main Material

Effective communication stands as a cornerstone of optimal performance within the realm of microservices architecture. However, it is pivotal to recognize that the microservices responsible for exposing an interface, and the corresponding microservices that leverage said interface, manifest as distinct and autonomously deployable entities [3]. In the event of modifications that disrupt backward compatibility, developers are confronted with a critical choice: either synchronize the deployment of consuming

microservices alongside the introduction of the altered interface, ensuring their seamless transition to the new iteration, or devise a strategy to stagger the introduction of the updated microservice contract. This underscores the imperative of effecting changes in microservices with utmost caution, vigilantly safeguarding compatibility with the intricate web of consuming microservices. An inherent tenet of successful implementation involves the seamless incorporation of backward-compatible alterations, underscored by the imperative to render interfaces conspicuously explicit. In this pursuit, schemas emerge as invaluable tools, assuming the role of custodians to ensure compatibility is sustained.

Furthermore, the underlying technology facilitating this evolution must be engineered with an unwavering commitment to agnosticism, shunning integration technologies that dictate the technology stacks mandated for microservices implementation. An exemplar technology poised to support the seamless evolution of microservices is the REpresentational State Transfer (REST). At the core of this architectural paradigm is the cardinal concept of resources and their multifaceted representations. A pivotal facet is the decoupling of external resource representations from their internal storage mechanisms. This structural underpinning empowers clients to orchestrate requests for alterations to resources, with the server endowed with the prerogative to accede or abstain from compliance.

In the realm of consumption, the OpenAPI specification assumes paramount significance. This specification offers a potent avenue to meticulously outline an array of essential information pertaining to REST endpoints, thereby furnishing the groundwork necessary for the generation of client-side codes across a gamut of programming languages. In essence, the article delves into the intricate fabric of microservices communication, emphasizing compatibility, technology-agnosticism, and the transformative potential of REST and the OpenAPI specification as pivotal instruments in the orchestrated evolution of microservices architectures.

Polymorphism is another important concept that enables procedures or data abstractions to work for multiple types. For example, a sorting procedure should work for any type of element in the array as long as it is possible to compare them. This is called polymorphism. When related types exist in a program, polymorphism is likely to be used, especially when a polymorphic module is needed. In such cases, the supertype is often virtual and serves as a placeholder in the hierarchy for the related types [7].

Let sets be defined in the information system: users U = {u1, u2, u3, u4, u5} and powers P = {p1, p2, p3, p4}. The mapping UP: U → 2P is given by the matrix UP (Fig. 1a). Consider the formal context PU = (P, U, PU), whose matrix PU is determined by the rule: PU = UPT (Fig. 1). The hierarchy of roles generated by the lattice of PU context concepts is shown in Fig. 1c. Roles have the following scope and content.

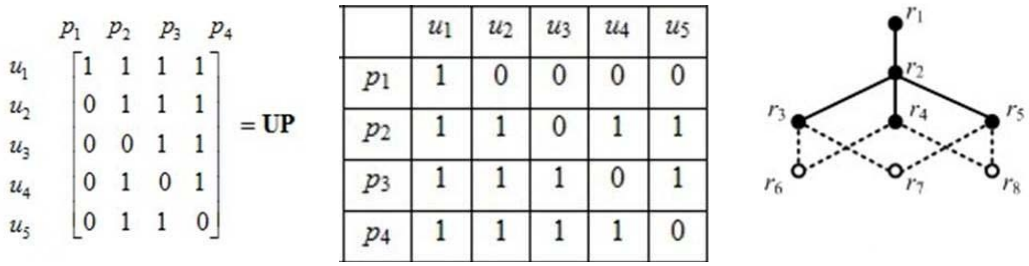


Fig. 1: The process of building a role graph: a) mapping *UP*; b) formal context; c) hierarchy of roles (orientation of arcs: "top-down").

Thus, the main mappings *RP: R → 2 P*, *UR: U → 2 R* and *RR: R → 2 R* will be described by the matrices RP, UR and RR respectively. The proposed algorithm for building a role-based access control model can be used to optimize the existing role hierarchy in the system. Let's demonstrate this approach with an example. This section will show how to move from interfaces and microservices systems to concept grids (FCA approvals).

