RESEARCH ARTICLE

WILEY

Microservices migration patterns

Armin Balalaie[1] | Abbas Heydarnoori[1] | Pooyan Jamshidi[2] | Damian A. Tamburri[3] | Theo Lynn[4]

[1]Department of Computer Engineering, Sharif University of Technology, Tehran, Iran

[2]Computer Science and Engineering Department, University of South Carolina, Columbia, South Carolina, USA

[3]Department of Electronics, Information Bioengineering (DEIB), Politecnico di Milano, Milan, Italy

[4]Irish Center for Cloud Computing and Commerce (IC[4]), Dublin, Ireland

Correspondence

Abbas Heydarnoori, Sharif University of Technology, Department of Computer Engineering, Azadi Avenue, Tehran 11365-11155, Iran.
Email: heydarnoori@sharif.edu

Present Address

A. Balalaie is currently a PhD Student in the Department of Informatics at the University of California, Irvine.

Summary

Microservices architectures are becoming the defacto standard for building continuously deployed systems. At the same time, there is a substantial growth in the demand for migrating on-premise legacy applications to the cloud. In this context, organizations tend to migrate their traditional architectures into cloud-native architectures using microservices. This article reports a set of migration and rearchitecting design patterns that we have empirically identified and collected from industrial-scale software migration projects. These migration patterns can help information technology organizations plan their migration projects toward microservices more efficiently and effectively. In addition, the proposed patterns facilitate the definition of migration plans by pattern composition. Qualitative empirical research is used to evaluate the validity of the proposed patterns. Our findings suggest that the proposed patterns are evident in other architectural refactoring and migration projects and strong candidates for effective patterns in system migrations.

KEYWORDS

cloud-native architectures, cloud computing, microservices, migration patterns

1 | INTRODUCTION

Microservices architecture is a cloud-native architecture[1] through which a software system can be realized as a set of small services. Each of these services are capable of being deployed independently on a different platform and run in their own process while communicating through lightweight mechanisms such as RESTful APIs.[2] In this setting, each service is a business capability, which can use various programming languages and data stores.[2]

Migrating current on-premise software systems to microservices introduces many benefits including, but not limited to, the ability to have differentiated availability and scalability considerations for different parts of the system, the ability to utilize different technologies and avoid technology lock-in, reduced time-to-market, and better comprehensibility of the code base.[3,4] Furthermore, the migrated system can make use of the elasticity and better payment models (pay-as-you-go) of the cloud environment, therefore, provide a better user experience for its end users.[3,4] Although the microservices architecture present many benefits, it introduces distribution complexities to the system for which new supporting components are needed such as a service registry. The decomposition of a system into small services, monitoring, and managing these services are among the other important factors that need to be considered. Therefore, migrating to the cloud, and in particular, migrating through cloud-native architectures, like microservices, is a multidimensional problem.[4-6] In the absence of a well-thought methodology, migration can be a trial-and-error endeavor, which not only can waste a lot of time but also may lead to a suboptimal solution.[5] Furthermore, given the wide variety of factors (including differing requirements

and the skills of team members) in different companies and scenarios, a unique and rigid methodology would not be adequate. Thus, instead of a one-size-fits-all methodology, a *situational method engineering* (SME)[7] approach is needed.

The first step toward the SME approach is to populate a method base or pattern repository with reusable process patterns or methodological steps, which we designate as *migration patterns* for the purpose of this paper. It is important to note that each of these migration patterns should conform to a predefined metamodel. To this end, using our previous experience in SME[8] in defining migration patterns and similar practices in the state of the art of microservices as migration patterns,[9] we generalize our experience in migration to microservices[3,4] as reusable migration patterns. These patterns are only focused on the *migration planning* phase of the migration process. We enrich each step in the migration process with the precise definition of the corresponding situation, the problem to be solved, and the possible challenges of a proposed solution. After that, we transfer them to a pattern template whose parts have a one-to-one correspondence with the metamodel elements. Parts of these patterns describe exactly the need for supporting components (eg, a service registry) in our architecture and the requirements for their introduction. Additionally, we provide some solutions and advice for the decomposition of a monolithic system into a set of services, and defining the current and the target architectures of the system as a road map for migration planning. We also provide insights including the containerization of services and their deployment into a cloud cluster.

Having a suitable pattern repository, migration engineers can select required patterns by employing the selection guidelines provided in this paper and construct a migration plan by composing the selected patterns. The migration plan can then be executed to transform the current architecture into a microservices-enabled architecture. The focus of this paper is to introduce a catalog of migration patterns and a strategy for composing them to construct a migration plan. The execution of the migration plan, however, is out of the scope of this paper. Moreover, we are assuming that the decision to migrate to the microservices architecture has been made before using these patterns. The patterns would not provide any advice regarding making a decision to migrate.

In evaluating the proposed patterns by means of empirical case study research, we observed that they are indeed present in all of the three large high-edge industrial migration projects we came in close contact with. Indeed, according to our data, almost 85% of our patterns are present in all of the three projects, which were part of our case study campaign. Finally, by cross-referencing this evaluation with extant ethnographical research, our findings suggest that the proposed patterns are in fact effective in migrating legacy solutions toward the microservices architecture.

With respect to the state of the art in pattern-based techniques that could be used in software cloud migration contexts, eg, the seminal works,[10-14] we provide the following concrete contributions: (1) a pattern repository made of 14 recurring cloud refactoring patterns, (2) a set of reusable methodological steps consistent with a *compositional* migration approach toward cloud-native microservice migration scenarios, (3) an empirically defined migration pattern metamodel that allows quick extensibility of contributions "a." and "b." with further practice, and (4) a rigorous qualitative empirical evaluation of the patterns in three real-life industrial scenarios.

The rest of this paper is structured as follows. In Section 2, we discuss the overall process used to identify and to extract the patterns. Section 3 elaborates the identified migration patterns in detail. Section 4 explains the procedures of migration pattern selection, composition, and extension. The correctness of the identified patterns is evaluated in Section 5. We review the related work in Section 6. Finally, we conclude this paper and provide future research directions in Section 7.

2 | PATTERNS IDENTIFICATION AND EXTRACTION PROCESS

Situational method engineering is an alternative approach for developing software, which involves all of the phases of the development process, and includes the use and adaptation of a method based on local conditions. In other words, a method can be engineered based on a specific situation.[7]

The first step toward the SME approach is to create a pattern repository consisting of reusable process patterns or method chunks, which we call *migration patterns*. The patterns in this repository should conform to a predefined metamodel and should be documented using a pattern specification template. These patterns can be abstracted away from either existing methods or experience reports.[7] At the time of writing this paper, extant methodologies for migrating to microservices could not be identified. As such, based on our previous experience with SME[8] and defining migration patterns,[9] we decided to generalize our previous experience in migrating to the microservices architecture[3,4] into migration patterns.

To this end, a customized metamodel and a specification template for our migration patterns is needed. This is discussed further in the following subsections. However, the most important point in extracting the patterns is to determine their granularity. Keeping the system in a stable state after applying a pattern and performing one architectural change at a
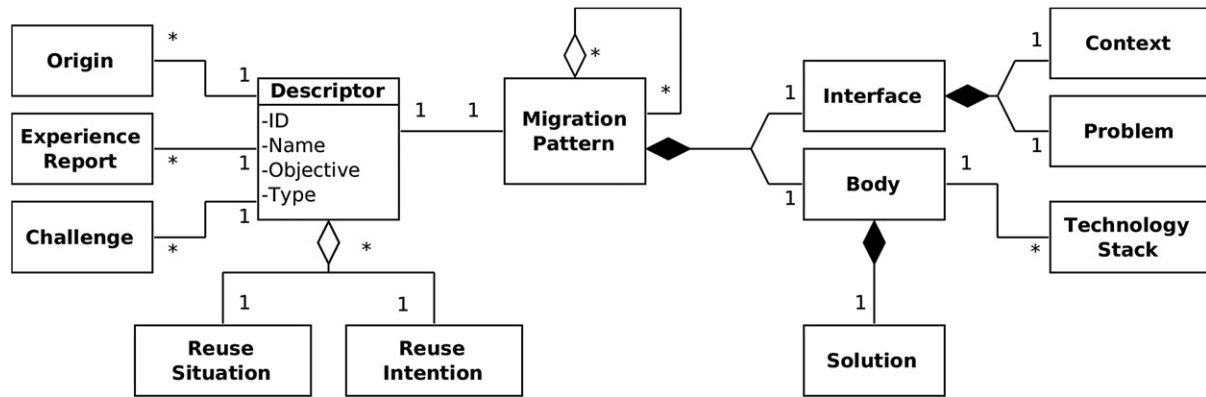
FIGURE 1 Migration patterns metamodel

time are deemed the most important factors in determining the granularity of the patterns. These are the reasons that some of the steps in our original experience report (ie, the work of Balalaie et al[3]) have been abstracted away as multiple migration patterns.

2.1 | Migration patterns metamodel

In order for migration patterns to be retrieved effectively from the repository and get reused, they should adhere to a metamodel. The presence of a metamodel can help to expand the repository by providing means for defining new patterns in a standard manner, ie, instantiating the metamodel by populating the pattern template. To this end, we adapted the metamodel proposed in the work of Henderson-Sellers et al[15] (see Figure 1) for SME. In this metamodel, each method chunk consists of the following parts.

- A *descriptor*, which is used for pattern selection and contains the *name*, *ID*, *objective*, and *type* of the method chunk. It explains the situation where the method chunk can be reused and the intent behind its reuse.
- A *body*, which has a *process* part describing the solution suggested by the method chunk and a *product* part outlining the artifacts, if any, that should be created upon the execution of the process part. More specifically, this part describes the architecture before and after the migration. This obviously includes the architectural changes to the source architecture.
- An *interface*, which provides information on the situation for which the method chunk is suitable and its high-level intention.

In the work of Mirbel and Ralyté,[16] a project development knowledge frame, namely, the *reuse frame*, has been proposed to populate values for the *reuse situation*. The proposed reuse frame is too general for migrating to microservices because it covers the general aspects of the traditional software development process. However, it does not concern itself with architecture-based development and operational parts of large scale software systems, which are very important in the microservices context. Consequently, we propose a set of architectural and operational factors, which can be used to specify the migration patterns' reuse situation. In Table 1, we briefly describe these factors. Additionally, for *reuse intention*, we use the linguistic approach proposed by Prat,[17] which uses a clause with a verb and a target for expressing the intention.

