Microservices: Architecting for Continuous Delivery and DevOps

Lianping Chen
Lianping Chen Limited
Dublin, Ireland
lianping.chen@outlook.com

***Abstract*—Businesses today need to respond to customer needs at unprecedented speeds. Driven by this need for speed, many companies are rushing to the DevOps movement and implementing Continuous Delivery (CD). I had been implementing DevOps and CD for Paddy Power, a multi-billion-euro betting and gaming company, for four years. I had found that software architecture can be a key barrier. To address the architectural challenges, we tried an emerging architectural style called Microservices. I have observed increased deployability, modifiability, and resilience to design erosion. At the same time, I also observed new challenges associated with the increased number of services, evolving contracts among services, technology diversity, and testing. I share the practical strategies that can be employed to address these new challenges, discuss situations for which Microservices may not be a good choice, and outline areas that require further research.**

Keywords—Continuous Delivery; Continuous Deployment; DevOps; Architecture; Microservices; Agile; SOA

I. INTRODUCTION

Businesses today need to respond to customer needs at unprecedented speeds [1]. Driven by this need for speed, a movement called DevOps, which aims at establishing a culture and environment in which software can be built and released quickly and reliably, has rapidly gained traction.

A key approach used in this DevOps movement is Continuous Delivery (CD) [2]. CD is a software engineering approach in which teams produce valuable software in short cycles and ensure that the software can be reliably released at any time [3]. Companies that have adopted CD have reported huge benefits [3], which have motivated more companies to adopt CD.

However, implementing CD can be very challenging [3, 4]. Based on my experiences in adopting CD over four years at a multi-billion-euro betting company, I found that software architecture can be a key barrier. However, discussions of CD have primarily focused on build, test, and deployment automation [5]; insufficient attention has been paid to software architecture.

To address the architectural challenges, we have tried an emerging architectural style called Microservices. In this article, I report on the main benefits we have achieved from using Microservices, describe the new challenges that have arisen from its use, present the practical approaches that can be employed to address these new challenges, discuss situations for which Microservices may not be a good choice, and outline areas that require further research.

II. THE CONTEXT

A. DevOps and CD at Paddy Power

Paddy Power was a rapidly growing company with approximately 5,000 employees and a turnover of €7 billion. It merged with Betfair in 2016, making it the world's largest public online betting and gaming company.

Paddy Power offers its services in regulated markets through betting shops, phones, and the Internet. The company relies heavily on several hundreds of custom software applications. These applications are developed using a wide range of technology stacks, including Java, Ruby, PHP, JavaScript, .NET, and C++. The diversity of applications and technology stacks results in diverse application architectures, which provided me with the opportunity to observe the role of software architecture in DevOps and CD adoption.

The adoption of DevOps and CD began in 2012. The CIO championed the cultural change and revamped the organizational structure to promote cross-functional collaboration both between and within teams. A team of eight people (our team) was established to move applications to CD.

In four year's time, we had implemented CD for more than 60 different applications and services. For each of these, when a developer commits changed code, a CD pipeline execution is automatically triggered. The code change passes through a series of stages that check the quality of this code change. If the change fails to pass any of the pipeline stages, the pipeline process is aborted, and the developers are notified to fix the problems. If the change passes all these stages, then the change can be released to production with the click of a button.

B. Architectural Challenge

Over this four-year journey, we had investigated many applications in the company and had attempted to move them to CD. As I reported in my previous work [6], an application's architecture can be a key barrier.

To better understand the characteristics that make an application amenable to CD, we compared the architectural characteristics of the applications that were easily moved to CD with those that were difficult to move to CD [6]. A set of

978-1-5386-6398-1/18/$31.00 ©2018 IEEE
DOI 10.1109/ICSA.2018.00013

characteristics, in the form of Architecturally Significant Requirements (ASRs) [7], was identified [6]. Of those, deployability and modifiability often shape architectural decisions.

1) Deployability: A level of deployability that is acceptable for a traditional multi-month release model is no longer acceptable for CD. When releasing twice a year, we can take the system down for a release, assign a significant task force to perform the deployment, and be less concerned with the actual time of the deployment as long as it completes within some time window.

