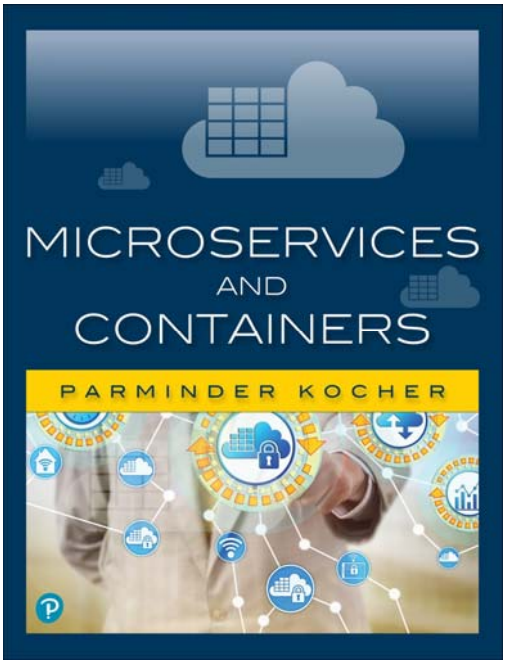
Transition to Microservices and DevOps to Transform your Software Effectiveness



Available March 30, 2018

ORDER & SAVE

SAVE 35% WHEN YOU ORDER

from **informit.com/kocher** and enter the code **INFOQ** during checkout

FREE US SHIPPING on print books

Major eBook Formats

Only InformIT offers PDF, EPUB, & MOBI together for one price

OTHER AVAILABILITY

Through O'Reilly Safari subscription service

Booksellers and online retailers including Amazon/Kindle store and Barnes and Noble/bn.com

Together, microservices and Docker containers can bring unprecedented agility and scalability to application development and deployment—especially in large, complex projects where speed is crucial but small errors can be disastrous. In **Microservices and Containers**, Parminder Singh Kocher demonstrates why and how these technologies can help you build, deploy, manage, and scale industrial-strength applications.

Learn how to leverage microservices and Docker to drive exponential improvements in DevOps effectiveness, on-demand scalability, application performance, time-to-market, code reuse, and application reliability. Kocher also offers detailed guidance and a complete roadmap for transitioning from monolithic architectures, and an in-depth case study walking through the migration of an enterprise-class SOA system.

- Understand how microservices enable you to organize applications into standalone components that are easier to manage, update, and scale
- Decide whether microservices and containers are worth your investment, and manage the organizational learning curve associated with them
- Apply best practices for interprocess communication among microservices
- Migrate monolithic systems in an orderly fashion
- Understand Docker containers, installation, and interfaces
- Network, orchestrate, and manage Docker containers effectively
- Use Docker to maximize scalability in microservices-based applications

Parminder Kocher is a lifelong technology learner with two decades of hands-on experience in building enterprise-grade software systems. He started at Cisco Systems in 2005 managing the Remote Management Service (RMS) platform, and has since worked as an innovation evangelist leading multiple software groups. Currently, he is engineering director for Cisco Networking Academy platform, where he leads the engineering teams responsible for developing the Academy's next-gen platform.

Pearson Addison-Wesley

informIT the trusted technology learning source

Microservices and Containers

Parminder Kocher

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco
Amsterdam • Cape Town • Dubai • London • Madrid • Milan
Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Library of Congress Control Number: 2017963682

ISBN-13: 978-0-13-459838-3
ISBN-10: 0-13-459838-5

1 18

Contents

Preface

As always, the technology sector is in the midst of momentous transitions—the Internet of things, software-enabled networking, and software as a service (SaaS), to name but a few. Because of these innovations, there is a large demand for platforms and architectures that can improve the process of application development and deployment. Companies of many sizes now require frameworks and architectures that can simplify their applications' update processes, allowing their latest versions to go to market more frequently without adding undue overhead to the development and deployment teams.