Information visualization is an important part of data mining (DMA). Most people perceive graphical information best. The visual representation enables information to be visualized and often reveals patterns at a glance that would otherwise only be found through time-consuming analysis. When solving many problems, the task of visualizing partially ordered sets, as well as their particular case, lattices, arises. This problem is especially relevant when using one of the most powerful methods of IAD-the analysis of formal concepts. Formal Concept Analysis (FCA) was proposed by Wille in 1981 [9] and is still actively developed today. In this thesis, the research methodology employed is design science research, which aims to create and evaluate IT artifacts designed to address specific organizational issues. This approach was deemed appropriate for the current project, as the

end goal is to develop and implement a DevOps pipeline as an artifact. However, several challenges must be tackled in order to achieve this, including identifying and motivating the problems, defining solution objectives, designing and developing the artifact, demonstrating its efficacy, evaluating its impact, and communicating the findings. Figure 9 depicts the process model for the design science research methodology (DSRM).

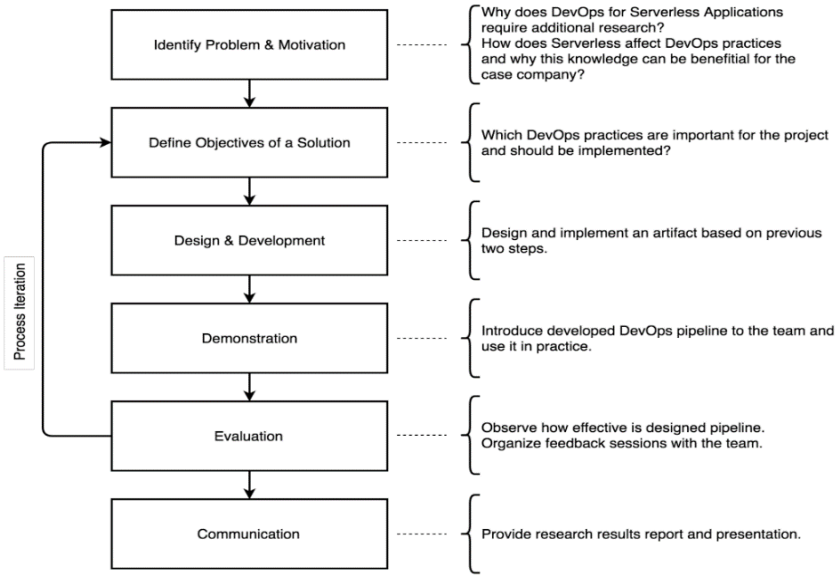


Fig. 2: DSRM Process Model

A formal concept (*concept*) consists of scope and content. Content is a set of properties that describe a concept. The scope includes all objects from the context that have all the properties from the content. In this case, the content should be maximum, i.e., include all the general properties of objects from the scope of the concept. A mathematically formal concept is given using the Galois correspondences [3].

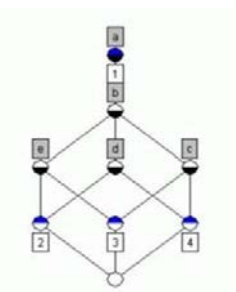
A precedence relation is established between concepts: concept $(A_1, B_1) \leq (A_2, B_2)$ if $A_1 \subseteq A_2$ and $B_1 \supseteq B_2$, where A_1, A_2 are the volumes of concepts 1 and 2, and B_1, B_2 - respectively, their content. Traditionally, Hasse diagrams are used to visualize partially ordered sets. In FCA, a variation of them is used, which uses an abbreviated label - each object and attribute is shown on the diagram only once [4]. The name of an object is assigned to the intersection of all concepts that contain this object, and the name of a property is assigned to the union of all concepts whose contents include this property. Thus, the name of an object is assigned to the smallest of the concepts in which the given object occurs, and the name of the property is assigned to the largest of the concepts in which the property occurs. Such charts are called line charts.