However, all of the above are unacceptable with the high frequency of releases in CD. To maintain system availability, reliable zero-downtime deployments are required. To handle the large number and high frequency of deployments in both testing and production environments, deployments must be automated. A deployment must also be fast to allow the execution of the CD pipeline to finish within an acceptable amount of time, so that developers can obtain prompt feedback regarding the production readiness of a change.

2) Modifiability: Deployability is only one aspect of increasing the speed of software delivery. The modifiability of an application itself is also important. Although modifiability is also a concern when architecting an application in a traditional context, the level of modifiability required in CD is often higher. This is because in CD, the delivery process is no longer a bottleneck, and thus, the speed at which the software development team can make modifications becomes the factor that determines the speed of software delivery. To compete effectively, the business must increase the modifiability of the application itself to respond more quickly to customer needs and requests.

Thus, in CD, deployability and modifiability have become more demanding, and we can no longer trade them off lightly.

C. The Use of Microservices

Driven by the increased priorities of deployability and modifiability and the concomitant increased degree of requirements on these quality attributes, we moved from a monolithic architecture to a Microservices architecture [8]. Rather than building large monolithic applications, we organize systems into small, self-contained, single-responsibility units that can be independently developed, tested, and deployed.

We decided to try Microservices for two practical reasons:

- We had tried to build well-modularized monolithic applications, but they often eventually evolved into "big balls of mud". Consequently, it became necessary to try a new approach.
- Microservices looked promising in bringing us the levels of deployability and modifiability that we needed.

The team size per service ranges from two to five people. We do not have strict rules on the lines of code per service but a rule of thumb is that each service must to be kept to a size that a single engineer can comprehend.

III. Observed Benefits of Microservices

After we moved our applications to a Microservices architecture, we observed increased deployability, modifiability, and resilience to architecture erosion, as shown in Fig. 1.

A. Increased Deployability

We have observed the following deployability improvements, which are essential for achieving CD, as discussed above.

1) Deployment Independency: Prior to using Microservices, multiple teams worked on the same code base of a large monolithic application. When one team made a change, that team was unable to release it independently. Instead, they first needed to merge the change into the master branch. To avoid the chaos caused by multiple simultaneous merges, they often needed to queue the merge. The merge could often involve conflicts. Resolving these conflicts required coordination with other teams. Then, after the merge, the entire application needed to be re-tested to find any new errors that might have been introduced. Finally, the decision to actually make the release was beyond the team's control; because the entire application was released as one unit, thus, many other teams' changes also needed to be considered when making the release decision.

After the application has been moved to Microservices, there is no need to queue for merges; no need to resolve merge conflicts with other teams; and no need to coordinate with and wait for other teams' changes. Finally, each team can deploy their changes independently.

2) Shorter Deployment Time: Prior to using Microservices, a deployment needed to deploy all components of a large monolithic application. Now, the monolith has been broken down into many smaller services. Each service is significantly smaller than the corresponding former monolith. Deploying such a small service takes a shorter amount of time than that for the corresponding former monolith. For example, one of the applications has been broken down into 11 microservices. The time for deploying a change has reduced from 30 minutes to 3 minutes.

This shorter deployment time does make a difference. For most pipelines, the software change will be deployed to four different testing environments (i.e., four deployments). Consequently, the reduction in deployment time helps to shorten the pipeline execution time. This shortened pipeline execution time is important for practicing CD. First, the pipeline duration determines the minimum time with which a code change can be released to production, which is particularly critical when we need to release an important software fix to production. Second, our team leads reported that the pipeline duration impacts the developers' code commit behavior. The shorter that time is, the more frequently the developers commit code.

