Microservices Architecture Enables DevOps: An Experience Report on Migration to a Cloud-Native Architecture

Armin Balalaie Abbas Heydarnoori Pooyan Jamshidi

Abstract

The *microservices architecture* is one of the first service-based architectural styles that has been introduced, applied in practice, and become popular when the DevOps practices gained momentum in the software industry. Migrating monolithic architectures to cloud-native architectures like microservices brings in many benefits such as flexibility to adapt to the technological changes and independent resource management for different system components. Here, we report our experiences and lessons learned during incremental migration and architectural refactoring of a commercial Mobile Backend as a Service to microservices. We provide a detailed explanation of how we adopted DevOps and its practices, and how these practices facilitated a smooth migration process. Furthermore, we have taken a step towards devising microservices migration patterns by wrapping our experiences in different projects into reusable migration practices.

1 Introduction

According to Google Trends, both DevOps and microservices are recognized as growing concepts and interestingly with an equal rate after 2014 (cf. Figure 1). Although DevOps practices can also be used for monoliths, but microservices enables an effective implementation of the DevOps through promoting the importance of small teams [1]. Microservices architecture is a cloud-native architecture that aims to realize software systems as a package of small services, each independently deployable on a potentially different platform and technological stack, and running in its own process while communicating through lightweight mechanisms like RESTful or RPC-based APIs (e.g., Finagle). 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 including, but not limited to flexibility to adapt to the technological changes in order to avoid technology lock-in, and more importantly, reduced time-to-market, and better development team structuring around services [3].

Here we explain our experiences and lessons learned during incremental migration of the *Backtory* (http://www.backtory.com/) platform, a commercial Mobile Backend as a Service (MBaaS), to microservices in the context of the DevOps practices. Microservices helped Backtory in a variety of ways, especially in shipping new features more frequently, and in providing scalability for the collective set of users coming from different mobile application developers. Furthermore, we report on a number of migration patterns based on our observations in different migration projects. These patterns can be used by either practitioners who aim to migrate their current monolithic software systems to microservices, or system consultants who help organizations to prepare migration plans for adopting the DevOps practices in the migration process towards the microservices. To save space, the details of these migration patterns are provided in a supplementary technical report [4].

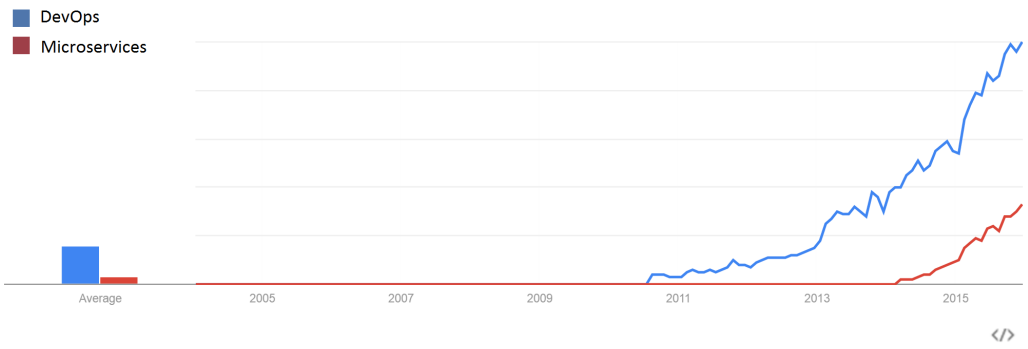

Figure 1: Google Trends report for DevOps and Microservices keywords

DevOps and Microservices (SIDEBAR) *DevOps* is a set of practices [1] which not only aims to decrease the time between applying a change to a system and the change being transferred to the production environment, but also insists on keeping the software quality in terms of both code and the delivery mechanism as one of the key elements in the development process. Any techniques that enables the mentioned goal is considered as a DevOps practice [1, 5].

Continuous Delivery (CD) [6] is a DevOps practice that enables on-demand deployment of a software to any environment through the automated machinery. CD is an essential counterpart for microservices as the number of deployable units increases. Yet another critical DevOps practice is the Continuous Monitoring (CM) [7] which not only provides developers with performance-related feedbacks but also facilitates detecting any operational anomalies [5].