In order to make it more suitable for our migration patterns, we also made some minor changes to the metamodel. First, we changed the name of the *method chunk* class to *migration pattern*. Second, the *situation* class is renamed to *context* in the pattern's interface. Third, the *intention* class is replaced by *problem* since it reflects the intention of the pattern in a question form. Fourth, the *process part* and *product part* classes are joined to form the *solution* class. Fifth, the *objective* class is removed as it overlaps with the *reuse intention* class. Sixth, we enriched the descriptor by adding zero or more *challenges* to it. As each migration pattern causes the system to undergo some changes and each change can introduce some side effects or challenges, we decided to put these challenges as a determining factor in the pattern selection. Seventh, since during the pattern selection and in some steps, alternative patterns can be selected, which have different trade-offs, we added zero or more similar patterns to the descriptor. It is useful in knowing alternative paths for migration. Eighth, some of the migration patterns, which have existing implementations, require new supporting components or tools for their realization. As a result, we added the *technology stack* class to the body of the migration pattern to provide illustrative tools and technologies that can be employed to realize that particular pattern.

TABLE 1 Migration pattern selection factors

Factor	Description
	Architectural Factors
Scalability	Increasing the scalability of an application by scaling out its services
High Availability	Increasing the availability of an application by replicating its services
Fault Tolerance	Decreasing the chance of failure in an application and providing means for handling the failures effectively
Modifiability	Increasing the ability to change an application with the least side effects and without affecting its end users
Polyglotness	Enabling an application to use different programming languages and data stores
Decomposition	Rearchitecting an application to a set of services
Understanding	Perceiving the current situation of an application
Visioning	Deciding on the final situation of an application after the migration
	Operational Factors
Dynamicity	Enabling an application to change in runtime without affecting its end users
Resource efficiency	Decreasing the amount of resources needed for an application's deployment
Deployment	Facilitating an application's deployment process and removing deployment anomalies
Monitoring	Enabling an application to be monitored in runtime effectively

2.2 | Patterns specification template

As will be discussed later in Section 3, each migration pattern is documented in a pattern template. The pattern's title is a combination of its ID and name joined by a "-." The pattern's type can be either *atomic* or *aggregate*. It should be noted that currently the repository only contains atomic patterns; however, for extensibility purposes, it should be also possible to define aggregate patterns in the future. As can be seen in Section 3, part names in the pattern template use those in the metamodel.

3 | MIGRATION AND REARCHITECTING PATTERNS

Migrating to microservices provides a software system with many benefits including, but not limited to, being able to have differentiated availability and scalability provisioning for different parts of the system, the ability to make use of different technologies and avoid technology lock-in, reduced time-to-market, and higher comprehensibility of the code base.[3,4] Furthermore, the migrated system can benefit from the elasticity and better payment models (pay-as-you-go) of the cloud environment and therefore provide a better user experience for its end users.[3,4] Despite the fact that microservices architecture presents many benefits, it adds distribution complexities to the system, which requires new supporting components, eg, a service registry. Difficulties like the decomposition of a system into small services, and monitoring and managing these services are among the other important factors that make migrating to microservices a nontrivial task. In this section, we discuss the patterns, which help to address these challenges.

Parts of these patterns describe exactly the need for supporting components, such as a service registry, in our architecture and the requirements for their introduction. Additionally, we provide some solutions and guidance for the decomposition of a monolithic system into a set of services and defining the current and the target architectures of the system as a road map for migration planning. We also provide insights on the containerization of services and their deployment in a cluster. As noted before, these migration patterns can help in planning the architectural refactorings when migrating legacy software systems to microservices and can facilitate the migration plan definition through pattern composition.

3.1 | MP₁: Enable continuous integration (Figure 2)

Reuse Intention: Build the *continuous integration*[18] pipeline.
Reuse Situation: Deployment.
Context: There is a working monolithic software system, and the team responsible for maintaining the system has decided to migrate this system to microservices.
Problem: Considering that by adopting microservices, the number of services will be increased, how can we always have available production-ready artifacts? How can the system be prepared for introducing continuous delivery?
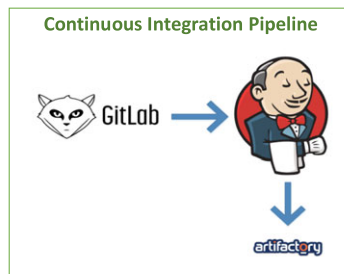
FIGURE 2 MP$_1$: Enable the continuous integration [Colour figure can be viewed at wileyonlinelibrary.com]

Solution: The first step toward *continuous delivery*[18] is to set up continuous integration.[18] Continuous integration automates the build and test processes and ensures the availability of production-ready artifacts. Normally, a continuous integration pipeline contains a code repository, an artifact repository, and a continuous integration server. First, each service should be placed in a separate repository, which enables a clearer history and separates the build life cycle of each service. Then, a continuous integration job should be created for each service. Each time a service's code repository changes, the job should be triggered. The job's responsibility includes fetching the new code from the repository, running the tests against the new code, building the corresponding artifacts, and pushing these artifacts to the artifact repository. Failure to execute each of these steps should terminate the job from proceeding and informs the corresponding development team of the occurred errors. This team should not do anything else until addressing the reported errors. One simple rule in continuous integration is that new changes should not break the system's stability and should pass all of the predefined tests.

Technology Stack: Gitlab, Artifactory, Nexus, Jenkins, GoCD, Travis, Bamboo, and Teamcity.

3.2 | MP$_2$: Recover the current architecture (Figure 3)

Reuse Intention: Create the migration plan's initial state.

Reuse Situation: Understanding.

Context: There is a working monolithic software system, and the team responsible for maintaining the system has decided to migrate this system to microservices. In order to plan the migration, the team needs to know the current system architecture which either does not exist or is rather obsolete.

Problem: What is the big picture of the system? What high-level information is needed for planning the migration to microservices?

Solution: With software architecture, people tend to imagine a bunch of formal diagrams. In practice, a helpful software architecture is a set of important things in the system communicated through some textual or visual artifacts. It is a good practice to keep these artifacts as simple as possible, thereby everyone can understand them easily using basic guidance. The following items are important in planning for migration to microservices.

- *Component and Service Architecture*: Using these two types of architecture together is not accidental. In practice, it is more useful to illustrate these architectures in one informal visual artifact so that everyone can easily grasp the inner structure of the system quickly. Indicating the direction of service calls is important as it could clearly separate service
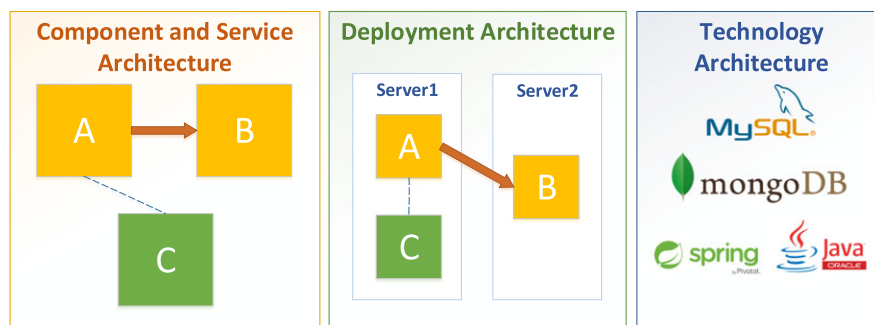

FIGURE 3 MP$_2$: Recover the current architecture [Colour figure can be viewed at wileyonlinelibrary.com]
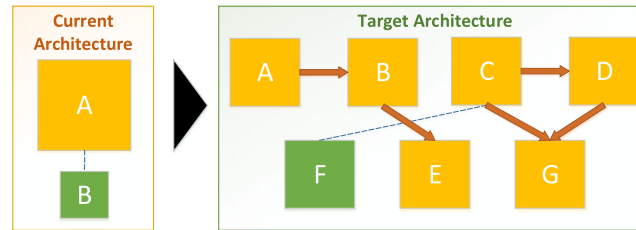
FIGURE 4 MP$_3$ and MP$_4$: Decompose the monolith [Colour figure can be viewed at wileyonlinelibrary.com]

providers and service consumers. Furthermore, it can provide some insights in to the dynamics of the system. Try to understand the inner domain of each component by considering their entities and their overall business logic. Discuss these items with the developers responsible for that specific component to identify additional details. Perceiving the domain of the system is important since it is required for the decomposition of the system to small services.

- *Technology Architecture*: Understanding the current technology stack is important as it could help the team to identify existing libraries that could facilitate the migration. These new libraries include those that can be used as supporting components in microservices that work well with the current technology stack, eg, a service registry for Java, and ones that facilitate nonarchitectural migration aspects, eg, data migration tools. Enumerate any programming languages, database technologies, middleware technologies, and third-party libraries that have been used in the project.
- *Deployment Architecture and Procedure*: Microservices have a close relationship with continuous delivery, which can be considered a radical change in software deployment. Understanding the current deployment architecture and procedure will help the team to gradually move toward continuous delivery.

Note that it is not necessary to document every detail as they will change in a near future. Try to involve developers and operations teams in a process of understanding the system architecture since it is more time efficient and they are needed for the rest of the migration. In this way, a common understanding of the system can be established. This information will be consolidated in the team members′ minds and communicated orally, which is far better than an unread architecture document. Furthermore, a good atmosphere for collaborating between developers and operations can be established, which is a good start for *DevOps*.[4] Put these artifacts in a place, eg, in a Git repository, so that everyone in the team can see them. Be open to the comments since they may reveal important details about the system.

3.3 | MP$_3$: Decompose the monolith (Figure 4)

Reuse Intention: Rearchitect a monolithic software system to a set of services using a *domain-driven design* (DDD).[19]
Reuse Situation: Polyglotness, decomposition, and modifiability.
Context: There is a monolithic software system with a complex domain, which has at least one of the following attributes.

- Due to the complexity of the system, comprehensibility of the source code is low.
- Different parts of the system have different nonfunctional requirements (eg, scalability).
- Every change in the system needs a whole redeployment, and all parts of the system are not equal in terms of change frequency.

Problem: How can the system be decomposed into smaller chunks? How big should these chunks (components) be?
Solution: Domain-driven design should be used to identify subdomains of the business that the system is operating in. Then, each subdomain can constitute a *bounded context* (BC) that is a deployable unit. The one-to-one correspondence between subdomains and BCs happens in *greenfield* projects.[†] In contrast, due to the existing limitations in legacy systems, it is not always possible. Nevertheless, since the candidate system is going to benefit from microservices, it is strongly suggested that the responsible team plan to migrate the system like it is a greenfield project. This type of planning may introduce many changes to the system, but it may also be the optimal solution. Executing the plan incrementally is another concern; however, a satisfactory endpoint for the migration is needed upfront so that all stakeholders including business decision makers can see the advantages.

[†]A greenfield project is one that lacks constraints imposed by existing work.

Domain-driven design is a good option for the initial decomposition of a system. Further decomposition can happen as a result of either different change frequency rate or different nonfunctional requirements for different parts of a BC. Additionally, DDD can be applied on a subdomain to decompose it to smaller chunks.