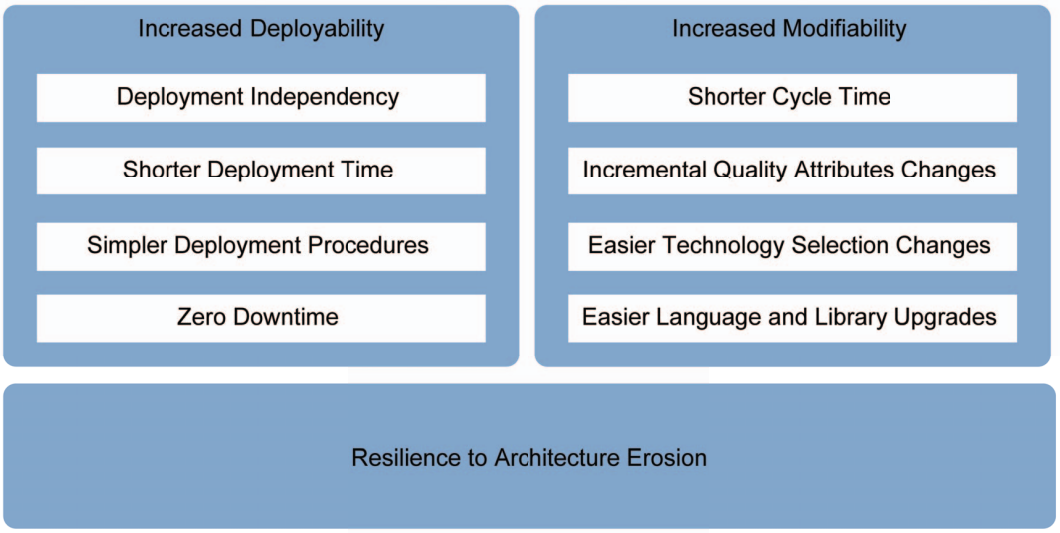

Fig. 1. Benefits of Microservices

3) Simpler Deployment Procedures: The procedures to deploy a large monolithic application are generally more complex than the procedures to deploy a microservice. A large monolith often has some peculiarities in its deployment. Because of these peculiarities and the complexity, it is often too difficult (or too costly) to build a common tool chain that can cater to the needs of all large monolithic applications.

Even worse, in some cases, building a tool dedicated solely to a particular big monolith could be difficult. For example, the company had attempted to automate the deployment of a complex monolith. Although the company invested more than a year's effort for a team of four people, the deployment was still not completely automated.

In contrast, after the application has been moved to Microservices, each service is small, which makes its deployment procedures simpler than that of the large monolith, and consequently, it is easier to develop deployment automation that completely eliminates all manual activities.

More importantly, building a common tool chain that can handle any microservice becomes feasible. After we built this tool chain, adding a new microservice no longer requires a new tool. Thus, this is a more scalable approach.

Developing the tool chain for CD is a formidable task, as reported by other researchers as well [9]. Thus, simplifying the tool chain requirements (e.g., simplifying the deployment procedures for the software) has a significant impact on the cost and feasibility of the tool chain development.

4) Zero Downtime: We were unable to release most of our large monolithic applications in a zero-downtime manner. One common barrier is associated with the database. For example, one of our large monolithic applications had about 100 database tables with complex interrelationships. Almost every release included some database schema changes. Due to the complexity of the database schema and the fact that each release contained changes from multiple development teams, the best available database migration tools could not handle all the scenarios [10], often the teams were confident in performing database migrations only when the application was shut down.

After the application has been moved to Microservices, we can choose the most suitable database for each individual service. It turned out only a small portion of the services actually required a relational database. Thus, all the other services use a NoSQL database, which is more amenable to schema evolution. Although we still need to address database schema evolution for services that require a relational database, because the schema for a microservice is considerably smaller and under the full control of the small team that owns the microservice, the team can design the schema and plan the changes in a way that the existing database migration tools can handle. Thus, it is easier to achieve zero-downtime releases with Microservices than with our former large monolithic applications.

