Microservices Architecture Enables DevOps

Migration to a Cloud-Native Architecture

Armin Balalaie and Abbas Heydarnoori, Sharif University of Technology

Pooyan Jamshidi, Imperial College London

// This article reports on experiences and lessons learned during incremental migration and architectural refactoring of a commercial mobile back end as a service to microservices architecture. It explains how the researchers adopted DevOps and how this facilitated a smooth migration. //

A LOOK AT the searches related to the term "microservices" on Google Trends revealed that the top searches are now technology driven. This implies that the time of general search terms such as "What is microservices?" has now long passed. Not only are software vendors (for example, IBM and Microsoft) using microservices and DevOps practices, but also content providers (for example, Netflix and the BBC) have adopted and are using them.

In addition, Google Trends reveals that both DevOps and microservices are growing concepts, with an equal rate of growth after 2014 (see Figure 1). Although DevOps can also be applied to monolithic software systems, microservices enable effective implementation of DevOps by promoting the importance of small teams.[1] (For more on DevOps and Microservices, see the related sidebar.)

A microservices architecture is a cloud-native architecture that aims to realize software systems as a package of small services. Each service is independently deployable on a potentially different platform and technological stack. It can run in its own process while communicating through lightweight mechanisms such as RESTful or RPC-based APIs—for example, Finagle. (REST stands for Representational State Transfer.) In this setting, each service is a business capability that can utilize various programming languages and data stores and is developed by a small team.[2]

Migrating monolithic architectures to microservices brings in many benefits. In particular, it provides adaptability to technological changes to avoid technology lock-in and, more important, reduced time-to-market and better development team structuring around services.[3]

Here we explain our experiences and lessons learned during incremental migration of Backtory (www.backtory.com), a commercial *mobile back end as a service* (MBaaS), to microservices in the context of DevOps. Microservices help Backtory in various ways, especially in shipping new features more frequently and providing scalability for the collective set of users from different mobile-app developers.

Furthermore, we report on migration patterns we developed on the basis of our observations in migration projects. Practitioners can use these patterns to migrate monolithic software systems to microservices. In addition, system consultants can use

 0740-7459/16/$33.00

DEVOPS AND MICROSERVICES

DevOps is a set of practices that aim to decrease the time between changing a system and transferring that change to the production environment. However, they also insist on maintaining software quality in terms of both code and the delivery mechanism. Any technique that enables these goals is considered a DevOps practice.[1,2]

In particular, continuous delivery (CD) is a DevOps practice that enables on-demand deployment of software to any environment through automated machinery.[3] CD is an essential companion of microservices as the number of deployable units increases.

Another critical DevOps practice is continuous monitoring (CM),[4] which not only provides developers with performance-related feedback but also facilitates detecting any operational anomalies.[2]

References

1. L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*, Addison-Wesley Professional, 2015.
2. A. Brunnert et al., *Performance-Oriented DevOps: A Research Agenda*, tech. report SPEC-RG-2015-11, Standard Performance Evaluation Corp., 2015; http://arxiv.org/pdf/1508.04752.pdf.
3. J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley Professional, 2010.
4. A. van Hoorn et al., *Continuous Monitoring of Software Services: Design and Application of the Kieker Framework*, research report, Kiel Univ., Nov. 2009.

them to help organizations plan the adoption of DevOps in their migration to microservices.

Architectural Concerns for Microservices Migration

Backtory, which was developed at PegahTech (www.pegahtech.ir), provides back-end services to mobile developers who don't know any server-side programming languages. It originally was an RDBMS (relational database management system) functioning as a service. Developers defined database schemas in Backtory's developer dashboard, and Backtory provided a software development kit for the desired target platform (for example, Android or iOS). Afterward, the developers coded on their desired platforms using their domain objects, which made service calls on their behalf to fulfill their requests. Over time, new services are being added to Backtory, such as chat, indexing, and NoSQL.

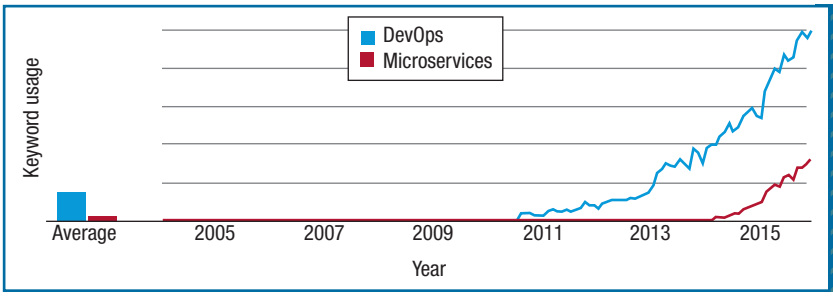

FIGURE 1. The increase in the use of the keywords "DevOps" and "microservices," according to a Google Trends report.