This transition, like many of its cousins, is still young, yet many technologies and frameworks in the space have already come and gone. The winners remain standing, however, continuing to improve the world's software by allowing its developers—us—to create new applications and update existing ones with more agility than ever before. Two such winners? Microservices and containers, red-hot topics that, in my opinion, also possess staying power. Compared to the monolithic approach, the most common way of developing and deploying applications, microservices simplify those processes, especially with large projects that require multiple teams and increasingly long code. In such cases, even a small change in the code can cause serious delays. Microservices can handle today's large codes by incorporating agility and scalability into application development and deployment, all within a proven paradigm.

That's where this book comes in. When I first started learning about microservices, there were several valuable online resources (in particular, I recommend the websites microservices.io, by Chris Richardson, and martinfowler.com, by James Lewis and Martin Fowler), but I could not find many books that systematically built a case for why a CTO or director of an engineering team should (or should not) make the transition to microservices. There was a clear gap in the market; the more I mastered the subject matter, the more I thought, "Why can't I be the one to fill that gap?" Soon I was brainstorming ideas for a book of my own.

Is This Book for You?

I wrote this book with two audiences in mind. The first group includes students, designers, and architects with experience in software and systems engineering. Although you might be familiar with microservices and/or containers, this is probably your first book dedicated entirely to them. It should provide you not only with a

comprehensive overview on the subjects but also with enough information and analysis to help you decide when—and when not—to utilize these technologies. Those of you who already have hands-on experience with microservices and/or containers may want to skim through Parts I and II and dive straight into Part III, which presents a full-fledged service desk example, written by following the standard service-oriented architectures (SOA) methodologies. This case study discusses how one such application's architecture can be converted to a microservices-based architecture as well as how Docker containers fit into the picture. I think this deep dive under the hood will be a real treat and ultimately pique your interest enough to delve into the world of microservices and containers yourself.

My other target readers are non-programmers coming at the topic from a business perspective—executives or project managers interested in learning the basics. Perhaps you read an intriguing blog post about microservices. Could that be the solution your team has been searching for but you couldn't seem to find a good follow-up book? Maybe you've overheard the engineers discussing Docker containers and want to learn enough to fit in and talk the talk. Whatever your reasons, this book—essentially a primer chock full of easy-to-understand examples and minimal jargon—should be ideal for any manager considering new ways to update or develop new applications more effectively.

This book is for anyone trying to accomplish any or all of the following:

- Make his or her organization more effective in building industrial-strength software.
- Transition into microservices and Docker containers while understanding how they differ from SOA.
- Learn microservices and Docker as part of his or her school curriculum to gain new, highly marketable skills.

In short, this book is for anyone who wants to learn more about microservices and Docker containers. I hope you are one of them! Let's get started.

Acknowledgments

As someone who has spent his entire career in tech, I never thought I would write a book. I was an engineer, not an author. And so, before embarking on this challenge, I had little idea what went into authoring a book—and how tough it would be. Let's just say, I knew it would be a lot of work, but not *this* much work. Writing this book would have been difficult enough if I had been able to devote my working days to it. Writing it while continuing to work full time seemed downright impossible at times! And it would have been, too, were it not for the many talented and generous people who guided and supported me every step of the way.

First, to the entire team at Pearson, thank you for accepting my proposal and guiding me through the entire editorial process. In particular, I want to thank my main contact there, Christopher Guzikowski for his guidance at every step, for his trust that I could do this, and for his patience while I worked on this book. Also big thanks to Michael Thurston for his indispensable editing and quick turnaround time.

This book would not have been possible without similar aid and support from my colleagues at Cisco, starting with Lenin Lakshminarayanan and Anuj Singh, who spent countless evenings and weekends with me helping with all the code-related aspects of the case study, a critical section of this book. Many thanks to Gerald Cantor, who read multiple drafts and provided honest, invaluable feedback; Ravi Papisetti, Sameer Nair, Gurvinder Singh, and Nawaz Akther for providing other useful insights and suggestions; and Michael Wolman for reviewing every word of this book—several times over.

This book also would have been impossible without the motivation and guidance I received. Whenever I had doubts, I would seek guidance from my mentors, who played a huge role in getting me to this point in my career. In particular, I would like to thank Greg Carter, my mentor for the past 12 years, for his unconditional support and guidance; Sunil Kripalani, for always trusting me and pushing me to be innovative and strive to make an impact; and Antonio Nucci, a true visionary—just talking with him motivates me to accomplish more.