B. Increased Modifiability

1) Shorter Cycle Time for Small Incremental Functional Changes: For most small incremental functional changes, we have reduced the cycle time from multiple months to two to five days. Although we cannot attribute all the cycle time reduction to the use of Microservices, the use of Microservices is a factor to achieve such short (two to five days) cycle time.

a) Faster decision making: With our large monolithic applications, many design decisions needed to be discussed and approved by many teams working on the application

because, after all, all the code was in the same code base. Although the time required of each person contributing to the decision may not be significant, reaching a mutual decision across these teams often took a long calendar time (days or weeks).

With Microservices, as long as a team does not change the contracts among services, a decision can be made within the small service team of two to five people rather than among the 50-odd people working on a large monolithic application. Often, the decision can be made with only a short chat among the team members.

b) Shorter communication path from customers to the responsible developers: I also noticed that the communication path for a piece of requirement to reach the responsible developers is shorter than that with a big monolithic application. A person that is responsible for a system that has been moved to Microservices said that, "for many things that people used to call me, they no longer call me anymore, they call the person responsible for the relevant microservice directly." This is probably because a microservice has a higher visibility – which comes from the fact that the service is developed, released, and operated by an autonomous team – than a module of a monolith. The shorter path of communication helps to shorten the cycle time.

2) Incremental Quality Attribute Changes: We are not only able to make incremental functional changes but also to make incremental changes to quality attributes.

For example, we had a monolithic application called RMP [1]. We broke RMP down to more than fifteen microservices.

As the business grows, the company offer betting opportunities for increasingly more sports events. The company had also attracted an increasing number of business-to-business customers, who buy price information (called odds) from the comnpany. Moreover, the company were opening businesses in more locations (e.g., in Italy, Australia, and the USA). We call each of these business customers and locations a price distribution destination (or destination for short).

After running successfully for two years, RMP was no longer able to handle the increased load introduced by the new sporting events and new destinations, requiring us to improve its scalability and performance.

We analyzed the system and identified the bottleneck, which involved only one service. When the team was building the service two years ago, they did not realize that the business could grow this fast. Thus, the service was not tested for the new, higher traffic level.

The team was able to make the changes in several weeks. These involved changing the service architecture to a Reactive Architecture and changing the programming model from imperative programming to functional programming. Although this change took longer than delivering a functional increment, it was feasible and not too difficult.

Our engineers commented that it would have been more difficult to make the change if the service were a component of a previous large monolith because they would need to consider how to scale the whole monolith. With Microservices, the change was localized to one service only; other services were not affected at all. In addition, the small size of the service also made the change easier [11].

Another interesting (and somewhat surprising) observation was that many quality attribute changes that were associated with the growth of the business did not require system-wide changes. This probably occurred because, with Microservices, we partition a system based on business capabilities. Each service is responsible for a small business domain that has only a single responsibility, so the factors that trigger the need for improved quality attributes differ from one service to another.

For example, the service responsible for displaying the betting results is affected by the number of concurrent users who view the betting results after an event. The popularity of the event together with the success of the marketing is the factor that triggers the need for improved scalability. This same factor will not directly trigger the need for improved scalability for a service that is responsible for calculating the price. Irrespective of how popular the event is, it is only one event for the price calculation service.

Certainly, this experience is not wide and long enough to make a definitive conclusion. However, I believe this is an interesting point to bring up for discussion and further investigation.

3) Easier Technology Changes: Teams can choose different technologies for implementing their services. I found that this is particularly useful when adopting a new technology. An example was the introduction of the Scala language, which is a better option for developing low-latency and high-throughput applications. A team was able to use Scala in their own service without impacting other services, which used C++. Such a change would not be technically easy if it were part of a monolithic application.

The speed with which we can experiment and adopt new technology is important to maintain our products' superiority. The impact of new technologies on our products can be as important as the new features of the products that we build in house.

4) Easier Language and Library Upgrades: Changing language and library versions often becomes exponentially more difficult as the code base expands in a monolith. For example, previously, when one part of a system required an upgrade of Java to use its new features, the team could not simply upgrade because the entire system could only use a single version of Java; upgrading only one part could break other parts of the system. Such dependencies and the required coordination with teams working on other parts of the system made upgrading difficult and painful.

[1] A surrogate name is used here for confidentiality reasons.