2 The Architectural Concerns for Microservices Migration

2.1 The Architecture of Backtory Before the Migration

Backtory is a commercial MBaaS platform that has been developed in PegahTech Co. (http://www.pegahtech.ir/). Backtory provides mobile developers who does not know any server-side programming languages with backend services. The original functionality of Backtory was a RDBMS as a Service. Developers could define their database schemas in the Backtory's developer dashboard, and Backtory would provide them an SDK for their desired target platform (e.g., Android or iOS). Afterwards, the developers can only code in their desired platforms using their domain objects, and the objects would make some service calls on their behalf in order to fulfill their requests. Over time, new services are being added to Backtory like Chat as a Service, Indexing as a Service, NoSQL as a Service, and so on.

Backtory is written in Java using the Spring framework. The underlying RDBMS is an Oracle 11g. Maven is used for fetching dependencies and building the project. All of the services were in a Git repository, and the modules feature of Maven was used to build different services. The deployment of services to development machines was done using the Maven's Jetty plugin. However, the deployment to the production machine was a manual task. The architecture of Backtory before the migration to microservices is illustrated in Figure 2(a). In this figure, solid arrows and dashed arrows respectively illustrate the direction of service calls and library dependencies. Figure 2(a) also demonstrates that Backtory consisted of five major components, see Appendix A

2.2 Why Did We Plan to Migrate Towards the Microservices in the First Place?

What motivated us to perform a migration to a microservices architecture was an issue raised with a requirement for a Chat as a Service. To implement this requirement, we chose *ejabberd* due to its known built-in scalability and its ability to run on clusters. To this end, we wrote a *python* script that enabled ejabberd to perform authentications using our Backtory platform. The big issue in our service was the *on-demand* capability. This issue made us to perform a series of actions which resulted in raising new motivations for migration to microservices:

- *The need for reusability:* To address the on-demand capability, we started to automate the process of setting up a chat service. One of these steps was to spin off a database for each user. In our system, there was a pool of servers, each of which had an instance of the Oracle DBMS installed and an instance of DeveloperServices running. During the RDBMS instantiation, a server was selected randomly and related users and table-spaces were created in the Oracle server. This had several issues since it was just designed to fulfill the RDBMS service needs, and it was tightly coupled to the Oracle server. Consequently, we needed a database reservation system that both of our services could use.

- *The need for decentralized data governance:* Another issue was that every time anyone intended to add some metadata about different services, they were added to DeveloperData. This was not a good practice since 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 was growing, another problem was to automate the deployment process and to decouple the build life cycle of each service from others.

- *The need for built-in scalability:* The vision of Backtory is to serve millions of users. By increasing the number of services, we needed a new approach for handling such kind of scalability because scaling services individually requires major efforts and can be error-prone if not handled properly. Therefore, our solution was to locate service instances dynamically through the Service Discovery [4] component and balancing the load among them using an internal Load Balancer component.

2.3 The Target Architecture of Backtory After the Migration

To realize microservices architecture and to satisfy our new requirements, we transformed the Backtory's core architecture to a target architecture by performing some refactorings. These changes included introducing microservices-specific components and re-architecting the system.

In the state-of-the-art about microservices [8, 9], *Domain Driven Design* and *Bounded Context* are introduced as common practices to transform the system's architecture into microservices [10]. As we did not have a complex domain, we decided to re-architect the system based on domain entities in the DeveloperData. The final architecture is depicted in Figure 2(g).

The new technology stack for Backtory was the Spring Boot for embedded application server and fast service initialization, the operating system's environment variables for configuration, and the Spring Cloud's Context and the Config Server for separating the configuration from the source code as recommended by the CD practices. Additionally, we chose the Netflix OSS for providing some of the microservices-specific components, i.e., Service Discovery [4], and the Spring Cloud Netflix that integrates the Spring framework with the Netflix OSS project. We also chose Eureka for Service Discovery [4], Ribbon as Load Balancer, Hystrix as Circuit Breaker [11] and Zuul as Edge Server [4], that all are parts of the Netflix OSS project. We specifically chose Ribbon among other load balancers, e.g. HAProxy, because of its integration with the Spring framework and other Netflix OSS projects, in particular Eureka.