A good line chart should be "transparent, easy to read and easy to interpret the data presented" [8]. However, how this is achieved depends on the goals of the interpretation and on the structure of the lattice.

When drawing a line diagram, it is mandatory that all subconcepts of a concept must be located below it.

	a	b	c	d	e
1	×				
2	×	×		×	×
3	×	×	×		×
4	×	×	×	×	

a)



b)

Fig. 3: a) Formal Context b) Line Diagram of the Lattice of Concepts

It is desirable that the following conditions be met [6]:

- Edges should be drawn as straight lines;
- Two vertices must not be located at the same point;
- The number of intersections between lattice edges should be as small as possible;
- An edge must not intersect a vertex that is not its end;
- The lattice structure should be visually represented;
- Using as few different directions as possible;
- Maximization of the number of parallel lines.

The Liskov Substitution Principle (LSP) is a fundamental principle of object-oriented programming that states that if a program is using a base class, it should be able to use any of its derived classes without knowing it. In the context of APIs evolution processes, this principle can be used to ensure that changes made to the system do not cause unexpected side effects or break existing functionality [11]. Here are some examples of how LSP can be applied in APIs evolution processes:

- Adding a new microservice to a system should not break any existing interfaces or dependencies between services. If the new microservice follows the same interface and dependency rules as the existing services, it can be safely added to the system without any negative impacts.
- Upgrading a software library or tool should not require any changes to the existing codebase that uses it. If the new version of the library or tool is backward-compatible and does not introduce any breaking changes, it can be safely upgraded without causing any issues.
- Adding new features to an application should not break any existing features or workflows. If the new features follow the same patterns and conventions as the existing ones, they can be safely added without causing any issues for users.

In all of these examples, following the Liskov Substitution Principle leads to permissible changes that do not break existing functionality or introduce unexpected side effects. By ensuring that changes are made in a way that adheres to this principle, teams can maintain the stability and reliability of the system over time. To ensure the necessary formalization of microservices, the method of formal concepts analysis (FCA) [14] is used to classify datasets describing microservices APIs into conceptual structures. Then a concept lattice is constructed, which can be used to extract the hierarchical order of concepts, as well as meanings and associations between concepts.

It is expected that this presentation of the API description as a concept and consideration of the evolution of the API as a lattice of concepts will help determine which changes can be compatible. To represent an API as a concept lattice within FCA, it is possible to use the following mathematical equations [14]:

1) A context C = (G, M, I) is defined as a binary relation between objects G (functions or methods) and attributes M (input parameters or output types). The incidence matrix I represents the presence or absence of attributes for each object.
2) The set of all concepts in the lattice L can be obtained using the Galois connection demonstrated in equations (1) and (2):

$$f: 2^G \to 2^M, f(A) = \{m \in M | A \subseteq \{g \in G | (g, m) \in I\}\} \quad (1)$$

$$g: 2^M \to 2^G, g(B) = \{g \in G | B \subseteq \{m \in M | (g, m) \in I\}\} \quad (2)$$

3) The set of all formal concepts in the lattice can be represented as equation (3):

$$L = \{(A, B) | A \subseteq G, B \subseteq M, f(A) = B, g(B) = A\} \quad (3)$$

4) The lattice is formed by pairs of concepts (A, B), where A is a subset of B. The bottom concept represents the set of all objects in the API, while the top concept represents the empty set.

The evolution of an API can be represented as a sequence of concept lattices. Each lattice can be represented as a node in the graph, and the changes made between versions of the API can be represented as edges between the corresponding nodes. The sublattice hierarchy can also be represented as edges between the nodes in the graph.

API compatibility is a measure of how well different versions of an API can interoperate with each other. In general, API compatibility means that changes made to an API should not break existing applications that rely on that API. Let API_v1 be the set of functions and methods provided by version v1 of the API. Let API_v2 be the set of functions and methods provided by version v2 of the API. API compatibility can be expressed as a relation between API_v1 and API_v2, denoted as API_v1 is compatible with API_v2, or symbolically, API_v1 ⊆ API_v2.