Backtory is written in Java using the Spring framework. The underlying RDBMS is Oracle Database 11g. Backtory uses Maven to fetch dependencies and build the project. Before migration, all the services were in a Git repository, and Backtory used Maven's modules to build services. Deployment of services to development machines was done using Maven's Jetty plug-in. However, deployment to the production machine was a manual task.

Figure 2a illustrates Backtory's architecture before migration and shows Backtory's five main components. For more on them, see the sidebar, "Backtory Components before the Migration."

Why We Migrated Backtory

What motivated us to migrate Backtory to microservices was an issue related to the requirement to provide chat as a service. To implement this requirement, we chose ejabberd because it's scalable and can run on clusters. To this end, we wrote

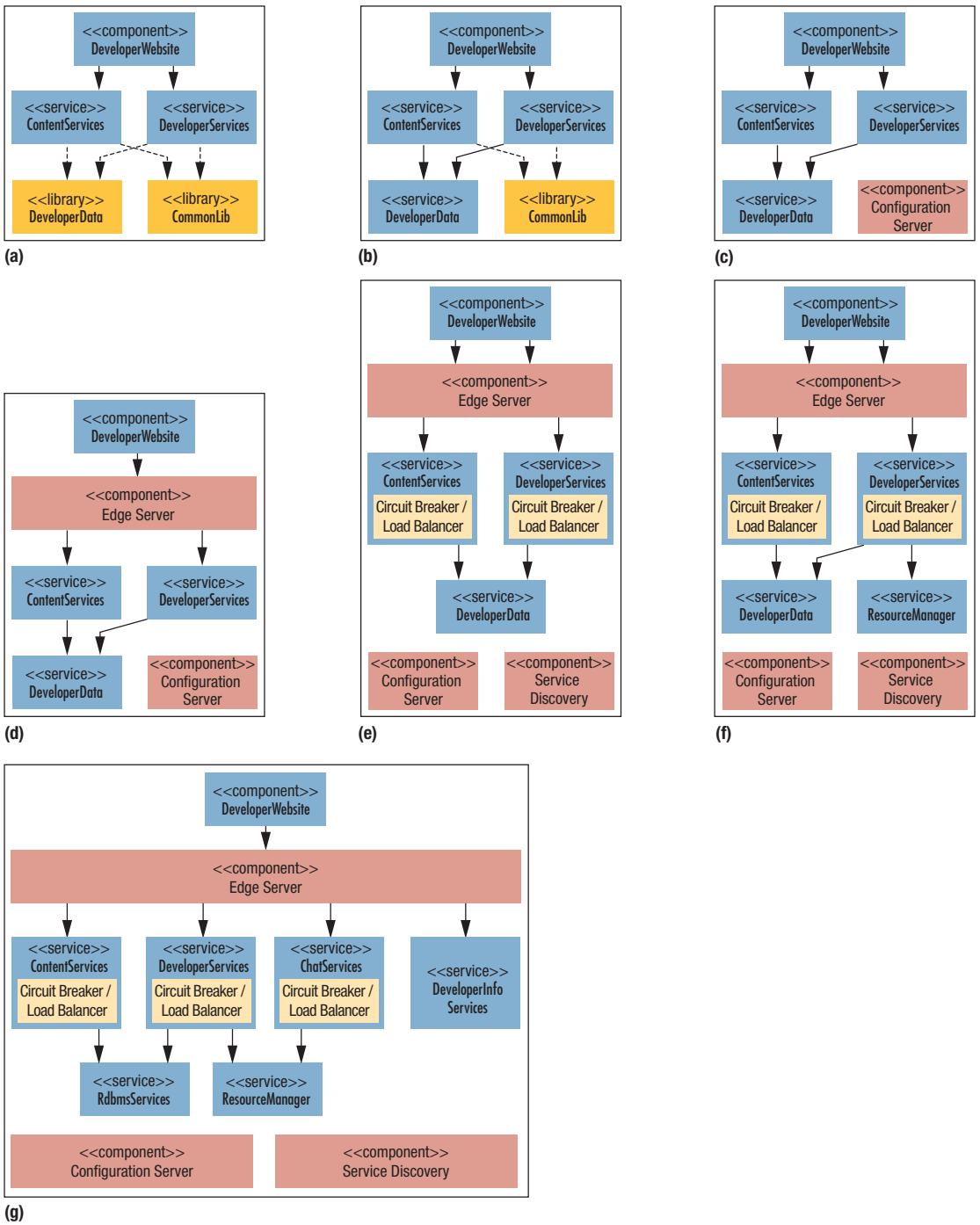

FIGURE 2. Migrating Backory to microservices. Solid arrows indicate service calls; dashed arrows indicate library dependencies. (a) Backtory's architecture before the migration. (b) Transforming **DeveloperData** to a service. (c) Introducing the Configuration Server. (d) Introducing the Edge Server. (e) Introducing dynamic service collaboration. (f) Introducing **ResourceManager**. (g) Backtory's target architecture after the migration.

a Python script that enabled ejabberd to perform authentication using Backtory. The major issue in our service was the on-demand capability. Dealing with this issue led us to actions that provided further motivations for migration.