The size of the BCs is difficult to recommend or suggest. It totally depends on the system's requirements. Initially, they can be as large as the corresponding domain. As time goes on, requirements will change, and the previous boundaries or BC sizes may not be appropriate anymore. It is recommended to start with a low number of services, two or three, and incrementally add more services as the system grows and the team understands microservices and the system's requirements better.

Challenges: A bad system decomposition can lead to performance penalties. For instance, if the decomposition creates chatty services, this can lead to performance degradation.

Similar Patterns: MP_4 and MP_5.

- MP_4: Decompose the monolith based on data ownership.
- MP_5: Change code dependency to service call.

3.4 | MP_4: Decompose the monolith based on data ownership (Figure 4)

Reuse Intention: Rearchitect a monolithic software system to a set of services using a *data ownership*.
Reuse Situation: Polyglotness, decomposition, and modifiability.
Context: There is a monolithic software system with a noncomplex domain, which has at least one of the following attributes.

- Due to the complexity of the system, comprehensibility of the source code is low.
- Different parts of the system have different nonfunctional requirements (eg, scalability).
- Every change in the system results in a whole redeployment, and all parts of the system are not equal in terms of change frequency.

Problem: How can the system be decomposed into smaller chunks? How big should these chunks (components) be?
Solution: Decompose the system based on the ownership of data. Find different cohesive sets of data entities that can be grouped together as a unit and can have a unique owner. Package each group and their corresponding business logic into a service. Each entity can be modified or created just through its owner (its corresponding service). Other services can have a copy of an entity that they do not own, but care should be taken about staleness, and their copies should be synchronized appropriately. Further decomposition can happen as a result of either different change frequency rate or different nonfunctional requirements for different parts of a service.

The size of the services is difficult to recommend or suggest. It totally depends on the entities that exist in the system.
Challenges: This pattern is suitable for when the domain of the system is not complex and the data entities can be grouped easily. In a large domain that has multiple subdomains, applying this pattern can be confusing and time consuming and can even lead to an inappropriate decomposition.
Similar Patterns: MP_3 and MP_5.

- MP_3: Decompose the monolith.
- MP_5: Change code dependency to service call.

3.5 | MP_5: Change code dependency to service call (Figure 5)

Reuse Intention: Transform code-level dependency to service-level dependency.
Reuse Situation: Decomposition and modifiability.
Context: A software system has been decomposed to a set of small services to use a microservices architectural style. There is a component in the system that is acting as a dependency to other services or components.
Problem: When is it appropriate to change this code-level dependency to a service-level dependency and when is it not?
Solution: Try to keep the services' code as separate as possible. When services share code as a dependency, there is a high chance that a service developer will fail the build process of another service by changing the shared code. In some cases, such as a code that is shared as a common library, which rarely changes, eg, a string manipulation library, it is reasonable to share code. However, sharing internal entities or interface schema should be prohibited since changing them would
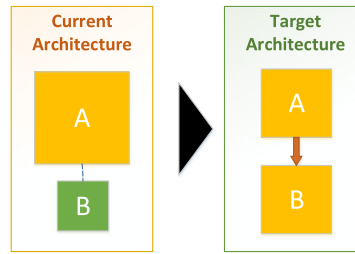
FIGURE 5 MP$_5$: Change code dependency to service call [Colour figure can be viewed at wileyonlinelibrary.com]
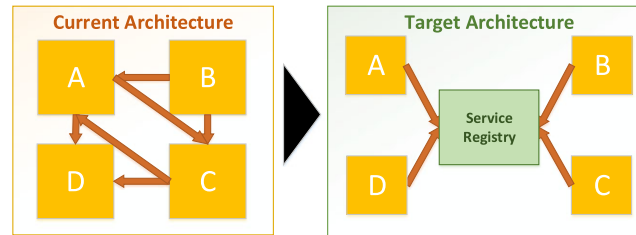

FIGURE 6 MP$_6$: Introduce service registry [Colour figure can be viewed at wileyonlinelibrary.com]

force other dependent services to change immediately even though they may want to change gradually. In cases where sharing is not a good choice, it is a good practice to share that piece of functionality as a service. This service could be either a completely isolated service or a part of one of the dependent services. Different scalability needs between the shared library and the dependent services could also result in changing a shared library to a service. In this case, by separating them in different services, they can be scaled independently.

Challenges: Sometimes separating libraries from their dependents and using a service call instead of a method call may introduce performance issues. Although performance issues can be handled using careful caching mechanisms, it adds another layer of complexity to the dependent service.

Similar Patterns: MP$_3$ and MP$_4$.

- MP$_3$: Decompose the monolith.
- MP$_4$: Decompose the monolith based on data ownership.

3.6 | MP$_6$: Introduce service registry (Figure 6)

Reuse Intention: Introduce the *dynamic location* of services' instances using a *service registry*.

Reuse Situation: Scalability, high availability, dynamicity, and deployment.

Context: A software system has been decomposed to a set of small services to use a microservices architectural style, and each of these services has one or more instances deployed in the production environment. The number of instances can be changed dynamically, and each of them can be deployed in different systems.

Problem: How can services locate each other dynamically? How can an edge server or a load balancer know the list of instances of a service to which it can route the traffic?

Solution: Setup a service registry, which stores service instances' addresses. Each service registers itself during the initiation. The removal of an instance from the registry can be triggered through either not receiving a periodic heartbeat from the instance or by the instance itself during its termination. Having a list of available services' instances, an edge server, a load balancer, or other services can locate their desired services dynamically through the service registry.

Service registry, when introduced, is a vital component of the system since the communication between different parts of the system depends on its availability. Thus, replication strategies should be leveraged as a high availability mechanism.

Challenges: The rest of the system is coupled to this component for the communications among each other. Thus, service registry, if not correctly implemented, could become a single point of failure.

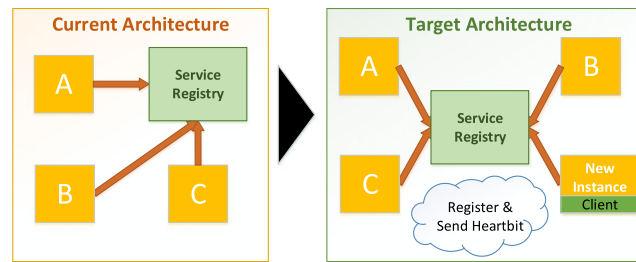**Technology Stack:** Eureka, Consul, Apache Zookeeper, and etcd.

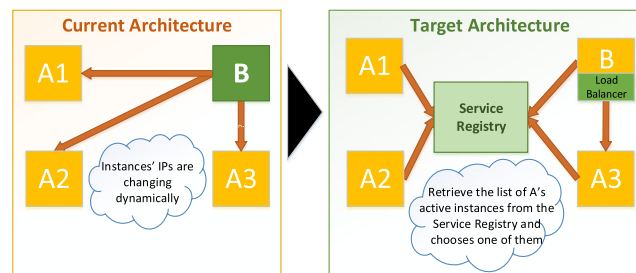**FIGURE 7** MP$_7$: Introduce service registry client [Colour figure can be viewed at wileyonlinelibrary.com]

FIGURE 8 MP$_8$: Introduce internal load balancer. IP, internet protocol [Colour figure can be viewed at wileyonlinelibrary.com]

3.7 | MP$_7$: Introduce service registry client (Figure 7)

Reuse Intention: Facilitate dynamic location of services' instances using a *service registry client*.

Reuse Situation: Scalability, high availability, dynamicity, and deployment.

Context: A software system has been decomposed to a set of small services to use a microservices architectural style, and a service registry has been set up. The number of instances of each service can be changed dynamically, and each of them can be deployed in different systems.

Problem: How does the service registry know a new instance has been deployed? How can it know an instance has been terminated?

Solution: Each service should know the address of the service registry and register itself during its initiation. Then, a periodic heartbeat should be sent from each instance toward the registry to keep the instance in the available instances list. Instance removal from the service registry can be done through either not sending the heartbeat anymore (ie, the failure scenario) or explicitly informing the registry of the instance termination.

Challenges: The drawback is that the client should be implemented for all of the programming languages in use and its bad implementation can complicate the service code.

Technology Stack: Eureka is a service registry, which has a Java client implementation for its server version.

3.8 | MP$_8$: Introduce internal load balancer (Figure 8)

Reuse Intention: Introduce load balancing between instances of a service using an *internal load balancer*.

Reuse Situation: Scalability, high availability, and dynamicity.

Context: A software system has been decomposed to a set of small services to use a microservices architectural style, and a service registry has been set up. In the production environment, numerous instances of each service exist. Each service can be a client of the rest of the services in the system.

Problem: How can the load on a service between its instances be balanced based on the conditions of the client? How can the load on a service be balanced without setting up an external load balancer?

Solution: Each service, as a client, should have an internal load balancer, which fetches the list of available instances of a desired service from the service registry, eg, through a service registry client. Then, this internal load balancer can balance the load between the available instances using local metrics, eg, the response time of the instances.

An internal load balancer removes the burden of setting up an external load balancer and brings in the possibility of having different load balancing mechanisms in different clients of a service.
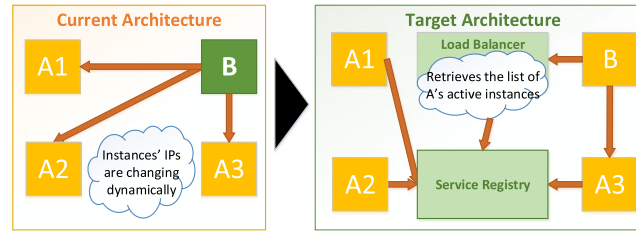
FIGURE 9 MP$_9$: Introduce external load balancer. IP, internet protocol [Colour figure can be viewed at wileyonlinelibrary.com]

Challenges: The downside is the need to create an internal load balancer for different programming languages in use which can be integrated with the service registry. Additionally, the load balancing mechanism is not centralized.
Similar Patterns: MP$_9$.

- MP$_9$: Introduce external load balancer.

Technology Stack: Ribbon is an internal load balancer for Java that works well with Eureka, a service registry tool.

3.9 | MP$_9$: Introduce external load balancer (Figure 9)

Reuse Intention: Introduce load balancing between instances of a service using an *external load balancer*.
Reuse Situation: Scalability, high availability, and dynamicity.
Context: A software system has been decomposed to a set of small services to use a microservices architectural style, and a service registry has been set up. In the production environment, numerous instances of each service exist. Each service can be a client of the rest of the services in the system.
Problem: How can the load on a service between its instances be balanced with the least changes in the service's code? How can a centralized load balancing approach be used for all of the services?
Solution: An external load balancer can be set up, which retrieves the list of available instances from the service registry and uses a centralized algorithm for balancing the load between instances of a service. This load balancer can act either as a proxy (not recommended) or as an instance address locator. Another solution is to choose a service registry that has a built-in support for load balancing and can act as a load balanced address locator.
Challenges: The downside is that local metrics, eg, the response time of the instances, cannot be used to improve the load balancing. Furthermore, different clients cannot have different load balancing strategies. Additionally, a highly available load balancing cluster is needed when the load balancer acts as a proxy.
Similar Patterns: MP$_8$.