This means that all the functions and methods provided by v1 are also provided by v2. It also means that the behavior of these functions and methods in v1 is preserved in v2, i.e., for any f in API_v1, f (args) in v1 should produce the same result as f (args) in v2. Finally, new functionality can be added to v2 without breaking existing applications that rely on v1, i.e., API_v2 \ API_v1 contains only new functions and methods that do not affect the behavior of the existing ones. At the same time, as a binary relation defining a semilattice of concepts, the type hierarchy rule is used to preserve compatibility, or in another way, the substitution principle of Barbara Liskov (Liskov Substitution

Principle) [7] with the following is accepted: φ(x), φ(y) ∈ M - multiple context attributes, x, y ∈ G - a set of context objects, where x has the type T, and y has the type S, in this case , subsets A, B are set to define concepts so that A ⊆ G и B ⊆ M. Definition of concepts: (A1, B1) - the concept of type S, (A2, B2) - the concept of type T. Then, according to the hierarchy of concepts, the concept of type S (A1, B1) is under the concept of a type concept T (A2, B2): (A1, B1) ≤ (A2, B2) when A1 ⊆ A2. Equivalent to (A1, B1) ≤ (A2, B2), B1 ⊇ B2, where in turn is the type S is a subtype of T. Let C be a concept lattice that represents an API, with a set of objects O and a set of attributes A. Let C1 and C2 be two concept lattices that represent different versions of the API, where C2 is a subclass of C1, denoted as C2 ≤ C1. This means that all the objects and attributes in C1 are also in C2, and C2 may have additional objects and attributes.

To apply the Liskov Substitution Principle to concept lattices, it need to ensure that replacing an object from C1 with an object from C2 does not affect the correctness of the program that uses the API. This can be mathematically expressed as [4]: for any object o ∈ O and attribute a ∈ A, if o has the attribute a in C1, then o also has the attribute a in C2. Therefore, by applying the Liskov Substitution Principle to concept lattices and using FCA algorithms to analyze the sublattice structure of the API, ensure that changes made to the API preserve the structure and behavior of the API and do not break backward compatibility.

To represent the API changes using concept lattices, we can first define the lattice structure in terms of the attributes and operations that are supported by the API. We can represent these attributes and operations as concepts in a lattice structure, as shown below:

```graphql
        Add
       /   \
Vector     Scalar
       \   /
      Multiply
```

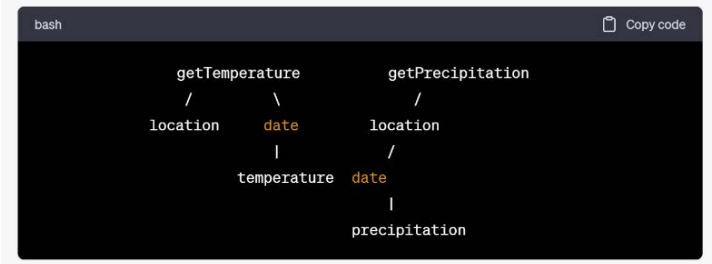
Now, let's say we want to update the API to support matrices. We can add a new attribute "Matrix" to the lattice, and update the "Multiply" operation to support both vectors and matrices, as shown below:

```graphql
        Add
       /   \
Vector     Scalar
       \   /
      Multiply
       /   \
   Matrix   Vector
```

Let's consider another example in the context of a specific microservice, where the objects are API methods and the attributes are variables and data objects (this is the context of a specific microservice). Suppose we have a microservice that provides weather information to users. The microservice has several API methods that provide information about temperature, wind speed, humidity, etc. Each method takes a set of parameters and returns a result. We can represent the API methods and their parameters as objects, and the variables and data objects used in the methods as attributes. For example, the "get Temperature" API method might have the following objects and attributes:

```bash
          getTemperature
           /         \
      location      date
                     |
                temperature
```

Suppose we want to add a new API method that provides information about precipitation. We can add a new object "get Precipitation" to the lattice, along with the relevant attributes. The lattice would now look like this:



We can use the Liskov substitution principle to check the compatibility of the updated API, as before. If the new API method and its parameters can be substituted for the existing API methods without causing unexpected behavior, then they are compatible.