The need for reusability. To address the on-demand capability, we started to automate the process of setting up a chat service. One step was to spin off a MySQL database for each user. Our system had a pool of servers, each containing an instance of the Oracle RDBMS and an instance of DeveloperServices running. During RDBMS instantiation, a server was selected randomly, and related users and table spaces were created in the Oracle server. This design raised several issues because its original purpose was just to fulfill the RDBMS service needs and it was tightly coupled to the Oracle server. So, we needed a database reservation system that both the RDBMS and chat services could use.

The need for decentralized data governance. Another issue was that whenever anyone added metadata about different services, that metadata was added to DeveloperData. This practice wasn't good because services are independent units that only share their contracts with other parts of the system.

The need for automated deployment. As the number of services grew, another problem was to automate deployment and decouple the build life cycle of each service from the other services.

The need for built-in scalability. Backtory aims to serve millions of users. By increasing the number of services, we needed a new approach for handling such scalability because individually scaling services requires major effort and can be error-prone if not handled properly.

BACKTORY COMPONENTS BEFORE THE MIGRATION

Before migrating to microservices, Backtory comprised these components:

- CommonLib contains shared functionalities, such as utility classes, that the rest of the system will use.
- DeveloperServices is where the services related to managing the domain model of developers' projects reside. Using these services, developers can add new models, edit existing ones, and so on.
- ContentServices holds the services the target software development kit uses to perform CRUD (create, read, update, and delete) operations on the model's objects.
- DeveloperData holds the information of developers who are using the Backtory service and their domain model metadata entities that are shared between DeveloperServices and ContentServices.
- DeveloperWebsite is an application written in HTML and JQuery that acts as a dashboard for developers. It leverages DeveloperServices.

Backtory's Target Architecture after the Migration

We transformed Backtory's core architecture to the target architecture through refactorings. These changes included introducing microservices-specific components and rearchitecting the system.

In the microservices state-of-the-art,[4,5] domain-driven design and the Bounded Context pattern are common practices to transform a system's architecture into microservices.[6] Because our domain wasn't complex, we rearchitected the system on the basis of domain entities in DeveloperData. We discuss the final architecture (see Figure 2g) in more detail later.

Backtory's new technology stack included Spring Boot for the embedded application server and fast service initialization, the OS's environment variables for configuration, and Spring Cloud Context and the Spring Cloud Config server to separate the configuration from the source code, as continuous delivery (CD) practices recommend. (For more on CD, see the "DevOps and Microservices" sidebar.) Additionally, the Netflix OSS (open source software) provided microservices-specific components (such as the Service Discovery), and Spring Cloud Netflix integrated the Spring framework with the Netflix OSS project. We also chose Eureka for the Service Discovery, Ribbon as the Load Balancer, Hystrix as the Circuit Breaker,[7] and Zuul as the Edge Server,[8] which all are parts of the Netflix OSS project. We specifically chose Ribbon instead of other load balancers—for example, HAProxy—because of its integration with the Spring framework and other Netflix OSS projects, particularly Eureka.

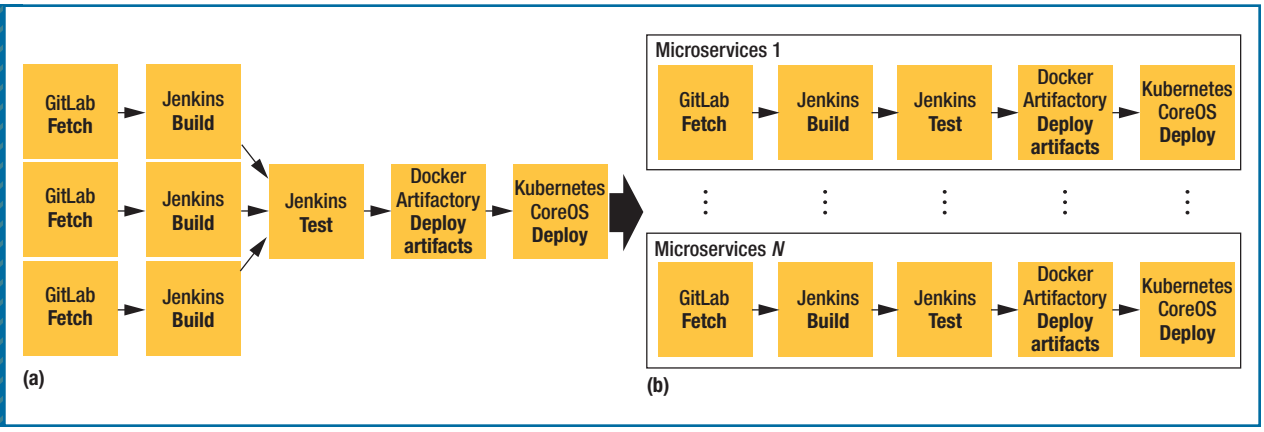

FIGURE 3. Moving from (a) a monolithic pipeline to (b) a microservices pipeline. The final delivery pipeline has independent delivery for each service, so each can be deployed independently.