After we moved a system to Microservices, each team can upgrade independently because each service no longer shares the same code base, compilation process, and language runtime (e.g., Java virtual machine) with other parts of the system.

Similar to technology changes, the speed with which we can adopt new features in the new library or language versions allows us to bring new enhancements to our customers earlier than our competitors.

In addition, language and library developers are also attempting to adopt CD. The speed at which they produce new versions is increasing, which enhances the importance of making language and library upgrades easier for our applications.

C. Resilience to Architectural Erosion

I found that the strong boundaries provided by Microservices have important practical implications. One such implication is better protection against architectural erosion—provided that the teams correctly define the service boundaries initially.

We had attempted to build a well-modularized monolith, but it often eventually evolved into a "big ball of mud". The concept of modularization is decades old, and our applications were built only 10 years ago. When we started, for all of them, the goal was to build well-modularized monoliths; we never aimed to build "big balls of mud".

However, I found that the boundaries between modules in a big monolith did not tolerate the chaos of real software development projects, where people are often in a hurry. It was too tempting and too easy for someone in a hurry to cross a boundary and grab something that she needed over there and use it directly. Often, the software eventually evolved into a "big ball of mud".

Microservices make the boundaries between components physical. Each service has its own code base, its own team, and its own set of virtual machines or containers in which it runs. Therefore, taking shortcuts is considerably more difficult in Microservices. For example, some engineers commented that sometimes, under pressure, they had thought of crossing a boundary but immediately realized that doing so was no longer a shortcut because of the physical boundaries between services. Thus, Microservices provide better protection against the temptation to break boundaries under pressure for short-term implementation convenience.

Some of the initial microservices had been running for more than two years. The boundary crossing problems that used to be common in monolithic applications had not occurred. Although it may still be too early to draw a definitive conclusion, early observations are positive.

IV. Handling the New Challenges

Despite the above benefits, Microservices is not a silver bullet. Adopting Microservices introduces new complexities and challenges. Without properly managing them, we can easily run into another problematic situation.

A. Increased Number of Services

When we break down a monolithic application into many smaller microservices, we get an increased number of services, which introduces a new complexity.

Traditionally, developers hand the software over to operations engineers for deployment to production. The deployment involves many manual activities. The traditional approach is not able to handle the significantly increased number of services with frequent releases. The large number of deployment requests greatly exceeds what the operations engineers can handle.

To tackle this complexity, we built the CD platform. The CD platform provides a CD pipeline for each service. Upon each code commit, the CD pipeline automatically builds the service, runs various automated tests, provisions the environments for testing, and enables the team to release the change to production with the click of a button. This removes the operations engineers as the bottleneck. The CD platform makes handling frequent releases of a large number of services possible.

B. Evolving Interactions/Contracts among Services

With Microservices, some interactions that were formerly internal communications within an application became external communications among services. This led to more interactions among services and more contracts to manage. Not only did the number of interactions/contracts increase, but all the participants in these interactions also undergo frequent changes. The strategies that can be used to tackle this challenge are as follows.

1) Keep Interactions Simple: I found that it is important to eliminate unnecessary complexities. When new interactions and contracts among services are introduced, the teams carefully review them to ensure that any addition is truly necessary. As a heuristic, I observed that when complicated interactions occur among services, the design is often incorrect.

2) Robustness Principle: When we write code that interacts with other services, we try to follow the robustness principle: Be conservative in what you send, be liberal in what you accept from others [12].

For example, we had a service S. An interface of S had a return message with two fields, F1 and F2. This interface was being used by five other consuming services. All these services followed the robustness principle. When they received a message from S, if the message contained extra fields, they ignored the extra fields rather than failed the call as long as the required fields F1 and F2 were present. At one point, a new service needed to use S. This sixth service required a new field, F3. Because all consuming services followed the robustness principle, we were able to modify the interface by adding the new F3 field without breaking the system.

By following the robustness principle, we give the producing service some leeway to make certain contract changes without breaking compatibility.