- MP$_8$: Introduce internal load balancer.

Technology Stack: Amazon Elastic Load Balancing, Nginx, HAProxy, and Eureka.

3.10 | MP$_{10}$: Introduce circuit breaker (Figure 10)

Reuse Intention: Introduce fault tolerance in interservice communications using a *circuit breaker*.[20]
Reuse Situation: Fault tolerance and high availability.
Context: A software system has been decomposed to a set of small services to use a microservices architectural style. Some of the end user requests need interservice communications in the internal system.
Problem: How can a system fail fast and not wait until reaching a service call timeout when calling a recently unavailable service? How can the system be more resilient when an unavailable service is called?
Solution: The service consumer can use a circuit breaker when calling the service provider. When a service provider is available, this component would not do anything (the *close circuit* state). It monitors the recent responses from the service provider and will act appropriately when the number of failure responses passes a predefined threshold (the *open circuit* state). The corresponding action could be either returning a meaningful response code or returning the latest cached data from the service provider (if it is acceptable for that specific response). After a specific timeout, in order to check the service provider's availability, the component will try to access the service provider again (the *half-open circuit* state),
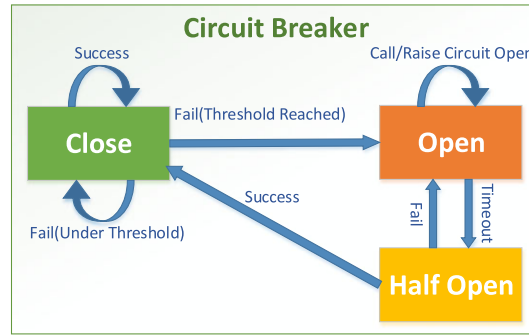
FIGURE 10 MP$_{10}$: Introduce circuit breaker [Colour figure can be viewed at wileyonlinelibrary.com]
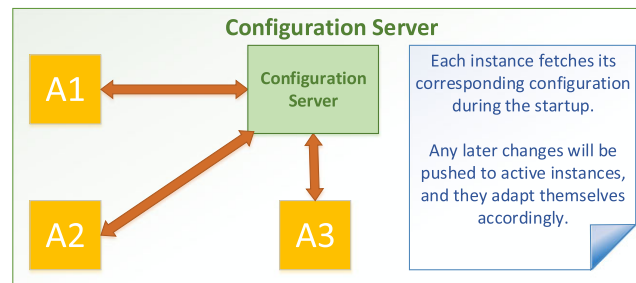
FIGURE 11 MP$_{11}$: Introduce configuration server [Colour figure can be viewed at wileyonlinelibrary.com]

and when there is a successful attempt, the state will be changed to a close circuit state. Otherwise, the state will be modified to an open circuit state.

Challenges: Recognizing the appropriate response in the open circuit state can be challenging and should be coordinated with the business stakeholders. Furthermore, if the response in open circuit state is not just an exception, the service provider team should certify the possibility of returning that response on their behalf.

Technology Stack: Hystrix.

3.11 | MP$_{11}$: Introduce configuration server (Figure 11)

Reuse Intention: Change a system's configuration at runtime using a *configuration server*.

Reuse Situation: Modifiability, dynamicity, and deployment

Context: A software system has been decomposed to a set of small services to use a microservices architectural style. Each service has numerous instances running in the production environment. The list of available instances is accessible through a service registry.

Problem: How can the running instances' configurations be changed without redeploying them?

Solution: There should be two separate repositories for storing the source code and the software configurations. Although there may be a need to synchronize these repositories when some changes happen in the configuration keys, they should evolve independently of each other. Furthermore, any changes to the configuration repository should be propagated to the corresponding running instances and they should adapt themselves accordingly.

Challenges: The configuration propagation endpoints in the services and the adapting strategy need to be implemented for all of the programming languages in use.

Technology Stack: Spring Configuration Server and Archaius.

3.12 | MP$_{12}$: Introduce edge server (Figure 12)

Reuse Intention: Enable dynamic rerouting of external requests to internal services using an *edge server*.

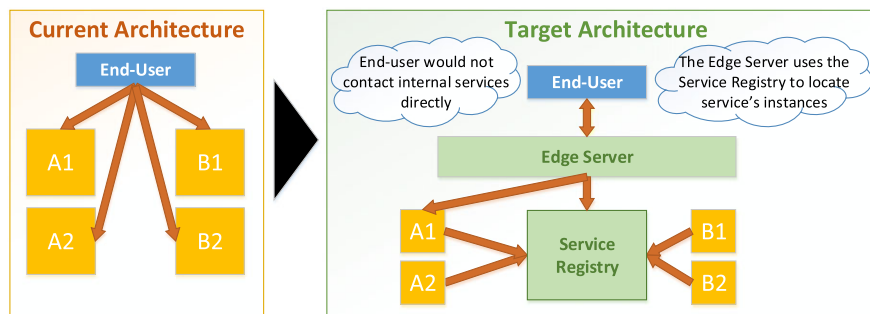**Reuse Situation:** Modifiability and dynamicity.

FIGURE 12 MP$_{12}$: Introduce edge server [Colour figure can be viewed at wileyonlinelibrary.com]
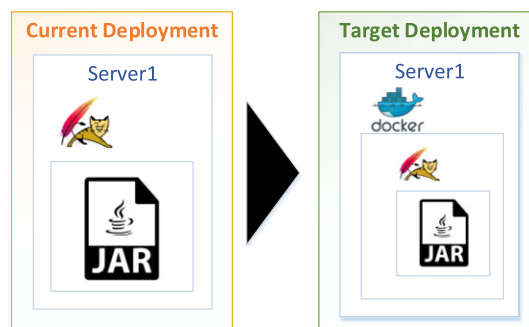

FIGURE 13 MP$_{13}$: Containerize the services [Colour figure can be viewed at wileyonlinelibrary.com]

Context: A software system has been decomposed to a set of small services to use a microservices architectural style. Due to ease of service initiation in the system, new services can be introduced easily, and existing services can be rearchitected based on new requirements.

Problem: How can the internal service structure complexity and evolution be hidden from the end users? How can the overall status and usage of a service in production be monitored?

Solution: There should be another layer of indirection in the system as a front door. This layer can do dynamic routing based on a predefined configuration. The service instances addresses for routing the incoming traffic can be either hard-coded or fetched from a service registry. End-users will depend on this layer's interface, and thus, the internal structure changes would not affect them and will be handled through new routing rules. Furthermore, since all of the traffic passes through this layer, it is the best place to monitor the overall usage of services.

Challenges: All the traffic passes through this layer. In order to remove the single point of failure, this layer should be replicated through load balancing mechanisms.

Technology Stack: Zuul.

3.13 | MP$_{13}$: Containerize the services (Figure 13)

Reuse Intention: Have the same behavior in the development and the production environment using *containerization*.

Reuse Situation: Resource efficiency and deployment.

Context: A software system has been decomposed to a set of small services to use a microservices architectural style, and a continuous integration pipeline is in place and is working. Each service needs a specific environment to run correctly, which is either set up manually or through a configuration management tool. The differences between the development and production environments can cause some problems, eg, the same code may produce different behaviors in these two environments. Therefore, the deployment of services in the production environment becomes a cumbersome task.

Problem: How can the development and the production environments produce the same results for the same code? How can the complexity of configuration management tools or difficulties of manual deployments be eliminated?

Solution: As each service may need a different environment for its deployment, a solution could be the deployment of each service in a virtual machine in isolation with their own desired environment using configuration management tools. The downside is that due to virtualization, a lot of resources are wasted for service isolation, and configuration
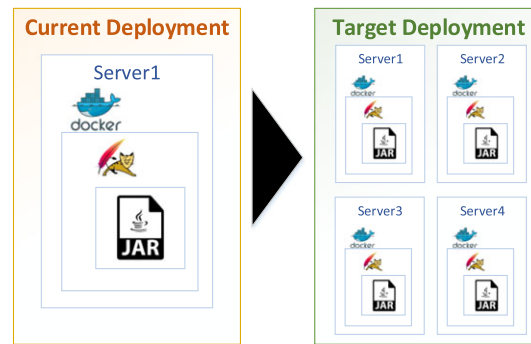
FIGURE 14 MP$_{14}$: Cluster management and container orchestration [Colour figure can be viewed at wileyonlinelibrary.com]

management is another layer of complexity in the deployment. Compared with the virtualization, containerization is more lightweight and it can remove the need for configuration management tools since there are a lot of ready images in the central repositories containing different applications, and any further configuration can be done in the new images building stage.

For this solution, add another step to the continuous integration pipeline to build container images and store the images in a private image repository. These images can then be run in both of the production and development environments that produce the same behavior. Each service should have its own container image creation configuration; the script for running any other required services' containers should reside inside its code repository.

It is a good practice to add environment variables as a high priority source for populating the software configuration. In this way, the configuration keys that can have different values in different environments, eg, database URL or any credentials, and they can be injected easily in the container creation phase. Having a list of required environment variables for running a service in its code repository is a good practice as it makes stakeholders aware of these changing variables.

Challenges: The trade-off for a lightweight, isolated and reproducible environment through containerization is potentially a greater computational overhead in comparison to deploying directly to an operating system. Furthermore, the development environment should be adapted to embrace containers.

Technology Stack: Docker.

3.14 | MP$_{14}$: Deploy into a cluster and orchestrate containers (Figure 14)

Reuse Intention: Deploy service instances' container images in a cluster by employing cluster management tools.

Reuse Situation: Resource efficiency and deployment.

Context: A software system has been architected based on a microservices architectural style, and a continuous integration pipeline is in place and is working. Therefore, a production-ready container image is available for the deployment of each service. A large number of services exist, and thereby, their deployment and redeployment are complex, cumbersome, and unmanageable.

Problem: How can a service's instances be deployed into a cluster? How can the deployment and redeployment of all of the services be orchestrated with the least effort?

Solution: There should exist a system that can manage a cluster of computing nodes. This management system should be able to deploy the services' container images on demand, with specified number of instances and on different nodes. Additionally, it should handle the failure of instances and restart the failed nodes or instances in such a case. It should also provide the means for auto-scaling the services. Having an internal name resolution strategy could be a good feature as some services, such as service registry, should be identified using an internal name instead of an IP address.

In order to effectively orchestrate the deployment process, the cluster management tool should provide the means to define the deployment architecture of services declaratively. With a declarative deployment architecture, any additional effort for deployment, eg, auto-scaling and failure management of the deployed services, should be delegated to the cluster management tool.

Challenges: Since the management tool will handle the deployment and other important tasks such as failure resilience and auto-scaling, its failure can stop the deployed software system from working. Thus, it should be deployed in a highly available manner and with no single point of failure.