The Migration

During migration, we performed the architectural refactorings we just mentioned and some crosscutting changes to enable DevOps.

Architectural Refactorings

Migrating the system wasn't a one-step procedure; we performed it incrementally without affecting the end users. We treated the migration steps as architectural changes (adding or removing components) comprising two states: before and after migration.

Preparing the continuous-integration pipeline. Continuous integration (CI) is the first step toward CD. It lets developers integrate their work with others' work early and regularly and helps prevent future conflicts.[9] Achieving this goal requires a CI server, an as-a-service or self-hosted code repository, and an artifact repository. We chose Jenkins as the CI server, self-hosted Gitlab as the code repository, and Artifactory as the artifact repository.

Because services can have multiple instances running, deploying microservices using virtualization isn't cost effective and introduces heavy computational overhead. Furthermore, we needed to use configuration management systems to create the production and test environments.

Using containers let us deploy service instances with lower overhead and better isolation than with virtualization. Another major benefit was portability; we could deploy anywhere that supports containerization without changing our source codes or container images. Docker is a tool for application containerization.[10] Because we were going to use Docker, our pipeline needed the Docker Registry too.

To summarize, in this step, we integrated Gitlab, Jenkins, Artifactory, and the Docker Registry as a CI pipeline. As Figure 3 shows, the fundamental difference between this delivery pipeline and a monolithic one is that ours has independent pipeline delivery for each service, so each can be deployed independently. Previously, we were using integration tests that required running the whole set of tests if just one service changed. We replaced the integration tests with consumer-driven contracts[11] and tests that led to independent testing of each service, using its consumers' expectations. This change minimized interteam coordination, which, even though the testing strategy was more complex, enabled forming smaller teams as a DevOps practice.

Transforming DeveloperData to a service. We changed DeveloperData to use Spring Boot because of its advantages (as we discussed before). Furthermore, as Figure 2b shows, we changed DeveloperData to expose its functionalities as a RESTful API. In this way, its dependent services won't be affected when its internal structure changes. Because DeveloperData entities have service-level dependency, a single service will handle their governance, and DeveloperData won't act as an integration database[12] for its dependent services anymore. Accordingly, we adapted DeveloperServices and ContentServices to use DeveloperData as a service and not as a Maven dependency.

Introducing CD. One of the best CD practices is to separate the source code, configuration, and environment specification so that they can evolve independently.[9] In this way, we can change the configuration

without redeploying the source code. By leveraging Docker, we removed the need for specifying environments because the Docker images produce the same behavior in different environments.

To separate the source code and configuration, we ported every service to Spring Boot and changed them to use the Spring Cloud Configuration Server and Spring Cloud Context to resolve their configuration values (see Figure 2c). In this step, we also separated services' code repositories to have a clearer change history and to separate each service's build life cycle. We also created a Dockerfile for each service, which is a configuration for creating Docker images for that service. We then created a CI job per service and ran the jobs to populate our repositories. Having the Docker image of each service in our private Docker registry, we could run the whole system with Docker Compose, using only one configuration file. Starting from this step, we had an automated deployment on a single server.

Introducing the Edge Server. Rearchitecting the system would change the internal service architecture. So, we introduced the Edge Server to minimize internal changes' impact on end users (see Figure 2d). Accordingly, we adapted DeveloperWebsite.

Introducing dynamic service collaboration. We then introduced the Service Discovery, Load Balancer, and Circuit Breaker (see Figure 2e). Dependent services should locate each other through the Service Discovery and Load Balancer, and the Circuit Breaker will make our system more resilient during service calls. Introducing these components made our developers more comfortable with these new concepts and accelerated the migration.

Introducing ResourceManager. We introduced ResourceManager by factoring out the server-related entities—for example, AvailableServer—from DeveloperData and introducing a feature—MySQL database reservation—to satisfy our chat service requirements (see Figure 2f). Accordingly, we adapted DeveloperServices to use this service for database reservations.

Introducing ChatServices and DeveloperInfoServices. The final refactoring step introduced two services (see Figure 2g). We introduced ChatServices to persist chat-service-instances metadata and create chat service instances. We introduced DeveloperInfoServices by factoring out developer-related entities (for example, Developer) from DeveloperData.