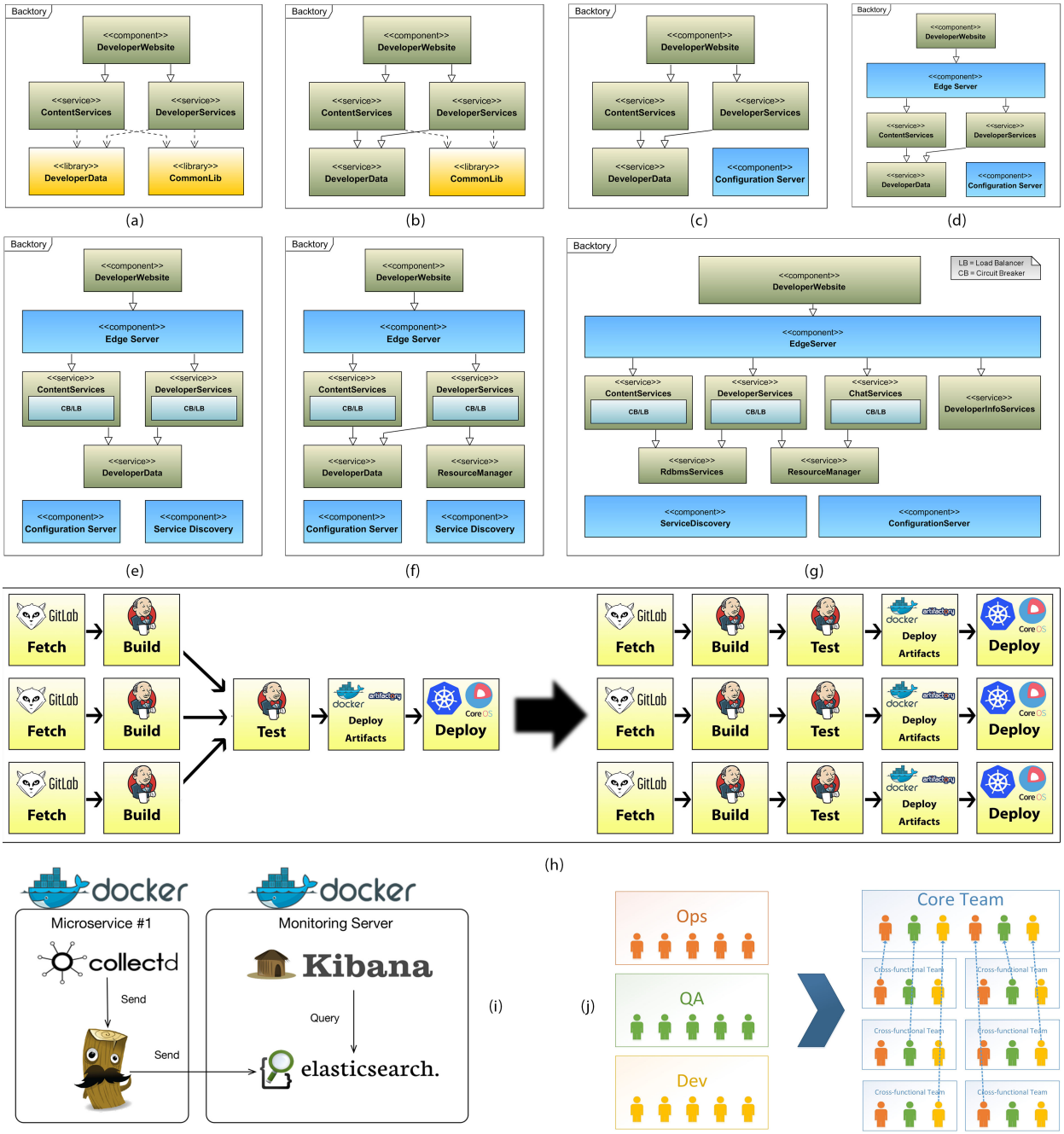

Figure 2: (a) The architecture of Backtory before the migration, (b) Transforming DeveloperData to a Service, (c) Introducing Configuration Server, (d) Introducing Edge Server, (e) Introducing Dynamic Service Collaboration, (f) Introducing ResourceManager, (g) Target architecture of Backtory after the migration, (h) The final delivery pipeline, (i) The monitoring and performance feedback infrastructure, (j) DevOps team formation

3 The Migration Process

The changes we made throughout the migration process included (i) architectural refactorings and (ii) some necessary changes to enable DevOps.