Last but certainly not least, I want to thank my family for putting up with me during this rewarding but frequently stressful experience! To my children, Prabhleen,

Jashminder, and Jasleen, for spending countless weekends without me and understanding that Papa was working on his passion. And finally, especially, to my beautiful wife, Raman, for her inspiration, encouragement, and trust in me. If not for her support, this book would have remained merely a dream, not a reality.

Thank you all so much!

About the Author

Parminder Kocher was born and raised in India and is a lifelong technology learner with two decades of hands-on experience in building enterprise-grade software systems. He has been with Cisco Systems since 2005 and managed the company's Remote Management Service (RMS) platform, and has since worked as an innovation evangelist leading multiple software groups. Currently, he is engineering director for Cisco Networking Academy platform, where he leads the engineering teams responsible for developing the Academy's next-gen platform accesses in 178 countries. In addition to bachelor's and master's degrees in computer science, Kocher has an executive MBA from Baylor's Hankamer School of Business and an executive certificate in strategy and innovation from MIT's Sloan School of Management. He lives in Austin, Texas, with his wife and three children, and serves as committee chair of the Boy Scout troop he founded in 2013.

Chapter 4

Migrating and Implementing Microservices

By this point you know what microservices are and how they work. If you're still reading, I have accomplished my first goal: piquing your interest enough that you are considering implementing microservices yourself! Now it's time to get down to brass tacks: namely, the very critical topic of how to approach the transition to microservices.

The Need for Transition

You'll recall that a monolithic application is very large (in terms of lines of code [LoC]) and complex (in terms of functions interdependencies, data, etc.), serving hundreds of thousands of users across geographical regions and requiring several developers and IT engineers. A monolithic app may look something like Figure 4.1.

Sometimes, even with all these characteristics, the application might run fine at first. You may not encounter challenges in terms of application scalability or performance. But with time and usage, issues will arise, and they may be different for different applications. For example, for a cloud or web application, you may hit scalability issues due to more users consuming your services, or it may become costly and hard to release regular new updates due to longer build times and regression testing. As shown in Figure 4.2, monolithic application users or the developers may experience one or more issues listed on the right.