Technology Stack: Mesos + Marathon and Kubernetes.

3.15 | MP₁₅: Monitor the system and provide feedback

Reuse Intention: Monitor the running services' instances and provide feedback to the development team.

Reuse Situation: Monitoring, modifiability, and deployment.

Context: A software system has been architected based on a microservices architectural style. The whole system is being run on a cluster of containers with each service having a number of instances in production.

Problem: How can the underlying infrastructure be monitored? How can the gathered data be used to rearchitect the system by providing feedbacks to the development team?

Solution: Each service should have its independent monitoring facility owned by the operational part of the service's team. These facilities should include the required components to gather the monitoring information, eg, the CPU and RAM usages, and send them to a monitoring server. Therefore, the following components should be added to each service container's image and be configured, either during the creation of the container images, or the creation of actual containers. In the monitoring server, this information should be parsed and aggregated into structured information and stored somewhere that can be queried efficiently such as in an indexing server or a time series database. Having a pool of timely monitoring information, a visualization tool can be used to obtain an overall view on the status of the system. This information helps the development part of the service's team to refactor the architecture to remove performance bottlenecks or other detected anomalies.

Challenges: None.

Technology Stack: Collectd + Logstash + ElasticSearch + Kibana.

4 | SELECTION, COMPOSITION, AND EXTENSION OF MIGRATION PATTERNS

4.1 | Methodology

Having an initial set of migration patterns at hand, a method engineer can construct a new migration plan on the fly based on the current migration project. The overall process of selecting and composing migration patterns is shown in Figure 15. For this purpose, first, the method engineer should identify migration drivers and requirements. After that, she can match the identified requirements with the migration pattern selection factors, which are described in Table 1. Based on each relevant selection factor, she can access a set of migration patterns from which the most suitable patterns can be selected by considering their *descriptor* and specifically their *reuse intention* and the *related challenges* in using them. Checking the *context* for using each pattern can also be helpful. After conducting the aforementioned process, the method engineer will end up with a set of selected patterns, which can then be composed to construct a migration plan.
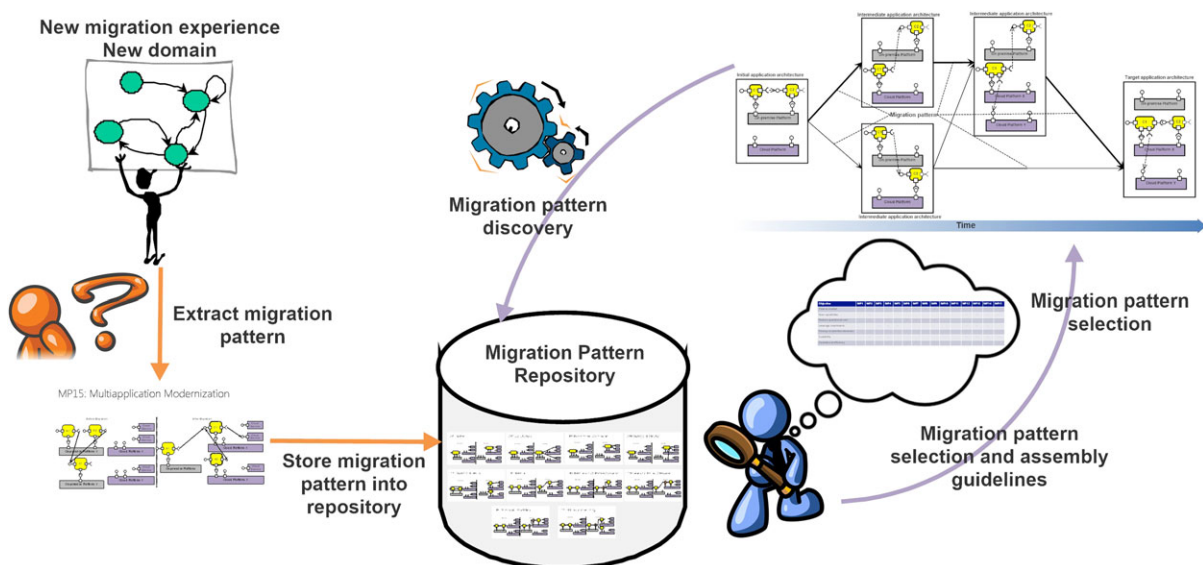

FIGURE 15 The process of selecting patterns, instantiating and composing a migration plan, and extending the repository [Colour figure can be viewed at wileyonlinelibrary.com]

Composition of the patterns can be project specific and to some extent depends on the project's priorities. However, it is reasonable to follow the following steps and diverge whenever needed inside a step. Choose the patterns that (1) do not have any dependencies. These patterns are normally operational or do not change the software system. (2) Change the software system but do not affect the end users. (3) Make internal changes transparent to the end users. (4) Solve the problems of turning a monolithic system into a distributed system (eg, fault tolerance and load balancing). (5) Any remaining decomposition or operational patterns.

In the next section, we revisit the migration of Backtory to the microservices architecture[3,4] and use it as a running example to show the applicability of the selection factors and pattern composition guidelines.

4.2 | Example: The case of migrating backtory

Backtory is a commercial mobile backend as a service that has been developed by PegahTech.[‡] The original functionality of Backtory was an relational database management system as a service. Developers could define their database schemas in the Backtory dashboard, and it would provide them with an SDK for their desired target platform, eg, Android or iOS. Afterwards, developers needed only to code in their desired platforms using their domain objects. The objects would then make service calls on their behalf to the Backtory API servers to fulfill their requests. At the time of writing this paper, new services have been added to Backtory such as authentication as a service, NoSQL as a service, gaming as a service, and so on.

As already discussed in our prior publications,[3,4] we migrated Backtory to microservices over a six-month period. The main migration drivers were reusability, decentralized data governance, automated deployment, and scalability. As no documentation on the architecture of the system were available at the start of the migration process, we needed to understand the system thoroughly. Moreover, we intended that the changes in the system to be transparent to its end users. Finally, we believed that due to the increased number of microservices and their corresponding interaction, having a fault tolerance mechanism was a must and inevitable. In the following, we revisit the migration process discussed in the works of Balalaie et al[3,4] using the patterns discussed in this article.

4.2.1 | Patterns selection

Based on Table 1, architecture-level reusability can be translated to factoring-out services, which can be reused; decentralization in data governance can be realized by decomposing the system into different services, which govern their own data; and finally, change transparency to end users equates to modifiability. Consequently, with respect to the mentioned decision-making factors in Table 1 (on Page 4), the patterns can be selected based on the following factors: *understanding*, *decomposition*, *modifiability*, *scalability*, *fault tolerance*, and *deployment*.

The need for understanding the system guides the method engineer to MP_2. This pattern is useful when there are no architecture documentation available for the system as in the case of Backtory. As such, we chose this pattern to recover the current architecture that shall be acted as the migration plan initial state.

Decomposition points to patterns MP_3, MP_4, and MP_5. MP_3 and MP_4 are important for decomposing a monolithic system into a starter set of microservices. This is particularly true where there is a need for scaling different parts of system independently, again as in the case with Backtory. However, these patterns use different methods to decompose a system. The distinguishing factor is the domain size of the system, which can be inferred from the challenges part of the patterns. Since the domain size in Backtory was not very large at the beginning, we selected the MP_4 pattern. In addition, we selected MP_5 as it could scale the DeveloperData component independently[3,4] by transforming the dependency between this component and its dependees from code to service level.

Modifiability directs the method engineer to MP_3, MP_4, MP_5, MP_{11}, MP_{12}, and MP_{15}. Only MP_{12} is related to being transparent to end users during the change. Additionally, after or during migration to the microservices architecture, there is a point that due to the common platform that has been built around microservices, it would be easy to introduce new services to the system or rearchitect the current services. This is the right context for this pattern. Furthermore, its reuse intention explicitly proposes a solution for not affecting end users during service rearchitecting. Thus, we chose it.

MP_6, MP_7, MP_8, and MP_9 are related to the scalability factor. The first two patterns were selected because, after decomposition of the system into a set of microservices, it should be possible to increase or decrease the number of running

instances for each service based on the workload. Moreover, MP_8 was selected since using local information for load balancing was preferred.

MP_{10} was employed to satisfy the need for fault tolerance. This is necessary because, after decomposing the system into microservices, the number of service interactions would be increased, and each external request may be transformed to a chain of internal service calls. It is important to fail fast[20] and not wait for each service call to be timed out in case of a failed service instance.

Finally, MP_1, MP_{11}, MP_{13}, MP_{14}, and MP_{15} are related to the deployment factor. All of these patterns are necessary for implementing an effective continuous delivery pipeline, which is a must for a microservices architecture. MP_{11} helps in changing software configuration without redeployment, and thus eliminates unnecessary redeployments and possible downtimes. MP_{13} and MP_{14} enable containerization, which is a tenet of microservices and helps with resource efficiency in production. With a large number of services in production, MP_{15} is necessary to know the situation in production and act if needed. MP_{15} is also useful for rearchitecting the system in the future. As such, all of these patterns were chosen.

To summarize, MP_1, MP_2, MP_4, MP_5, MP_6, MP_7, MP_8, MP_{10}, MP_{11}, MP_{12}, MP_{13}, MP_{14}, and MP_{15} patterns were employed to migrate the Backtory system to microservices architecture. Interested readers are referred to the works of Balalaie et al[3,4] for further details.

4.2.2 | Patterns composition

To construct a migration plan by composing the patterns, we need to first select the ones that need the minimum prerequisites for being applied, ie, the decision to migrate to microservices. Hence, MP_1 and MP_2 were selected as the first steps in the migration plan. Since they do not have any dependencies to each other, they can be applied simultaneously.

For the next steps, it is important to select the patterns that do not affect the end users. To address this requirement, first, MP_5 was employed to turn the DeveloperData component into a service since this component is not a user-facing one. Second, as the services could not be decomposed any further without affecting the end users, the configuration server was added to the system by applying the MP_{11} pattern. As a result, the services' configuration could be changed without redeploying the services. Third, to improve the current deployment process and making it automated in a single server, we used MP_{13} as the next step.

Up to this point, the selected steps did not affect the end users. However, subsequent steps would affect them, and an isolation mechanism was needed to make the following changes transparent to them. This could be achieved by using an edge server and applying the MP_{12} pattern.

Without having an approach to discover existing services, the edge server and services would use a static configuration to find others. To make it dynamic and remove manual configuration, we used a service registry solution by putting the MP_6 as the next step. Moreover, for easy and automatic registration of services, MP_7 was applied. Up to this point, all services know about the others; however, there was no systematic load balancing mechanism in place. Applying MP_8 as the next step solved this problem.

After setting up the dynamic communication between services, fault tolerance was injected by applying the MP_{10} pattern.

Having the dynamic communication between services, fault tolerance, change transparency to end users, and a simple automated deployment process enabled the system to be decomposed further. To this end, MP_4 was applied next.