3.1 Architectural Refactoring

Migrating the system towards the target architecture was not a one-step procedure and we performed it incrementally without affecting the end-users. We treated the migration steps as architectural changes (adding or removing components) that consists of two states: (i) before the migration, and (ii) after the migration.

3.1.1 Preparing the Continuous Integration Pipeline

Continuous Integration (CI) is the first step towards CD. It allows developers to integrate their work with the others' early and on a regular basis, and helps to prevent future conflicts [6]. To this end, a CI server, an as-a-service or self-hosted code repository, and an artifact repository are needed. We chose Jenkins as the CI server, self-hosted Gitlab as the code repository, and Artifactory as the artifact repository (cf. Figure 2(h)).

Since services can have a number of instances running, deploying microservices using virtualization is not cost-effective and introduces a heavy computational overhead. Furthermore, we needed to use the Configuration Management systems in order to create the production and test environments.

By utilizing containers, we could deploy service instances with lower overheads than the virtualization, and with a better isolation. Another major benefit is the *portability* since we could deploy anywhere that supports containerization without any changes to our source codes or container images. Docker is a tool for the containerization of applications [12]. As we were going to use Docker, we needed the Docker Registry to be in our pipeline as well.

To summarize, in this step, we integrated the Gitlab, Jenkins, Artifactory, and Docker Registry as a CI pipeline. As can be seen in Figure 2(h), the fundamental difference between this delivery pipeline with a monolithic one is in the existence of *independent pipeline delivery* for each service, thereby each of them can be deployed independently. Previously, we were using integration tests which needed running the whole set of tests in case of a change in just one service. We replaced integration tests with consumer-driven contracts [13] and tests which leaded to independent testing of each service using expectations of their consumers. This change minimizes the inter-team coordination which despite of a more complex testing strategy, enables *forming smaller teams* as a DevOps practice.

3.1.2 Transforming DeveloperData to a Service

In this step, we changed the DeveloperData to use Spring Boot because of its advantages (see Section 2.3). Furthermore, as shown in Figure 2(b), we changed it to expose its functionalities as a RESTful API. In this way, its dependent services would not be affected when the internal structure of DeveloperData changes. Since they have service-level dependency, the governance of DeveloperData entities will be done by a single service and DeveloperData would not act as an Integration Database [14] for its dependent services anymore. Accordingly, we adapted DeveloperServices and ContentServices to use DeveloperData as a service and not as a Maven dependency.

3.1.3 Introducing Continuous Delivery

One of the best practices in the CD is to separate the source code, the configuration, and the environment specification so that they can evolve independently [6]. In this way, we can change the configuration without

redeploying the source code. By leveraging Docker, we removed the need for specifying environments since the Docker images produce the same behavior in different environments. In order to separate the source code and the configuration, we ported every service to Spring Boot and changed them to use the Spring Cloud Configuration Server and the Spring Cloud Context for resolving their configuration values (cf. Figure 2(c)). In this step, we also separated services' code repositories to have a clearer change history and to separate the build life cycle of each service. We also created the Dockerfile for each service that is a configuration for creating Docker images for that service. We then created a CI job per service and ran them to populate our repositories. Having the Docker image of each service in our private Docker registry, we were able to 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.

3.1.4 Introducing Edge Server

As we were going to re-architect the system and it was supposed to change the internal service architecture, we introduced Edge Server [4] to the system to minimize the impact of internal changes on end-users as shown in Figure 2(d). Accordingly, we adapted the DeveloperWebsite.

3.1.5 Introducing Dynamic Service Collaboration

In this step, we introduced the Service Discovery [4], Load Balancer, and Circuit Breaker [11] to the system as shown in Figure 2(e). Dependent services should locate each other via the Service Discovery [4] and Load Balancer; and the Circuit Breaker [11] will make our system more resilient during the service calls. By introducing these components to the system, we made our developers more comfortable with these new concepts, and it accelerated the rest of the migration process.

3.1.6 Introducing ResourceManager

In this step, we introduced the ResourceManager by factoring out the entities that were related to servers, i.e., the AvailableServer, from the DeveloperData and introducing some new features, i.e., MySQL database reservation, for satisfying our chat service requirements (cf. Figure 2(f)). Accordingly, we adapted the DeveloperServices to use this service for database reservations.