Clusterization. In this step, we set up a cluster of CoreOS instances containing Kubernetes agents. We then deployed our services on this cluster instead of a single server. As Figure 3 shows, independent testing of services using consumer-driven tests enabled us to also deploy each service independently. So, a change in a service would no longer result in redeploying the whole system.

Crosscutting Changes

The necessary changes involved enabling continuous monitoring to bridge the gap between development and operations, and changing the team structures.

Bridging development and operations. In the context of microservices, each service can have its own independent monitoring facility owned by the operations team. This enables independent flow of per-service performance information to the development team. The development team can adopt appropriate parametric performance models to estimate the end-to-end system performance or facilitate what-if analyses. This helps the team refactor the architecture to remove performance bottlenecks.[13]

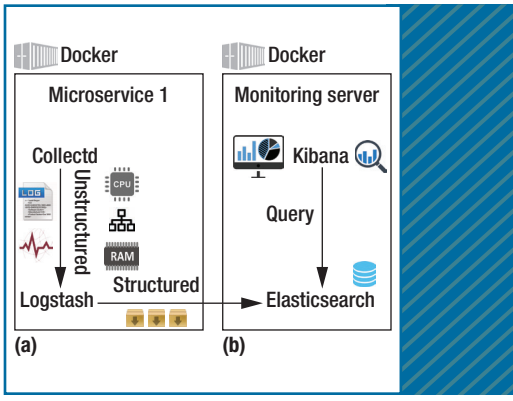

FIGURE 4. The monitoring and performance feedback infrastructure. (a) The client container. (b) The server container. This architecture lets us monitor each microservice independently and react to any anomalies we uncover on the basis of the online monitoring data.

As Figure 4 shows, our monitoring solution comprises both server and client containers. A server container manages the monitoring tools. In our deployment, it contains Kibana for visualization and Elasticsearch for consolidating the monitoring metrics. With Elasticsearch, users can horizontally scale and cluster multiple monitoring components.

A client container contains the monitoring agents and the facilities to forward the data to the server. In this particular instance, it contains Logstash and the collected modules. Logstash connects to the Elasticsearch cluster as the client and stores the processed and transformed metrics data there.

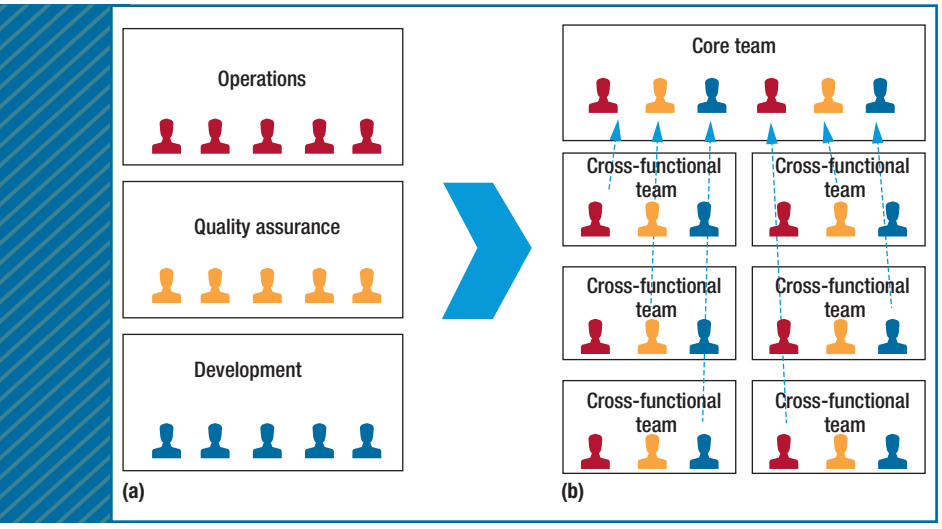

FIGURE 5. DevOps team formation. (a) Traditional horizontal teams. (b) Vertical teams in DevOps. In DevOps, each team is responsible for a service and contains people with different skills, such as development and operations skills. The team members cooperate from the project's start to create more value for the particular service's end users.

This architecture lets us monitor each microservice independently and react to any anomalies we uncover on the basis of the online monitoring data. To detect anomalies, we use a statistical model we trained using the monitoring data in normal situations. Then, for each new incoming monitoring data point, the anomaly detection module calculates a score, using principal component analysis, to spot outliers.

Changing team structures. Traditional software methods encourage horizontal division of project members into functionally separated teams. This division normally causes the creation of development, quality assurance, and operations teams (see Figure 5a). Such separation delays the development life cycle owing to transitions between teams and the teams' various reactions to change frequency. Moreover, with microservices, because each team should be responsible for its own services, functionally separated teams can't benefit from the increased comprehensibility of code and easier assimilation of new team members that system decomposition enables.