Finally, by benefiting from the MP_{14} and MP_{15} patterns as the next steps, we could turn the system into a scalable and observable system. The final migration plan is depicted in Figure 16.
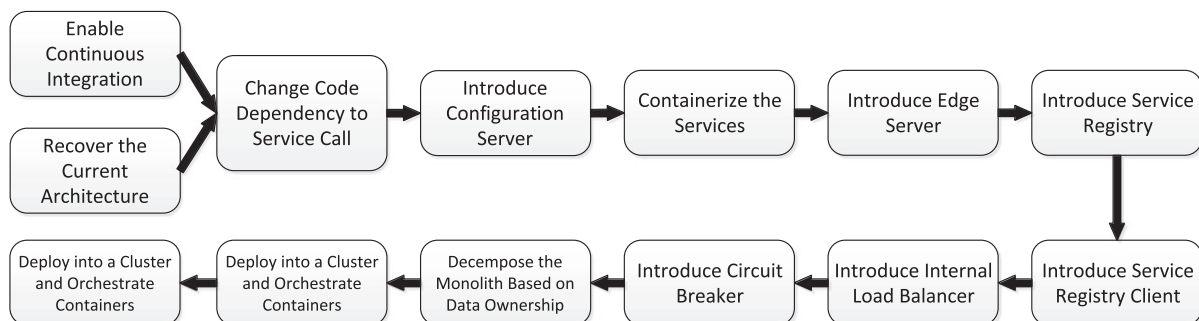

FIGURE 16 The Backtory's migration plan

4.3 | Adding new patterns to the repository

As microservices is not mature yet, and it is still in its early stages of exploration in a variety of domains; the completeness of patterns cannot be guaranteed. Instead, there should be means for expansion of the repository with new migration patterns generalized from new experiences in migrating to microservices. As we followed a systematic approach in defining patterns by providing a metamodel, selection factors, and factors in determining the granularity of the patterns, any new experiences can augment the repository if they conform to the metamodel and exist at an acceptable granularity. Additionally, if new selection factors are needed, they can also be added to the existing ones.

5 | EVALUATION

To evaluate the migration patterns introduced in this article, we adopted a mixed-method approach with the following evaluation objectives.

1. Understand our patterns' migration effectiveness, ie, whether the new software solution addresses the migration scope and purpose, ie, this assessment shall offer qualitative evidence over the validity of the proposed migration patterns in action from an in-depth perspective.
2. Understand the generalizability of our patterns, ie, this assessment shall offer qualitative evidence over the validity of our proposed approach as a whole, cross-referencing validity by means of triangulation of cases.

To address the first objective, we discuss the effectiveness of our patterns using an ethno-methodological approach adapted from the work of Hammersley and Atkinson.[21] More in particular, we report on our observations while extending and maintaining the Backtory platform introduced in the previous section. We discuss how applying the migration patterns and their inner design decisions paid off in practice (see Section 5.1).

Subsequently, to address the second objective, we perform our evaluations by applying case study research[22] carried out by means of empirical qualitative inquiry (ie, *grounded theory* (GT)[23]) and content analysis.[24] This part of our evaluation offers evidence of the identified patterns in industrial software rearchitecture specifications, which are completely independent of each other and have been stemmed from three distinct case studies (see Section 5.2).

5.1 | How did the migration patterns helped in future extensions and maintenance of Backtory?

The first version of Backtory was released in April 2016, and it only had the relational database management system as a service and limited authentication capabilities. However, during this period, more services like NoSQL as a service, game as a service, cloud code as a service, authentication as a service, etc, have been added to it such that Backtory currently includes more than 20 microservices. Thanks to the good design decisions, which have been made from the start of this project by applying the patterns proposed in this article, managing this number of related services is not a major issue at the moment. In the following, we explain how using these patterns helped the Backtory's team to scale from both a development and operational perspective.

5.1.1 | The development perspective

By separating the microservices' code repositories, the Backtory's team had a better experience in adding new team members and new services. In particular, for existing microservices, the learning curve was much lower for a new member compared with when the microservices architecture was not used. Additionally, to call another service, it is sufficient to know the service contracts (available through Swagger[§]) and the desired microservice name. In addition, many of the calls to cross-functional services (eg, security) are handled in the service template, and a new developer would not need to know about them from the beginning. Moreover, a new service can be added easily by planning out its initial microservices, forking the appropriate microservices templates, and starting to code. Furthermore, continuous integration pipelines together with containerization helped the team to easily have production-ready dependent services set up in the development environment. These enhancements were achieved by applying the MP_1 and MP_{13} patterns.

[§]http://swagger.io/

5.1.2 | The operational perspective

From the operational viewpoint, the Backtory's team experienced several improvements. First, by applying the MP_{11} pattern and having a configuration server in place, it became possible to change the services' configurations without redeploying them. In the beta release, configuration changes are quite common, and this helped the team to have fewer redeployments, and thus, fewer instances of downtime. Second, the MP_6, MP_7, MP_8, MP_{12}, and MP_{14} patterns helped the team to have zero-downtime redeployments. This is because services could discover themselves dynamically through MP_6 and MP_7, and contact each other in a scalable way through MP_8. Therefore, it became possible to redeploy a service independently without worrying about wiring configurations. In addition, MP_{12} hides the deployment of new instances of a microservice. MP_{14} made redeployment really easy by providing capabilities like rolling updates, ie, gradually replacing a service's instances and rolling back gradually in the case of a failure. Third, by applying MP_{10}, a better user experience was provided in failure situations. Moreover, by monitoring the circuit states, the team could find the connectivity problems in the production and investigate them further to find the root cause. Finally, by using MP_{15}, the team could monitor the microservices' resource usage and tune their corresponding containers accordingly. Furthermore, by having a historical resource usage, it was possible to alert a microservice's team of its unusual resource usage burst. To summarize, adapting the migration patterns introduced in this article helped the Backtory team from both a development and operational perspective.

5.2 | Case study research evaluation

5.2.1 | Research design

As previously outlined, to address evaluation objective 2, we adopted a case study research methodology.[22]

More specifically, the *objective* of our evaluation was to verify whether the proposed patterns could be found in industrial software specification documents whose aim is to specify the new version of a previously existing *analysis objects*, namely, large-scale software systems during the effort of modernizing their architecture. Our working evaluation hypothesis is that discovering the proposed patterns shows the practical external validity of our research solution.

To strengthen the external validity further by limiting the source information bias,[22] we *triangulate* our analysis using three industrial architecture modernization cases that are sensibly distant (in terms of domain) and different (in terms of architecture), yet comparable since they adopt the same architectural style, in this case, service-oriented architectures. Furthermore, to strengthen external and construct validity further by limiting observer bias,[22] we triangulate our analysis using two additional observers who coded the same data set separately, thus allowing subsequent interrater reliability score assessment, ie, for such assessment, we used the widely used Krippendorff alpha coefficient.[25] Our K_α score amounted to 0,81, higher than the standard reference score of $K_\alpha > 0,800$.

Specifically, the evaluation process was conducted as follows. First, through ethnography, we obtained data for three distinct case studies. Each case study is a software system for a commercial firm with industrial-scale architectures and extant design specifications,[¶] to which they were willing to give us access. We refer to these case studies as A, B, and C. The aforementioned design specifications were originally used to modernize very large legacy systems toward cloud-based (micro)services solutions between the years 2011 and 2014, and contain over 400 pages of diagrams and models carrying architecture refactoring decisions and rationale, augmented by making references to legacy architecture features and decisions.

Once the specification documents were acquired, we coded them by means of GT[23] using the concepts and relations stemmed from the patterns as GT labels.

Finally, we used content analysis[24] to identify the occurrence and frequency of any of the identified migration patterns defined in Section 3. The coding method we followed can be summarized in the steps as follows (see Section 5.2.6 for more details).

1. We elicited an initial list of concepts from the set of identified patterns in our research solution to be used as the initial list of GT codes for labeling the case study documents.[#]
2. We carried out GT microanalysis by manually labeling across our working documents with the list elaborated in the previous step.

[¶]Specifications carry industrial secrets protected by nondisclosure agreements. Watermarked replicas can be made available through written request for study replication purposes only.
[#] The list of codes is available upon request.

TABLE 2 The occurrences of migration patterns in our case studies identified through content analysis

Migration Patterns	Case Study A	Case Study B	Case Study C
MP$_1$: Enable continuous integration	0	0	0
MP$_2$: Recover the current architecture	4	0	1
MP$_3$: Decompose the monolith	34	31	14
MP$_4$: Decompose the monolith based on data ownership	3	5	3
MP$_5$: Change code dependency to service call	9	31	11
MP$_6$: Introduce service registry	31	29	14
MP$_7$: Introduce service registry client	2	1	10
MP$_8$: Introduce internal load balancer	0	1	1
MP$_9$: Introduce external load balancer	3	1	0
MP$_{10}$: Introduce circuit breaker	0	0	0
MP$_{11}$: Introduce configuration server	1	2	1
MP$_{12}$: Introduce edge server	0	1	10
MP$_{13}$: Containerize the services	34	31	14
MP$_{14}$: Deploy into a cluster and orchestrate containers	34	31	14
MP$_{15}$: Monitor the system and provide feedback	3	2	2

3. We elaborated the presence of the proposed patterns by counting the co-occurrences of the codes elicited from those patterns in diagrams, text, or models in our reference data set. For example, MP$_5$ proposes to reduce a code dependency into a service call; thus, for this pattern, the following four codes need to be found together: (1) current architecture feature, (2) modification or refactoring, (3) code dependency, and (4) remote service call.

Finally, as previously stated, the goal of our evaluation was to understand whether there was evidence of our proposed migration patterns in extant industrial migration specification projects. We also evaluate the relevance and impact of the patterns through a simple frequency analysis (see Table 2).

5.2.2 | Case studies overview

The three case studies were selected from over 90+ possible ones according to three parameters.

1. *Company Size*: The case studies include two very large global businesses (A and B) and a small- to medium-sized business (C).
2. *Market Segment*: The case studies have different target market segments, ie, an airport management system (A), a smart vehicles management system (B), and a smart building management system (C).
3. *Specification Structure*: The specification document structure must be homogenous, ie, the contents of the specifications across case studies must be comparable to minimize the possibility of overlaps or conflicts of themes and patterns identified by means of qualitative inquiry.

By applying the aforementioned parameters, we obtained the specifications outlined in the following sections.

5.2.3 | Case study A: An airport management system

The case study A is a subsidiary of a global information and communications technology company, which is active in the specification, construction, management, and deployment of any-size, any-domain, and information-technological business machines. The use case is the specification and redesign of an airport control system for one of the largest airport hubs in the world. The architecture in question includes 500+ components in the form of mixed structure services and different levels of abstraction. Moreover, the design specification comes with stakeholder concerns and domain assumptions. Figure 17 illustrates a sample business service specification from the documentation received from the case study A.