3.1.7 Introducing ChatServices and DeveloperInfoServices

As the final step in the architectural refactoring, we introduced the following services (see the target architecture in Figure 2(g)):

- The DeveloperInfoServices by factoring out developer related entities (e.g., the Developer) from the DeveloperData.
- The ChatServices for persisting chat service instances metadata and handling chat service instance creations.

3.1.8 Clusterization

In this step, we set up a cluster of CoreOS instances with Kubernetes agents installed on them. We then deployed our services on this cluster instead of a single server. As can be seen in Figure 2(h), independent testing of services using *consumer-driven tests* enabled us to also deploy each service independently. Hence, the occurrence of a change in a service would not result in the redeployment of the whole system anymore.

3.2 Crosscutting Changes

3.2.1 Filling the Gap Between the Dev and Ops via Continuous Monitoring

In the context of microservices, each service can have its own independent monitoring facility owned by the Ops team and thereby, enables independent flow of per service performance information to the development. Appropriate parametric performance models can be adopted in order to provide a good estimate of the end-to-end system performance or to facilitate what-if analyses. This helps the Dev team to refactor the architecture in order to remove the performance bottlenecks [5].

As shown in Figure 2(i), the microservices monitoring solution we used consists of both of the client and server containers. The server container takes care of the actual monitoring tools. In our deployment, it contains Kibana for visualization and Elasticsearch for consolidation of the monitoring metrics. Through the capabilities of Elasticsearch, one can horizontally scale and cluster multiple monitoring components. The client container contains the monitoring agents and the facilities to forward the data to the server. In this particular instance, it contains the Logstash and the collected modules. The Logstash connects to the Elasticsearch cluster as the client and stores the processed and transformed metrics data there. Note that with this architecture, we are able to monitor each microservice independently and react to any anomalies that we can uncover based on the online monitoring data. For detecting anomalies, we used a statistical model that we trained using the monitoring data in normal situations and then for each new incoming monitoring data point, the anomaly detection module calculates a score, with the help of principle component analysis, to spot outliers.

3.2.2 Changing Team Structures

Traditional software methods encourage horizontal division of project members to functionally separated teams. This division normally causes the creation of Dev, QA, and Ops teams (cf. Figure 2(j)). This type of separation delays the development life cycle in transition between teams and through various reactions which exist regarding the change frequency in different teams. Moreover, in the microservices setting, as each team would be responsible for their own services, they cannot benefit from the higher comprehensibility of the code and easier joining of new team members which is caused by the decomposition of the system.

In contrast, DevOps recommends vertical dividing of project members into small cross-functional teams which also fits microservices well. Each team is responsible for a service and consists of people with different skills, like development and operations skills, and they cooperate from the beginning of the project to create more value for the end-users of that particular service through more frequent release of new features to the production, and hence, remove the transition overheads which were existed in the horizontal team formation. Furthermore, as each team is focused on a particular service, the maintainability and comprehensibility of each service's code is much higher, and new members can be added to the teams with a lower learning curve.

During the migration, as depicted in Figure 2(j), we gradually formed small cross-functional teams for each new service constructed as a result of architectural refactorings. Furthermore, we formed a core team consisted of representatives of each service's team which is responsible for shared capabilities. This team has an overall view of the service interactions in the system and are in charge of any critical architectural decision. Additionally, any inter-service refactorings which involves transferring of functionalities between services and updating the corresponding rules in the Edge Server [4] are handled through this team.

4 Lessons Learned

Migrating an on-premise application to a microservices architecture is a non-trivial task. During this migration, we faced several challenges that we were able to solve. In the following, we share some of the lessons

we have learned in this process that we think might be helpful for others who are also trying to migrate to microservices:

- *Deployment in the development environment is difficult:* It is true that the application's code is now in isolated services, however, to run those services in their machines, developers need to deploy the dependent services as well. We had this problem since the Introducing Dynamic Service Collaboration step. In our case, we chose the 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.

- *Service contracts are critical:* Changing so many services that only expose their contracts to each other could be an error-prone task. Even a small change in the contracts can break a part of the system or even the system as a whole. Service versioning is a solution. Nonetheless, it could make the deployment procedure of each service even more complex. Therefore, people usually do not recommend service versioning for microservices. Thus, techniques like Tolerant Reader [13] are more advisable in order to avoid service versioning. Consumer-driven contracts [13] could be a great help in this regard, as the team responsible for the service can be confident that most of their consumers are satisfied with their service.