III. Conclusion

The challenge encompassing the evolution of microservices is decisively delineated, underpinned by the compelling necessity to curtail resource expenditure entailed in the verification of microservices compatibility during the intricate realm of DevOps processes and versioning. The clarion call for this endeavor emanates from the imperative to streamline and optimize this vital facet of software development. To illuminate the path forward, a comprehensive literature review is meticulously undertaken, both to glean insights from extant solutions and to discern the contours of unsolved quandaries that persist within this intricate arena.

In the subsequent phases of this scholarly odyssey, a bespoke methodology emerges as the central fulcrum. This strategic blueprint is meticulously crafted, hinging upon the synthesis and harmonization of established theories – notably, the edifice of formal concept analysis and the venerable Liskov Substitution Principle. The crux of this methodology envisages the crystallization of the API's description into intricate concept lattices, an ingeniously conceived framework aimed at elucidating the contours of compatibility. Concurrently, the safeguarding of adherence to the Liskov Substitution Principle assumes paramount significance, unfurling a vista wherein alterations are meticulously scrutinized to discern their compatibility quotient.

The culmination of this holistic approach is a rigorous evaluation, manifested through an empirical analysis of a case study conducted within controlled laboratory environs. This qualitative inquiry discerningly appraises the efficacy, resilience, and practical viability of the constructed artifact. As the curtain falls on this endeavor, its resonating impact reverberates across the expansive landscape of microservices evolution. The research unfurls a pioneering formal methodology, one poised to transcend the realms of academia and seamlessly integrate into the crucible of industrial exigencies. With an overarching commitment to fortify and expedite DevOps processes, this research is imbued with the potential to reshape and optimize the trajectory of microservices evolution within contemporary software engineering paradigms.

The work itself consists of the introductory part, which presents the relevance of the problem; the review and setting part, which demonstrates the thorough review of the existing research on the problem and formulation of the problem. Also, the work includes software implementation for the analysis of applying changes and visualization of the system's evolution, as well as guidelines for a proper usage of the proposed method.

References Références Referencias

1. Andrikopoulos, V., Benbernou, S., & Papazoglou, M. P. On the evolution of services//IEEE Transactions on Software Engineering, 2011, Vol. 38, Issue 3, pp. 609-628.
2. Bernhard Ganter, Rudolf Wille. Formal Concept Analysis: Mathematical Foundations. Springer Berlin, Heidelberg, 1999. 284 pp.
3. Building Microservices. by Sam Newman. Released February 2015. Publisher(s): O'Reilly Media, Inc. ISBN: 9781491950357.
4. Fish, J. (2021). Preface: Revolutionizing Engineering Practice Bymultiscale Methods. International Journal for Multiscale Computational Engineering.
5. He, X., Tu, Z., Liu, L., Xu, X., & Wang, Z. Optimal evolution planning and execution for multi-version coexisting microservice systems//18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14–17, 2020, pp. 3-18.
6. Jonkman, J. M., Wright, A. D., Hayman, G., & Robertson, A. N. (2018). Full-System Linearization for Floating Offshore Wind Turbines in OpenFAST: Preprint.
7. Liskov, B. H., & Wing, J. M. Behavioral subtyping using invariants and constraints//Formal methods for distributed processing: a survey of object-oriented approaches, 2001, pp. 254-280.

8. Taran T. A., Gatin A. R., Ushinkina E. S. Study of the Cognitive Space of the Personality//4th International School-Seminar on Artificial Intelligence for Students and Postgraduates (Braslav School - 2000)/ Sat. scientific _ works. - Minsk: BSUIR, 2000.
9. Wille R. Lattices in data analysis: how to draw them with computer//Algorithms and Order (I. Rival, ed.) Kluwer , Dordrecht -Boston. 1989.
10. X. He, Z. Tu, L. Liu, X. Xu, Z. Wang. "Optimal evolution planning and execution for multi-version coexisting microservice systems," In International Conference on Service-Oriented Computing, vol. 3, pp. 3–18, 2020 Springer, Cham.
11. Yu, S., & Gang, Z. (2020). The Construction of Resource Discovery System Platform Based on Microservices Architecture.