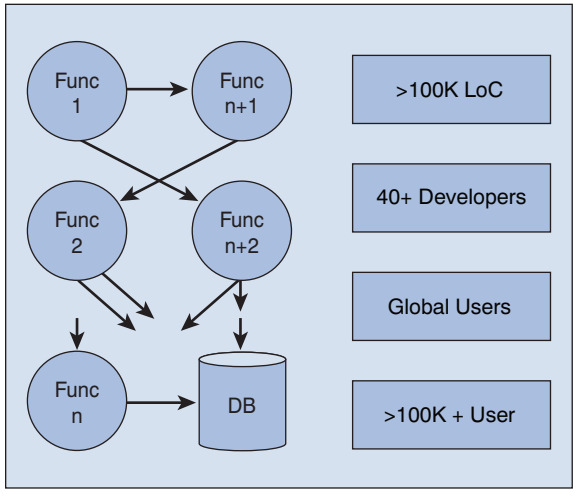


Figure 4.1 *Basic structure of a monolithic app*

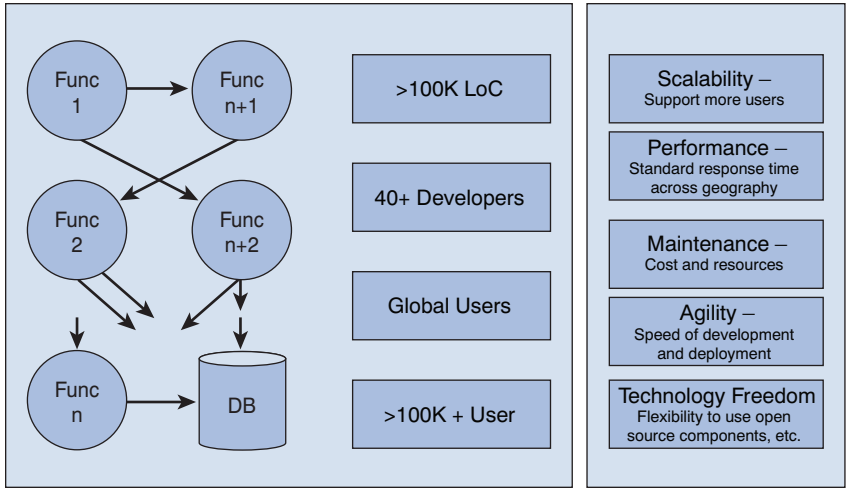


Figure 4.2 *Potential issues with a monolithic app*

That's when a migration to microservices may start sounding like more than a trendy idea; it will sound like a lifesaver. We already learned a bit about microservices in previous chapters, so we know our transition will look something like the application shown in Figure 4.3.

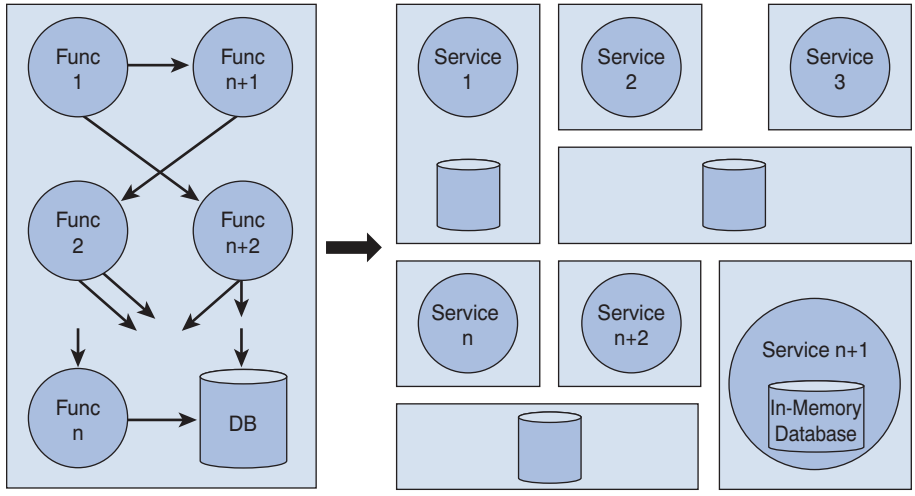


Figure 4.3 *Transition from monolithic to microservices*

So, how do we go about making such a change? There are two possible scenarios: creating a brand-new application or converting or migrating a monolithic application that already exists. The latter scenario is far more likely, but it is worth knowing the ins and outs of both scenarios regardless of the current situation.

Creating a New Application with Microservices

Before we begin, let me say that I have not seen many real-world scenarios of building a microservices-based application from scratch. Typically, an application is already in place, and most applications I have worked on are more of a transition to a microservices architecture from a monolithic architecture. In these cases, the intention of architects and developers has always been to reuse some of the existing implementation. As skills become readily available in the market and some successful implementations are published, we will see more examples of building microservices-based applications from scratch, so it is certainly worthwhile to discuss this scenario.

Let's say you have all the requirements figured out and ready to go into the architecture design of the application you are going to build. There are many common best practices you need to think about as you get started, which are covered in the following sections.

Organization Readiness

As we discussed in Chapter 2, "Switching to Microservices," the first question you have to ask yourself is whether your organization is ready to transition to microservices. That means the various departments of your organization now need to think differently about building and releasing software in the following ways:

- **Team structure.** The monolithic application team (if one exists) needs to be broken down into several small high-performance teams aware of or trained in microservices best practices. As you saw in Figure 4.3, the new system will consist of a set of independent services, each responsible for offering a specific service. This is one key advantage of the microservices paradigm—it reduces the communication overheads, including those multiple nonstop meetings. Teams should be organized by business problems or areas they are trying to address. The communication then becomes about the timing and set of standards/protocols to follow so that these microservices can work with each other as one platform.
- **Agility.** Each team must be prepared to function independently of others. They should be the size of a standard scrum team; otherwise, communication will become an issue again. Execution is the key, and each team should be able to address the changing business needs.
- **Tools and training.** One of the key needs is the organization's readiness to invest in new tools and people training. The existing tools and processes, in most cases, would need to be retired and new ones picked up. This will require a large capital investment as well as investment in hiring people with new skills and retraining existing staff members. In the long term, if the decision is right to get on microservices, organizations will see savings and recoup the investment.