- *Distributed system development needs skilled developers:* Microservices is a distributed architectural style. Furthermore, in order for it to be fully functional, it needs some supporting services like service discovery, load balancer, and so on. At the early steps of migration, we tend to spend a lot of time for describing these concepts and their corresponding tools and libraries to novice developers, and still, there were many situations that they misuse them. Hence, to get the most out of microservices, those team members are needed who are familiar with these concepts and are comfortable with this type of programming.

- *Creating service development templates is important:* Polyglot persistence and the usage of different programming languages are promises of microservices. Nevertheless, in practice, a radical interpretation of these promises could result in a chaos in the system and make it even unmaintainable. As a solution, from the beginning of architectural refactoring steps, we started to create service development templates. We have different templates for creating a microservice in Java using different data stores which includes a simple sample of a correct implementation. Recently, we are also working on creating templates for NodeJS. One simple rule is that each new template, should be first considered by a senior developer for identifying potential challenges.

- *Microservices is not a silver bullet:* Microservices was beneficial for us because we needed that amount of flexibility in our system, and that we had the Spring Cloud and Netflix OSS that made our migration and development a lot easier. However, as mentioned before, by adopting microservices several complexities would be introduced to the system that require a considerable amount of effort to resolve them.

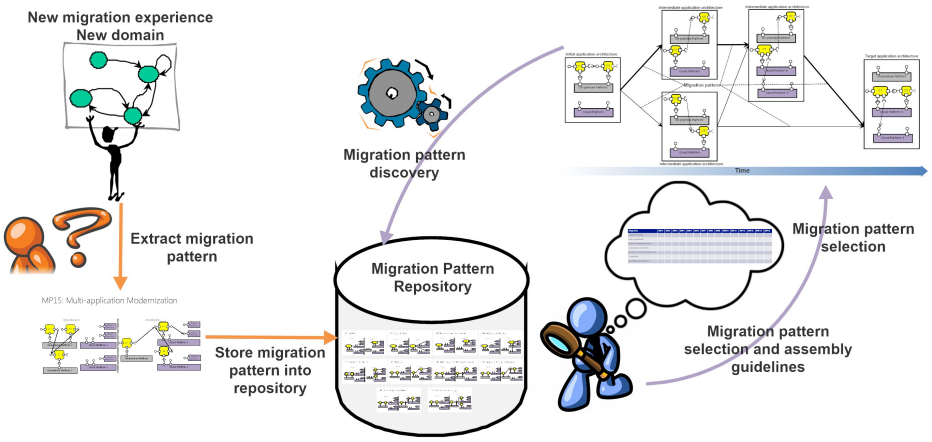

Figure 3: The process of selecting patterns, instantiating and composing a migration plan as well as extending the repository.

5 Microservices Migration Patterns

After conducting the migration project described so far in this article, we decided to make our experiences and best practices more accessible for other similar projects by abstracting them as *migration patterns*. In this way, these practices can be reused to create a migration plan by instantiating and composing the patterns.

Migrating to cloud, and specifically migrating through cloud-native architectures, like microservices, is a multi-dimensional problem, and consequently, it is a non-trivial task [15]. Therefore, in the absence of a well-thought methodology, it can be transformed to a trial-and-error endeavor which may not only waste a lot of time, but also lead to a wrong solution. Furthermore, as factors like the requirements, the current situation, the skills of team members, etc., could vary in different companies and scenarios, a unique and rigid methodology would not suffice. Thus, instead of a one-fit-all methodology, we decided to choose a *Situational Method Engineering* (*SME*) [16] approach.

The first step towards the SME approach is to prepare a method base or pattern repository consisting of reusable process patterns or method chunks, each instantiated from a predefined meta-model. To this end, using our previous experience in defining migration patterns [17], we documented our experience in this project and similar practices in the state-of-the-art of microservices (see [4, 3], `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 be solved, and the possible challenges of the proposed solution forming a pattern template (see [4]). Part of these patterns relates to exactly describing why we need supporting components, e.g., Service Discovery [4], and the prerequisites for their introduction. Additionally, we provided some solutions and advices for decomposing a monolithic system to constituting services, and preparing the current and the target architectures of the system as a roadmap for migration planning. Still, we provide some hints about the containerization of services and their deployment in a cluster. The details of our patterns can be found in [4]. However, for traceability purposes, a mapping between the identified patterns and the corresponding sections in this article that describe those patterns can be found in Table 1.