3) Expand and Contract: Even when all services follow the robustness principle, we sometimes still need to make interface changes that could break compatibility.

To avoid breaking compatibility, rather than modifying the interface, we instead first add a new interface, migrate all consumer services to use the new interface and, finally, remove the old interface. In this way, we are able to make contract changes and still maintain service compatibility.

How do we ensure the service consumers migrate to the new interfaces, so that we can remove the old interfaces timely? The following strategies can be used.

a) Grace period: We mark the old interface as deprecated and set a grace period. After the grace period expires, if a team still uses the old interface, that team becomes responsible for any negative consequences.

b) Interface usage monitoring: To evaluate whether a deprecated interface is still receiving traffic and which teams are using it, we need a monitoring system to monitor the usage of each interface. When the service owner can validate that a deprecated interface has no traffic, the team can safely remove that interface.

c) Incentivize migration: What happens if, after the grace period, a team still uses the deprecated interface? This is a major practical challenge. Some tactics to incentivize the migration are as follows: 1) Show alerts on that team's monitoring system when they use the deprecated interface, 2) temporarily disable the interface. The consuming team often came and asked, "What happened?" The service owner often replied with "We notified everyone that we were going to turn it off after the grace period. We will give you another week to migrate. After that, we will turn it off permanently."

C. Technology Diversity

In Microservices, the traditional technical constraints for all the components of a large monolithic application to use the same technology stack disappear. Technically, each team can use whatever technology stack they prefer. This is an important feature that Microservices advocates promote.

However, using diverse technology stacks in an uncontrolled manner can lead to problems. For example, each team using a different technology stack creates knowledge silos that make moving individuals from over-staffed to under-staffed teams difficult. Moreover, each introduced technology stack requires operational overhead, which can outweigh the benefits of introducing a new technology stack.

To address these problems, we put the use of polyglot (multiple technology stacks) under proper governance. Every new technology stack that a team wants to use must undergo a review process. This practice helps to avoid the technology landscape becoming unmanageable while utilizing the flexibility of polyglot when it is really needed.

D. Testing

Microservices introduces testing challenges, which mainly arise from changes that impact interactions among services. We reduce these challenges by using the following strategies.

1) Test-first Mindset and Practices: We realized that an application's testability becomes essential. To ensure testability, the most effective strategy we found so far is enforcing the use of test-first mindset and practices. Before a user story can be moved to development, the acceptance criteria must be discussed and defined among the testers, business analysts, and developers. Test code is reviewed as rigorously as application code. For any code change, if the test coverage (both line and condition) is below 90%, the CD pipeline will fail the build.

With these test-first practices, the teams immediately feel the pain when they make a design decision that makes testing difficult. This motivates them to come up with designs that improve testability.

2) Consumer-Driven Contract Testing: The developers of a consuming service write contract tests to ensure that producing services meet their expectations. These contract tests are also run in the pipeline of the producing service. In this way, the developers of the producing service know whether their changes will break the contracts expected by their consumers. These tests provide a safety net for contract changes.

3) Always Online System Integration Test Environment: The CD pipeline includes a stage in which a service is deployed to a system integration testing environment, which is kept always online. In this environment, the service is configured to use its dependent services, which all have a deployment in this environment. Tests are executed to ensure that the service works with its dependent services. When each service is doing this, we can ensure that the entire set of services can work together.

4) Test in Production (TiP) and Monitoring: TiP was introduced. We also enhanced logging and monitoring to quickly spot anomalies in production. Therefore, even when an issue arises, we can take immediate action, which minimizes the impact of the issue. Together, these strategies give us confidence to release changes to production frequently.

V. Does Microservices Suit Every Situation?

Considering the above new complexities and challenges, does Microservices suit every situation? No, it does not, according to my experience.

Dealing with these new complexities and challenges can require significant costs. For example, building the CD platform is not a trivial task: doing so has taken a team of eight people four years, amounting to 32 person-years of effort. Therefore, if your system is simple enough that it can be comfortably managed as a monolith, it is not "worth the trouble" to use Microservices. The Microservices approach is for handling complex systems that require high speed changes.