5.2.4 | Case study B: A smart vehicles management system

The case study B is part of a global information and communications technology solutions provider. The use case is the specification and redesign of a service-oriented architecture–based solution to create incentives for nonelectric vehicle drivers to switch to an electric vehicle and to increase the sustainability of the traffic control and management.
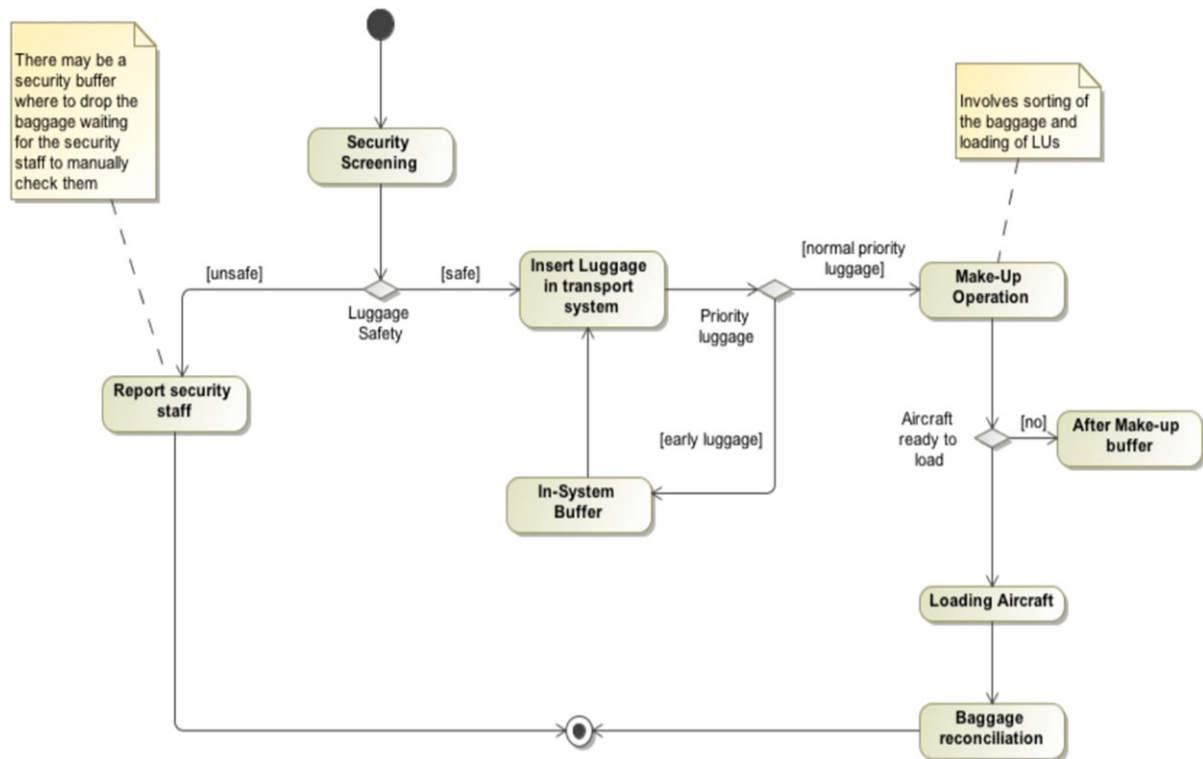
FIGURE 17   Sample business service from case study A. LU, Loading Unit [Colour figure can be viewed at wileyonlinelibrary.com]

The application was also redesigned to enable seamless switching to electric charging and management and operation of the smart grid infrastructure around such an application. The architecture in question includes over 350+ components arranged between internet of things handlers, middleware, innovative services, legacy, and open-source components. The available documentation covers also project organizational and integration dynamics and design decision making during stakeholder meetings. Figure 18 indicates a sample business service specification, named *smart vehicle charging*, from the documentation of this case study.

5.2.5 │ Case study C: A smart buildings management system

The case study C is a small to medium-sized smart building architecture service firm. The use case is the specification and redesign of energy-efficient buildings. As such, the redesign includes a set of services that offer the most energy-aware and cost-efficient solution for any physical building. The main focus of the project is smart automation and preventing the needless use of energy. The architecture in question includes around 400 components mostly dealing with hardware-software interfacing and driver programs, ambient control daemons, wireless sensor managers and security, safety, and other nonfunctional aspect handlers, or controllers. Figure 19 presents a sample business service specification, called *remote boiler temperature adjustment*, from the documentation of this case study.

5.2.6 │ Evaluation results: The occurrences of migration patterns

Through our GT-based approach and using content analysis, we identified evidence of the migration patterns in our case studies. Our study involved the analysis of 654 pages of systems design documentation and 46 UML diagrams. The data set in question reflects system-level UML specifications of *activities*, eg, see Figure 17 or Figure 19, *sequences*, *classes*, and *components*. The diagrams also reflect the (micro)service-based architectures for the cases under investigation (see Sections 5.2.3 to 5.2.5). Note that these service-based architectures reflect the currently deployed versions of the described systems and therefore reflect real industrial-strength applications. Essentially, our GT-based approach can be elaborated as follows.

- We reduced the entire set of patterns from Section 3 into separated lists of codes, ie, labels for concepts and/or relations.
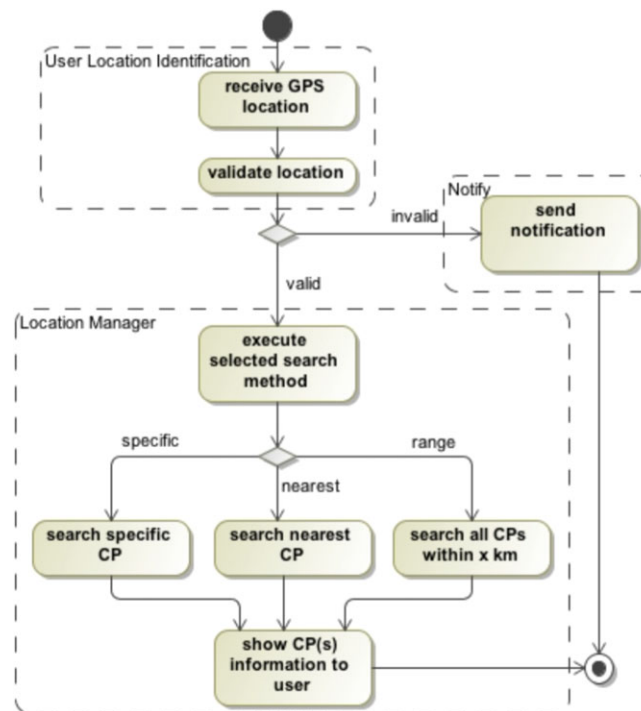- We hand-coded the aforementioned data set with said lists of codes by means of visual inspection and microanalysis.[23]

FIGURE 18 Sample business service from case study B. CP, Component Piece; GPS, global positioning system [Colour figure can be viewed at wileyonlinelibrary.com]
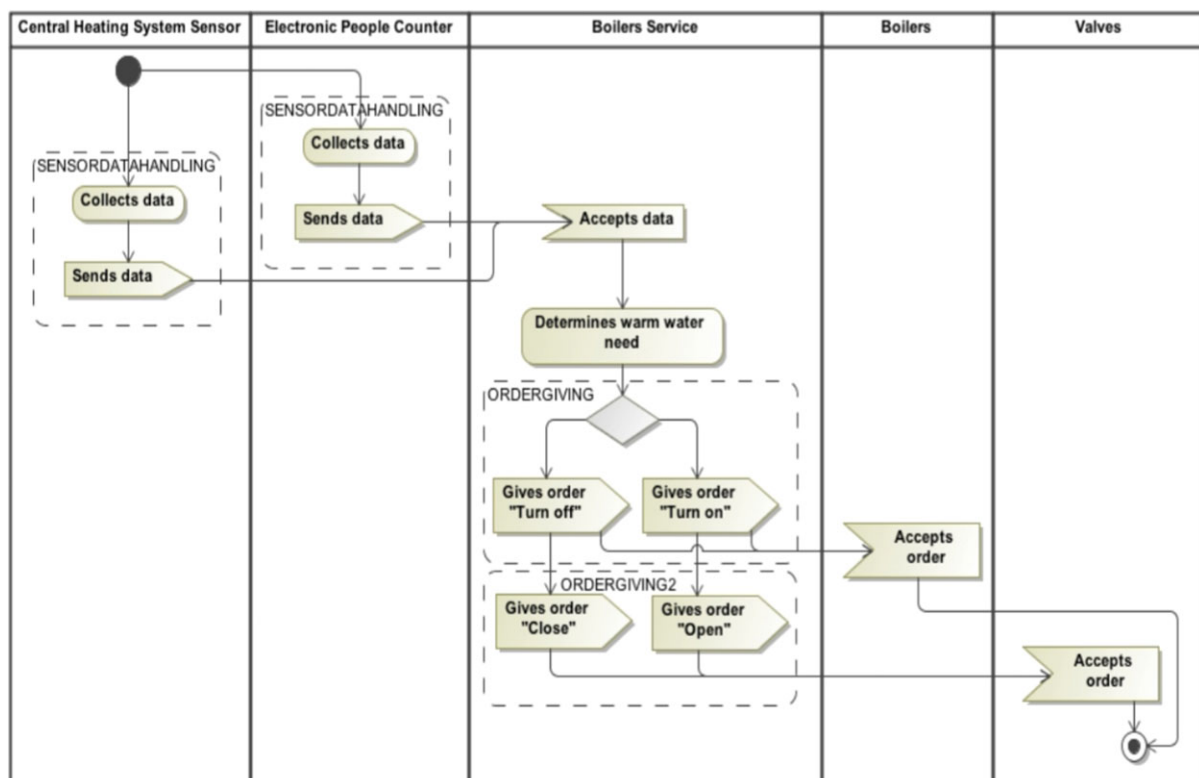

FIGURE 19 Sample business service from case study C [Colour figure can be viewed at wileyonlinelibrary.com]

- Results from the coding were imported into a qualitative data-processing engine, named *ATLAS.ti* (http://atlasti.com), for further analysis.

Using the ATLAS.ti analysis tool, we were able to identify the following.

1. *The recurrent relations across codes/labels:* The tool support allows us to define recurrent relations as a series of relations across defined codes and verify to what degree the occurrences of certain codes match a defined relation (ie, coded migration patterns). In other words, it allows us to confirm that when a certain label, eg, the "edge server" component of MP_{12} appears, then the label "service registry"$^{\parallel}$ is also applied and is following the recurrent relation defined for MP_{12}.
2. *The occurrences of the aforementioned recurrent relations:* The tool support allows us to traverse a document and the series of codes/labels applied to it as if it were a graph to be analyzed for recurrent patterns. As a result of this analysis, we report the number of times certain relations across codes/labels, ie, the migration patterns, existed throughout our data set.