As shown in Figure 3, having an initial set of these patterns available, a method engineer can construct a concrete method based on his specific requirements for the migration by applying the construction guidelines. For example, following the need for *polyglotness*, one can reach the decomposition patterns. Then, with respect to her specific needs, she can select a suitable pattern.

Table 1: Patterns and Sections map

Identity	Pattern Name	Section	DevOps Impact
MP1	Enable the Continuous Integration	3.1.1	CI is the first step towards CD which is a known DevOps practice
MP2	Recover the Current Architecture	2.2	Enables the decomposition of the system to smaller services which leads to smaller teams
MP3	Decompose the Monolith	2.3	
MP4	Decompose the Monolith Based on Data Ownership	2.3	
MP5	Change Code Dependency to Service Call	3.1.2	
MP6	Introduce Service Discovery	3.1.5	Dynamic discovery of services removes the need for manual wiring, thereby advocates more independent deployment pipelines
MP7	Introduce Service Discovery Client	3.1.5	
MP8	Introduce Internal Load Balancer	3.1.5	
MP9	Introduce External Load Balancer	3.1.5	
MP10	Introduce Circuit Breaker	3.1.5	Failing fast can decrease the coupling between services and thereby, contributes to the independency of services' deployment
MP11	Introduce Configuration Server	3.1.3	Separating configuration from code is a CD best practice, and CD is a known DevOps practice
MP12	Introduce Edge Server	3.1.4	Edge Server not only allows Dev to more easily change the internal structure of the system but also permits Ops to better monitor the overall status of each service
MP13	Containerize the Services	3.1.3	Containers can produce the same environment in both production and development, thus, reduces the conflicts between Dev and Ops
MP14	Deploy into a Cluster and Orchestrate Containers	3.1.8	Cluster management tools reduce the difficulties around deployment of many instances from different services in production, thus, reduces the resistance of Ops towards Dev changes
MP15	Monitor the System and Provide Feedback	3.2.1	Performance monitoring enables systematic collection of performance data and sharing to enhance decision making. For example, the Dev team can use such information to refactor the architecture if they find out there exists a performance anomaly in their system

The architectural refactorings as the result of pattern applications cannot occur in an ad hoc manner. There exists some invariants that should be satisfied during the architectural transition [18]. Keeping the system in a stable state after applying a pattern, performing one architectural change at a time, and keeping the end-users of the system unaffected are the most important invariants which should be taken into account during an architectural change. Although a single step itself needs to conform to these invariants, the steps and their execution order collectively can violate them, and therefore, the method engineer should consider this during the migration pattern selection. In the migration process of Backtory, we have introduced the Edge Server before introducing components related to dynamic collaboration between services for making all of the following changes transparent to end-users. If the order of these steps have been changed, we could not satisfy some of the mentioned invariants.

Each of the future migrations can contribute to the repository of patterns as well. This can be done via introducing new migration patterns that were lacking before and were used in that particular project. This repository will serve as an extensible source for the DevOps community through which they can reuse patterns for migrating towards microservices, like the one for architectural patterns at http://microservices.io/.

5.1 How the Microservices Migration Patterns Enable DevOps

Traditional methods for software development advocate separated Dev and Ops team in which the Dev team provides the Ops team with deployment artifacts and details. The problem is that these teams have different behaviors regarding the frequency of changes such that the Dev team tends to produce more changes and the Ops team insists on higher stability. Furthermore, due to the existence of large teams working on monolithic systems, any changes needs a lot of coordination. Even with the componentization of the system, the final integration needs that amount of coordination, and the problem is still present. These issues collectively delays the whole development life cycle.

DevOps together with microservices are tackling the above mentioned issues via providing the necessary equipments for minimizing the coordination amongst the teams responsible for each component, and removing the barriers for an effective and reciprocal relationship between the Dev and the Ops teams. Indeed, in the DevOps setting, these teams help each other through continuous valuable feedbacks. To elaborate further, in Table 1, we briefly describe the impact of our devised patterns on the DevOps and the problems they tackle.