In contrast, DevOps recommends vertically dividing project members into small cross-functional teams, which fits microservices well. Each team is responsible for a service and contains people with different skills, such as development and operations skills. The team members cooperate from the project's start to create more value for the particular service's end users. This added value results from more frequent releases of new features to production, which eliminates the transition overheads with horizontal teams. Furthermore, because each team focuses on a particular service, each service's code has much higher maintainability and comprehensibility, and teams can add members with a lower learning curve.

During the migration, we gradually formed small cross-functional teams for each new service constructed as a result of architectural refactorings (see Figure 5b). Furthermore, we formed a core team that's responsible for shared capabilities; it consists of representatives of each service's team. This core team has an overall view of the service interactions in the system and is in charge of critical architectural decisions. It also handles interservice refactorings that involve transferring functionalities between services and updating the corresponding rules in the Edge Server.

Lessons Learned

Here, we share five lessons we learned that might be helpful for others trying to migrate to microservices.

First, deployment in the development environment is difficult. Although the application code is now in isolated services, developers must also deploy the dependent services to run the isolated services on their machines. This problem occurred after we introduced dynamic service collaboration. To solve it, we chose Docker Compose and put a sample deployment description file in each service so that the dependent services can be easily deployed from our private Docker registry.

Second, service contracts are critical. Changing so many services that expose their contracts only to each other could be error-prone. Even a small change in the contracts can break part or even all of the system. One possible solution is service versioning, but it could make deploying each service even more complex. So, people usually don't recommend service versioning for microservices. Thus, techniques such as the Tolerant Reader service design pattern[11]

are more advisable to avoid service versioning. Consumer-driven contracts could help greatly in this regard because the team responsible for a service can be confident that most of its customers are satisfied with the service.

Third, distributed-system development needs skilled developers. Microservices is a distributed architectural style. Furthermore, for such architectures to be fully functional, they need supporting services such as service discovery and a load balancer. During the early migration steps, we tended to spend much time describing these concepts and their corresponding tools and libraries to novice developers. Still, those developers often misused these things. So, to get the most out of microservices, teams need members who are familiar with these concepts and comfortable with this type of programming.

Fourth, creating service development templates is important. Polyglot persistence and the use of different programming languages are promises of microservices. Nevertheless, in practice, a radical interpretation of these promises could result in chaos in the system and even make it unmaintainable. As a solution, after architectural refactoring began, we started to create service development templates. We have different templates for creating microservices in Java using different data stores; these templates include a simple sample of a correct implementation. We're also creating templates for Node.js. One simple rule is that a senior developer should first examine each new template to identify potential challenges.

Finally, microservices architecture isn't a silver bullet. It was beneficial for us because our system needed that flexibility and because we had Spring Cloud and Netflix OSS, which made migration and development much easier. However, as we mentioned before, adopting microservices will introduce complexities to the system that require much effort to resolve.

Microservices Migration Patterns

After our migration project, we decided to make our experiences and best practices more accessible for other similar projects by abstracting them as migration patterns. In this way, other developers can reuse these practices to create migration plans by instantiating and composing the patterns.

Migrating to the cloud, specifically through cloud-native architectures such as microservices, is a multidimensional problem and thus nontrivial.[14] So, without a well-thought-out methodology, migration could become a trial-and-error endeavor that not only wastes much time but also can lead to a wrong solution. Furthermore, because factors such as the requirements, current situation, and team members' skills could vary among companies and scenarios, a

TABLE 1

Migration patterns related to this article.[8]

Pattern name	DevOps impact
Enable the Continuous Integration (CI)	CI is the first step toward continuous delivery (CD), a DevOps practice.
Recover the Current Architecture Decompose the Monolith Decompose the Monolith Based on Data Ownership Change Code Dependency to Service Call	These patterns enable decomposition of the system into smaller services, which leads to smaller teams.
Introduce Service Discovery Introduce Service Discovery Client Introduce Internal Load Balancer Introduce External Load Balancer	Dynamic discovery of services removes the need for manual wiring, thereby promoting more independent deployment pipelines.
Introduce Circuit Breaker	Failing fast can decrease the coupling between services, thereby contributing to independent service deployments.
Introduce Configuration Server	Separating configuration from code is a CD best practice.
Introduce Edge Server	The Edge Server not only allows the development team to more easily change the system's internal structure but also permits the operations team to better monitor each service's overall status.
Containerize the Services	Containers can produce the same environment in both production and development, thus reducing conflicts between the development and operations teams.
Deploy into a Cluster and Orchestrate Containers	Cluster management tools reduce the difficulties around deployment of many instances from different services in production, thus reducing the operations team's resistance to the development team's changes.
Monitor the System and Provide Feedback	Performance monitoring enables systematic collection of performance data and sharing to enhance decision making. For example, the development team can use such information to refactor the architecture if it discovers a performance anomaly in the system.