When teams do not have sufficient domain knowledge and experience to get the service boundaries right and expect

that the boundaries will evolve dramatically over time, Microservices can be a dangerous option because refactoring service boundaries is difficult, has no tool support, and there is no successful experience reported yet. In contrast, refactoring a monolithic application is much easier, tool support exists, and such refactoring is a common practice. In this situation, it is probably a better option to start by building a monolith.

Some technical constraints can also rule out the use of Microservices. For one of the applications that require sub-microsecond latency, the network latency introduced by Microservices is unacceptable. Furthermore, when an application absolutely requires strong consistency, the difficulties in ensuring strong consistency across all micro services can outweigh the benefits obtained from Microservices.

Organizational structure and culture is also an important factor. If your organization requires numerous handoffs between requirements engineers, developers, testers, and operations engineers, things will become very difficult to manage with Microservices. To make Microservices work, we have removed the handoffs and built autonomous teams. Each team is responsible for building the service, deploying it, operating it, and is empowered to make decisions to react to user feedback.

VI. Challenges for Research

A. Taming Interactions among Microservices

By using the strategies described above, we had not encountered major issues caused by interactions among services. However, I anticipate more challenges in the future. The number of services is increasing. As the number continues to rise, manually analyzing interactions among services will become increasingly difficult. Research is required to develop techniques to automatically analyze the interactions among services, identify harmful interactions, and generate warnings when such harmful interactions are introduced.

When we need to change interactions among a large number of services, how to effectively test these changes to gain high confidence before releasing to production also requires researchers' attention. This is especially critical for applications in risk-averse regulated environments, where bugs in production can have significant repercussions.

B. Refactoring Services Boundaries

According to my experience, as a large monolithic application evolves over time, not only can the interactions among modules change but the scope and boundaries of modules can also change to cater to domain evolution, requirement changes, and to correct the wrong module separations caused by a lack of domain understanding when the modules were initially designed (this is particularly common when a team is new to a domain).

I anticipate that scope and boundaries changes will happen in Microservices as well; however, in comparison to the same situation in monolithic applications, where we have extensive experience and tool support for refactoring module boundaries, there is not much reported experience concerning refactoring service boundaries, and there is no tool support to help in refactoring service boundaries.

Therefore, research is needed to answer questions such as: How difficult it is to refactor service boundaries? What factors impact the need to refactor service boundaries? How can we minimize the need to refactor service boundaries? We also need researchers to help develop approaches and tools to help with refactoring when boundary refactoring is unavoidable.

The problem is difficult because, compared to a monolith, service boundaries are more physical, changing the boundaries becomes more difficult and more architecturally significant in terms of the cost of such changes [7].

C. Finding and Evaluating Alternative Architectural Styles

As discussed above, Microservices do not suit every situation. Thus, research is also needed to identify and evaluate alternative architectural styles for architecting applications for CD in those situations where Microservices are not suitable.

VII. Limitations

This work has several limitations. First, all the experiences and observations stem from one company; hence, they represent only one data point. Although I believe that this data point is interesting and important, it may not be representative; other organizations may have different experiences.

Second, our description is primarily qualitative. I found that obtaining accurate quantitative data concerning the observed benefits is not easy in a real-world industrial setting for a topic like this.

Third, while describing the observed benefits of Microservices, I also attempted to explain why the observed benefits can only be obtained (or are easier to obtain) with Microservices. However, the explanation is largely experience-based. More rigorous empirical studies are needed to confirm the correctness of the explanations.

Fourth, the strategies to address the challenges that arose have not been validated outside of our company; other organizations may well have faced different challenges, and different strategies may work better.

Finally, due to company security and public relations policies, I cannot disclose a clear full picture of the application landscape, the exact architecture of an application, or an exact list of applications that we moved to Microservices. However, I do not think this renders this experience sharing useless.

VIII. Related Work

To the best of our knowledge, not much research has been conducted on the topic of architecting software applications for DevOps and CD, especially in industrial settings. This is consistent with the observations of other researchers, such as Shahin, et al. [13].