Appendix A Backtory Components Before the Migration

Backtory consisted of the following five components before the migration:

- *CommonLib:* This is a place for putting shared functionalities, like utility classes, that are going to be used by the rest of the system.
- *DeveloperData:* This holds the information of developers who are using the Backtory service and their domain model metadata entities that are shared between the DeveloperServices and the ContentServices components.
- *DeveloperServices:* This is where the services related to managing the domain model of developers' projects reside in. Using these services, developers could add new models, edit existing ones, and so on.
- *ContentServices:* This holds the services that the target SDK is using to perform the CRUD operations on the model's objects.
- *DeveloperWebsite:* This is an application written in HTML and JQuery and acts as a dashboard for developers. For this purpose, it leverages the DeveloperServices component.

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." http://martinfowler.com/articles/microservices.html, March 2014. [Last accessed 3-October-2015].

[3] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Migrating to cloud-native architectures using microservices: An experience report," in *In Proceedings of the 1st International Workshop on Cloud Adoption and Migration*, September 2015.

[4] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices migration patterns," Tech. Rep. TR-SUT-CE-ASE-2015-01, Automated Software Engineering Group, Sharif University of Technology, Tehran, Iran, October 2015. [Available Online at http://ase.ce.sharif.edu/pubs/techreports/TR-SUT-CE-ASE-2015-01-Microservices.pdf].

[5] A. Brunnert, A. van Hoorn, F. Willnecker, A. Danciu, W. Hasselbring, C. Heger, N. R. Herbst, P. Jamshidi, R. Jung, J. von Kistowski, A. Koziolek, J. Kroß, S. Spinner, C. Vögele, J. Walter, and A. Wert, "Performance-oriented devops: A research agenda," *CoRR*, vol. abs/1508.04752, 2015.

[6] J. Humble and D. Farley, *Continuous delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[7] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst, "Continuous monitoring of software services: Design and application of the kieker framework," research report, Kiel University, November 2009.

[8] S. Newman, *Building Microservices*. O'Reilly Media, 2015.

[9] M. Stine, *Migrating to Cloud-Native Application Architectures*. O'Reilly Media, 2015.

[10] V. Vernon, *Implementing Domain-driven Design*. Addison-Wesley Professional, 2013.

[11] M. Nygard, *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2007.

[12] C. Pahl, "Containerization and the paas cloud," *IEEE Cloud Computing*, no. 3, pp. 24–31, 2015.

[13] R. Daigneau, *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Addison-Wesley Professional, 2011.

[14] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2004.

[15] P. Jamshidi, A. Ahmad, and C. Pahl, "Cloud migration research: A systematic review," *IEEE Transactions on Cloud Computing*, vol. 1, pp. 142–157, July 2013.

[16] B. Henderson-Sellers, J. Ralyt, P. Agerfalk, and M. Rossi, *Situational Method Engineering*. Springer-Verlag Berlin Heidelberg, 2014.

[17] P. Jamshidi, C. Pahl, S. Chinenyeze, and X. Liu, "Cloud migration patterns: A multi-cloud architectural perspective," in *Proceedings of the 10th International Workshop on Engineering Service-Oriented Applications*, 2014.

[18] A. Ahmad, P. Jamshidi, and C. Pahl, "Classification and comparison of architecture evolution reuse knowledgea systematic review," *Journal of Software: Evolution and Process*, vol. 26, no. 7, pp. 654–691, 2014.

About the Authors

Armin Balalaie is a master's student at the Sharif University of Technology. His research interests include software engineering, cloud computing, and distributed systems. Contact him at armin.balalaie@gmail.com.

Abbas Heydarnoori is an assistant professor at the Sharif University of Technology. Before, he was a post-doctoral fellow at the University of Lugano, Switzerland. Abbas holds a PhD from the University of Waterloo, Canada. His research interests focus on reverse engineering and re-engineering of software systems, mining software repositories, and recommendation systems in software engineering. Contact him at heydarnoori@sharif.edu.

Pooyan Jamshidi is a post-doctoral research associate in the Department of Computing at Imperial College London, UK. Pooyan holds a PhD in Computing from Dublin City University, Ireland. His primary research interest is in the areas of self-adaptive software, where he applies statistical machine learning and control theory to enable self-organizing behaviors in distributed systems which are designed to process big data. Contact him at p.jamshidi@imperial.ac.uk.