unique and rigid methodology won't suffice. Thus, instead of a one-fits-all methodology, we chose a situational-method-engineering approach.[15]

The first step toward this approach is to prepare a method base or pattern repository consisting of reusable process patterns or method chunks, each instantiated from a predefined metamodel. To this end, using our previous experience in defining migration patterns,[16] we documented our experience in this project and similar practices in the microservices state-of-the-art[3,8] (see http://microservices.io) as method chunks. We tried to enrich each step in our migration with the precise definition of the corresponding situation, the problem to solve, and the proposed solution's possible challenges, thus forming a pattern template.[8]

Part of these patterns describes why we need supporting components—for example, the Service Discovery—and the prerequisites for their introduction. We also provided solutions and advice for decomposing a monolithic system to the constituting services and preparing the system's current and target architectures as a roadmap for migration planning. In addition, we provided hints about the containerization of services and their deployment in a cluster.

Table 1 lists the patterns related to this article; details on them appear in a supplementary technical report.[8]

As Figure 6 shows, with an initial set of these patterns, method engineers can apply the construction guidelines to create a concrete method based on their migration requirements. For example, in response to the need for

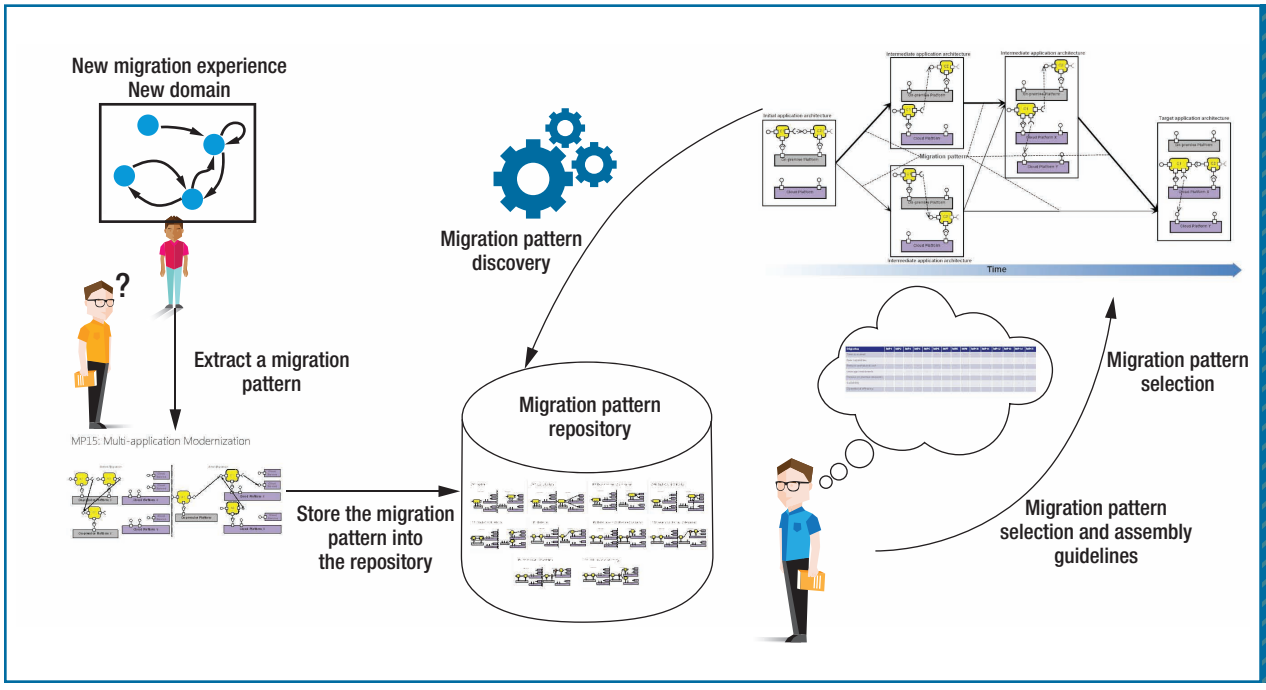

FIGURE 6. Selecting migration patterns, instantiating and composing a migration plan, and extending the migration pattern repository. With an initial set of patterns, method engineers can apply the construction guidelines to create a concrete method based on their migration requirements.

"polyglotness," method engineers can access the decomposition patterns. Then, they can select a pattern suitable for their needs.

The architectural refactorings resulting from pattern applications can't be ad hoc. Invariants exist that must be satisfied during the architectural transition.[17] The most important invariants are to keep the system stable after applying a pattern, perform one architectural change at a time, and keep the system's users unaffected. However, although a single step must conform to these invariants, the steps and their execution order collectively might violate them. Method engineers should consider this when selecting patterns.