My previous work presents ASRs that CD amenable applications should meet [6], but it does not tell how to meet those ASRs. Bellomo, et al. [14] provided an in-depth description of deployability. However, the description does not cover modifiability.

There is literature concerning service-oriented architecture [15], but these studies do not explicitly examine the effectiveness of Microservices.

Balalaie, et al. [16] reported their experiences when migrating to Microservices. Bass, et al. [2] presented a similar case study. Their works focus on the migration process, whereas this work focuses on the observed benefits and challenges of this new architectural style and on how to address the new challenges that arose. In addition, they did not explicitly discuss situations in which Microservices may not be a good choice.

IX. Summary

Driven by the need for speed [1], in the context of DevOps and Continuous Delivery (CD), Paddy Power turned to a new architectural style called Microservices. Increased deployability, increased modifiability, and increased resilience to design erosion have been observed.

At the same time, new challenges associated with the increased number of services, evolving contracts among services, technology diversity, and testing have been encountered.

Although we were able to address these challenges using practical strategies and achieve CD's significant benefits [3], research in taming interactions among services, refactoring service boundaries, and finding alternative architectural styles for situations where Microservices are not suitable could dramatically advance this field.

Acknowledgment

I thank my colleagues, Klaas-Jan Stol, this paper's reviewers for their help and thoughtful comments. This paper is based on my experience at Paddy Power. It represents only my own views and does not necessarily reflect those of Paddy Power.

References

[1] J. Bosch, "Speed, Data, and Ecosystems: The Future of Software Engineering," IEEE Software, vol. 33, pp. 82-88, 2016.

[2] L. Bass, I. Weber, and L. Zhu, DevOps: A Software Architect's Perspective. New York: Addison-Wesley, 2015.

[3] L. Chen, "Continuous Delivery: Huge Benefits, but Challenges Too," Software, IEEE, vol. 32, pp. 50-54, 2015.

[4] L. Chen, "Continuous Delivery: Overcoming adoption challenges," Journal of Systems and Software, vol. 128, pp. 72-86, 2017.

[5] J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. New York: Addison-Wesley, 2010.

[6] L. Chen, "Towards Architecting for Continuous Delivery," in Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on, 2015, pp. 131-134.

[7] L. Chen, M. Ali Babar, and B. Nuseibeh, "Characterizing Architecturally Significant Requirements," Software, IEEE, vol. 30, pp. 38-45, 2013.

[8] S. Newman, Building Microservices. Sebastopol, California: O'Reilly Media, 2015.

[9] S. Tony, D. Mitchell, G. Michael, W. Laurie, B. Kent, and S. Michael, "Continuous deployment at Facebook and OANDA," presented at the Proceedings of the 38th International Conference on Software Engineering Companion, Austin, Texas, 2016.

[10] J. Michael de and D. Arie van, "Continuous deployment and schema evolution in SQL databases," presented at the Proceedings of the Third International Workshop on Release Engineering, Florence, Italy, 2015.

[11] L. Chen and M. A. Babar, "Towards an Evidence-Based Understanding of Emergence of Architecture through Continuous Refactoring in Agile Software Development," in Software Architecture (WICSA), 2014 IEEE/IFIP Conference on, 2014, pp. 195-204.

[12] J. Postel, "Transmission Control Protocol," 1980.

[13] M. Shahin, M. Ali Babar, and L. Zhu, "The Intersection of Continuous Deployment and Architecting Process: Practitioners' Perspectives," presented at the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Ciudad Real, Spain, 2016.

[14] S. Bellomo, N. Ernst, R. Nord, and R. Kazman, "Toward Design Decisions to Enable Deployability: Empirical Study of Three Projects Reaching for the Continuous Delivery Holy Grail," in Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on, 2014, pp. 702-707.

[15] M. Razavian and P. Lago, "A systematic literature review on SOA migration," Journal of Software: Evolution and Process, vol. 27, pp. 337-372, 2015.

[16] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture," IEEE Software, vol. 33, pp. 42-52, 2016.