The latter two analyses, conducted by means of the ATLAS.ti, can be referred to as a content analysis[26] that counts and relates the occurrences of certain concepts against those concepts that represent the migration patterns described in this article. Any evidence of a pattern in the design specification document as obtained in our analysis essentially equates to the successful instantiation of the pattern itself. Results of the aforementioned process are summarized in Table 2. In this table, rows represent the patterns' occurrences, ie, an instance of a full-match occurrence for a pattern previously reported in Section 3, while columns represent the cases under study.

Table 2 indicates that our proposed patterns are prevalent in the case studies and use cases identified. The majority of patterns were reported once in at least two independent case studies supporting the degree of generalizability despite the small number of cases. Notwithstanding this, not all of the patterns are equally distributed across projects reflecting the influence of parameters for the case study selection. However, as can be seen in this table, a number of patterns are less prominent in our case studies, eg, MP_8 and MP_{11}. This lack of occurrence may have been influenced by the availability of resources in the domains targeted by our case studies, ie, airports and buildings. Additionally, from Table 2, it is clear that there are no occurrences of MP_1 and MP_{10}. Although we have no data that allows more sensitive analysis of this gap, considering the nature of the projects involved in our case studies, we can speculate two possible conjectures. In particular, the lack of these two patterns may be related to the relative absence from our documentation of functional implementation and service operation, ie, deployment, details. Therefore, more extensive qualitative and quantitative research is needed to assess the generalizability of the proposed migration patterns to other industrial scenarios.

Nevertheless, our numbers indicate that the key activities in any service migration attempt, namely, (1) service decomposition, (2) service registry management, and (3) service containment and management, are covered almost equally by our patterns and across all three case studies. In addition, given the substantial occurrence of our migration patterns throughout a number of completely distinct case studies, sampled according to sound sampling principles and analyzed through empirical qualitative inquiry, we conclude that our migration patterns can play a key role in structuring and supporting industrial strength migration attempts to microservices.

5.2.7 | Threats to validity

As with any work evaluated by means of empirical research, this work is subject to challenges of validity and generalizability. While the case study method and associated qualitative analysis is common in industry and practice-driven research,[27] future studies can strengthen the validity of our results through (1) the operationalization and use of the proposed patterns within more projects and domains to understand their recurrence, and (2) conducting an ethnographic study to provide a design-science basis behind the patterns.

6 | RELATED WORK

Microservices have only become the focus of attention in academic literature relatively recently. As most of the efforts on maturing this architectural style have been done in industry (eg, by Netflix, Pivotal, and Thoughtworks), there is little or no evidence of microservices-related works before 2014 in academia. Many of the researchers currently working

---

$^{\parallel}$These label names are fictitious and only partially reflect the mnemonic labels used in our analysis.

on microservices architecture have come from the cloud architecture and migration community where they aim to use microservices architecture in the context of the cloud and as a cloud-native architecture.[1,28] Migrating applications to the cloud, and more specifically, pattern-driven migration[9,29] has been a popular topic in recent years.[5] However, none of the proposed approaches use microservices as their first-class citizens, which may lead to a nonoptimal solution.[3,4]

There is also a lack of experience reports on migrating to and using microservices in practice. To mitigate this, in the works of Balalaie et al,[3,4] we reported the migration process of Backtory to microservices and the lessons we have learned during this migration. Moreover, there are blog posts published by industry (eg, SoundCloud**), which lack details about the migration process.

Furthermore, many researchers see microservices as an architectural-level facilitator for DevOps practices. In the work of Balalaie et al,[4] we reported our experience of migrating Backtory to microservices in the context of DevOps, and explain how DevOps is related to microservices. In addition to our work, Bass et al[30] introduced microservices as a suitable architectural style for DevOps.

There are good guidelines for building microservices in the work of Newman,[31] but it does not offer a holistic and systematic approach for migrating to microservices. Krause[32] proposed a set of patterns and approaches for building and migrating to microservices, but it lacks details and does not consider DevOps practices. Furthermore, in his blog,†† Arun Gupta proposed a set of patterns for composing microservices. However, unlike our work, they are not related to migrating to microservices.

Interested readers are referred to the works of Jamshidi et al[5] and Pahl and Jamshidi[28] for detailed surveys of related work. In particular, Jamshidi et al[5] provided a systematic review of proposed solutions for migrating legacy systems into the cloud. Furthermore, Jamshidi and Pahl[28] identified, taxonomically classified, and systematically compared the work on microservices and their application in the cloud.

7 | CONCLUSIONS AND FUTURE WORK

In this work, we have presented a catalog of migration and rearchitecting patterns. These migration patterns facilitate rearchitecting noncloud-native architectures to migrate them to a cloud-native microservices-based architecture. The migration patterns were derived from our observations in several medium to large-scale industrial projects from different domains. In addition, we proposed a methodology based on SME to select and compose the migration patterns to prepare a migration plan.

Evaluating the proposed patterns in action, we observed a distinctive recurrence of the proposed patterns, especially four patterns, which reflect the basic tenets behind microservices. These observations clearly indicate their value for industrial practices. In addition, we observed that there is still space for improvement of the proposed patterns application and their methodological counterpart.

In future, we intend to develop a pattern language that enables us to compose the patterns in an automated way. Furthermore, the proposed catalog can become more comprehensive via introducing other new migration patterns, and they can be evaluated with further case studies.

ORCID

Abbas Heydarnoori 🅾 http://orcid.org/0000-0001-9785-2880
Pooyan Jamshidi 🅾 http://orcid.org/0000-0002-9342-0703
Damian A. Tamburri 🅾 http://orcid.org/0000-0003-1230-8961

**http://philcalcado.com/2015/09/08/how_we_ended_up_with_microservices.html
††http://blog.arungupta.me/microservice-design-patterns/

REFERENCES

1. Kratzke N, Quint P-C. Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study. *J Syst Softw*. 2017;126:1-16.

2. Fowler M, Lewis J. Microservices. http://martinfowler.com/articles/microservices.html; 2014. Accessed September 15, 2017.

3. Balalaie A, Heydarnoori A, Jamshidi P. Migrating to cloud-native architectures using microservices: an experience report. Paper presented at: Advances in Service-Oriented and Cloud Computing. Workshops of ESOCC 2015; 2015; Taormina, Italy.

4. Balalaie A, Heydarnoori A, Jamshidi P. Microservices architecture enables DevOps: migration to a cloud-native architecture. *IEEE Softw*. 2016;33(3):42-52.

5. Jamshidi P, Ahmad A, Pahl C. Cloud migration research: a systematic review. *IEEE Trans Cloud Comput*. 2013;1(2):142-157.

6. Pautasso C, Zimmermann O, Amundsen M, Lewis J, Josuttis N. Microservices in practice. Part 1: reality check and service design. *IEEE Softw*. 2017;34(1):91-98.

7. Henderson-Sellers B, Ralyté J, Ågerfalk PJ, Rossi M. *Situational Method Engineering*. New York, NY: Springer; 2014.

8. Gholami M, Sharifi M, Jamshidi P. Enhancing the OPEN Process Framework with service-oriented method fragments. *Softw Syst Model*. 2014;13(1):361-390.

9. Jamshidi P, Pahl C, Chinenyeze S, Liu X. Cloud migration patterns: a multi-cloud architectural perspective. Paper presented at: 10th International Workshop on Service-Oriented Computing (ICSOC 2014); 2014; Paris, France.

10. Menychtas A, Konstanteli K, Alonso J, et al. Software modernization and cloudification using the ARTIST migration methodology and framework. *Scalable Comput Pract Exp*. 2014;15(2):131-152.

11. Gansner ER, North SC. An open graph visualization system and its applications to software engineering. *Softw Pract Exper*. 2000;30(11):1203-1233.

12. Van der Hoek A, Wolf AL. Software release management for component-based software. *Softw Pract Exper*. 2002;33(1):77-98.

13. van Gurp J, Bosch J. Design, implementation and evolution of object oriented frameworks: concepts and guidelines. *Softw Pract Exper*. 2001;31(3):277-300.

14. Doolan EP. Experience with fagan's inspection method. *Softw Pract Exper*. 1992;22(2):173-182.

15. Henderson-Sellers B, Gonzalez-Perez C, Ralyté J. Comparison of method chunks and method fragments for situational method engineering. Paper presented at: 19th Australian Conference on Software Engineering; 2008; Perth, Australia.

16. Mirbel I, Ralyté J. Situational method engineering: combining assembly-based and roadmap-driven approaches. *Requir Eng*. 2006;11(1):58-78.

17. Prat N. Goal formalisation and classification for requirements engineering. In: Proceedings of the 3rd International Workshop on Requirements Engineering: Foundations of Software Quality; 1997; Barcelona, Spain.

18. Humble J, Farley D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Boston, MA: Addison-Wesley; 2010.

19. Evans E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston, MA: Addison-Wesley; 2004.

20. Nygard MT. *Release It! Design and Deploy Production-Ready Software*. Raleigh, NC: Pragmatic Bookshelf; 2007.

21. Hammersley M, Atkinson P. *Ethnography*. New York, NY: Routledge; 2003.

22. Yin RK. *Case Study Research: Design and Methods*. 2nd ed. Thousand Oaks, CA: SAGE Publications; 1994.

23. Corbin JM, Strauss A. Grounded theory research: procedures, canons, and evaluative criteria. *Qual Sociol*. 1990;13(1):3-21.

24. Hsieh H-F, Shannon SE. Three approaches to qualitative content analysis. *Qual Health Res*. 2005;15(9):1277-1288.

25. Krippendorff K. *Content Analysis: An Introduction to Its Methodology*. 2nd ed. Thousand Oaks, CA: SAGE Publications; 2004.

26. Kaid LL, Wadsworth AJ. Content analysis. In: Emmert P, Barker LL, eds. *Measurement of Communication Behavior*. New York, NY: Longman; 1989;197-217.

27. Proper E, Gaaloul K, Harmsen F, Wrycza S. Practice-driven research on enterprise transformation. In: Proceedings of the 4th Working Conference (PRET 2012); 2012; Gdańsk, Poland.

28. Pahl C, Jamshidi P. Microservices: a systematic mapping study. In: Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016); 2016; Rome, Italy.

29. Jamshidi P, Pahl C, Mendonça NC. Pattern-based multi-cloud architecture migration. *Softw Pract Exp*. 2016;47(9):1159-1184.

30. Bass L, Weber I, Zhu L. *DevOps: A Software Architect's Perspective*. New York, NY: Pearson Education, Inc.; 2015.

31. Newman S. *Building Microservices: Designing Fine-Grained Systems*. Sebastopol, CA: O'Reilly Media, Inc.; 2015.

32. Krause L. *Microservices: Patterns and Applications: Designing Fine-Grained Services by Applying Patterns*. Lucas Krause; 2015.