During Backtory's migration, we introduced the Edge Server before introducing components related to dynamic collaboration between services to make the following changes transparent to users. If the order of these steps changed, we couldn't satisfy some invariants.

Future migrations can add patterns to the pattern repository. This repository will serve as an extensible source for the DevOps community through which they can reuse patterns for migrating to microservices. For an example repository, visit http://microservices.io.

Traditional methods for software development advocate separated development and operations teams in which the development team provides the operations team with deployment artifacts and details. The problem is that these teams behave differently regarding the frequency of changes, such that the development team tends to produce more changes and the operations team insists on higher stability. Furthermore, because large teams are working on monolithic systems, changes need much coordination. Even with system componentization, the final integration needs considerable coordination. These issues lengthen the development life cycle.

DevOps, together with microservices, is tackling these issues by providing the necessary equipment to minimize coordination among the teams responsible for each component and removing the barriers for an effective, reciprocal relationship between the development and operations teams. Indeed, in the DevOps setting, these teams help each other through continuous valuable feedback. Table 1 briefly describes the

problems our patterns tackle and how they affect DevOps.

References

1. L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*, Addison-Wesley Professional, 2015.
2. M. Fowler and J. Lewis, "Microservices," 26 Mar. 2014; http://martinfowler.com/articles/microservices.html.
3. A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Migrating to Cloud-Native Architectures Using Microservices: An Experience Report," to be published in *Proc. 1st Int'l Workshop Cloud Adoption and Migration*.
4. S. Newman, *Building Microservices*, O'Reilly Media, 2015.
5. M. Stine, *Migrating to Cloud-Native Application Architectures*, O'Reilly Media, 2015.
6. V. Vernon, *Implementing Domain-Driven Design*, Addison-Wesley Professional, 2013.
7. M. Nygard, *Release It! Design and Deploy Production-Ready Software*, Pragmatic Bookshelf, 2007.
8. A. Balalaie, A. Heydarnoori, and P. Jamshidi, *Microservices Migration Patterns*, tech. report TR-SUTCE-ASE-2015-01, Automated Software Eng. Group, Sharif Univ. of Technology, Oct. 2015; http://ase.ce.sharif.edu/pubs/techreports/TR-SUT-CE-ASE-2015-01-Microservices.pdf.
9. J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley Professional, 2010.
10. C. Pahl, "Containerization and the PaaS Cloud," *IEEE Cloud Computing*, vol. 2, no. 3, 2015, pp. 24–31.
11. R. Daigneau, *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*, Addison-Wesley Professional, 2011.
12. G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Addison-Wesley Professional, 2004.
13. A. Brunnert et al., *Performance-Oriented DevOps: A Research Agenda*, tech. report SPEC-RG-2015-11, Standard Performance Evaluation Corp., 2015; http://arxiv.org/pdf/1508.04752.pdf.
14. P. Jamshidi, A. Ahmad, and C. Pahl, "Cloud Migration Research: A Systematic Review," *IEEE Trans. Cloud Computing*, vol. 1, no. 2, 2013, pp. 142–157.
15. B. Henderson-Sellers et al., *Situational Method Engineering*, Springer, 2014.
16. P. Jamshidi et al., "Cloud Migration Patterns: A Multi-cloud Architectural Perspective," *Service-Oriented Computing—ICSOC 2014 Workshops*, LNCS 8954, Springer, 2014, pp. 6–19.
17. A. Ahmad, P. Jamshidi, and C. Pahl, "Classification and Comparison of Architecture Evolution Reuse Knowledge: A Systematic Review," *J. Software: Evolution and Process*, vol. 26, no. 7, 2014, pp. 654–691.

ABOUT THE AUTHORS

ARMIN BALALAIE is a master's student in the Sharif University of Technology's Department of Computer Engineering. His research interests include software engineering, cloud computing, and distributed systems. Balalaie received his BSc in software engineering from Shiraz University. Contact him at armin.balalaie@gmail.com.

ABBAS HEYDARNOORI is an assistant professor in the Sharif University of Technology's Department of Computer Engineering. His research interests include reverse engineering and reengineering software systems, mining software repositories, and recommendation systems in software engineering. Heydarnoori received a PhD from the University of Waterloo's School of Computer Science. Contact him at heydarnoori@sharif.edu.

POOYAN JAMSHIDI is a postdoctoral research associate in Imperial College London's Department of Computing. His primary research interest is self-adaptive software, in which he applies statistical machine learning and control theory to enable self-organizing behaviors in distributed systems for processing big data. Jamshidi received a PhD in computing from Dublin City University. Contact him at p.jamshidi@imperial.ac.uk.

See www.computer.org/software-multimedia for multimedia content related to this article.