Services-Based Approach

Unlike with monolithic applications, with microservices you need to take a self-sustained services-based approach. Think of your application as a bunch of loosely coupled services that communicate with each other to provide complete application functionality. Each service must be thought of as an independent, self-contained service with its own lifecycle that can be developed and maintained by independent teams. These teams may select from a variety of technologies, including languages or databases that best suit their services' needs. For example, for an e-commerce site, the team would write a completely independent service, such as a shopping

cart microservice, with an in-memory database, and another one, such as an ordering microservice, with a relational database. A real-world application may employ microservices for basic functions such as authentication, account, user registration, and notification with the business logic encapsulated in an API gateway that calls these microservices based on the client and external requests.

Just a reminder: a microservice may be a small service implemented by a single developer or a complex service requiring a few developers. With microservices, the size does not matter; it all depends on one function that a service has to provide.

Other aspects that must be considered at this point are scaling, performance, and security. Scaling needs can be different and provided on an as-needed basis at each microservice level. Security should be thought of at all levels, including data at rest, interprocess communication, data at motion, and so on.

Interprocess (Service-to-Service) Communication

We discussed the topic of interprocess communication in depth in Chapter 3, "Interprocess Communication." Key aspects that must be thought of are security and communication protocol. Asynchronous communication is the way to go, as it keeps all requests on track and does not hold resources for extended periods of time.

Using a message bus such as RabbitMQ may prove to be beneficial for this kind of communication. It is simple and can scale to hundreds of thousands of messages per second. To prevent the messaging system from becoming a single point of failure if it goes down, the messaging bus must be properly designed for high availability. Other options include ActiveMQ, which is another lightweight messaging platform.

Security is key at this stage. In addition to selecting the right communication protocol, industry standard tools such as AppDynamics may be used to monitor and benchmark the interprocess communication. Any anomalies must be reported automatically to the security team.

When there are thousands of microservices, it does become complex to handle everything. We already discussed how to address such issues through discovery services and API gateways in Chapter 3.

Technology Selection

The biggest advantage of transitioning to microservices is that it enables choices. Each team can independently select the language, technology, database, and so on, that is the best fit for the given microservice. Usually in a monolithic approach, the team does not have this flexibility, so make sure you do not overlook and miss the opportunity.

Even if a team is handling multiple microservices, each microservice must be looked at as a self-contained service, and it needs be analyzed. Scalability, deployment, build time, integrations and plugins operability, and so on, must be kept in mind when choosing the technology for each microservice. For microservices with lighter data but faster access, an in-memory database may be most suitable, whereas others may share the same relational or NoSQL databases.

Implementation

Implementation is the critical phase; this is where all the training and best practices knowledge comes in handy. Some of the critical aspects to keep in mind include the following:

- **Independency.** Each microservice should be highly autonomous with its own lifecycle and treated as such. It needs to be developed and maintained without any dependencies on other microservices.
- **Source control.** A proper version control system must be put at place, and each microservice must follow the standards. Standardizing on a repository is also helpful, as it ensures all the teams use the same source control. It helps in various aspects, such as code review, providing easy access to all the code in one place. In the long term, it makes sense to have all the services on the same source control.
- **Environments.** All different environments, such as dev, test, stage, and production, must be properly secured and automated. The automation here includes the build process—that way the code can be integrated as required, mostly on a daily basis. There are several tools available, and Jenkins is widely used. Jenkins is an open source tool that helps automate the software build and release process including continuous integration and continuous delivery.
- **Failsafe.** Things can go wrong, and software failure is inevitable. Handling failures of downstream services must be addressed within the microservice development. Failure of other services must be graceful to the extent that the failure should be invisible to the end user. This includes managing service response times (timeouts), handling API changes for downstream services, and limiting the number of auto-retry.
- **Reuse.** With microservices, don't be shy about reusing the code by using copy and paste, but do it within limits. This may cause some code duplication, but it's better than using shared code that may end up coupling services. In microservices, we want decoupling, not tight coupling. For example, you will write

code to consume the output response from a service. You can copy this code every time you call the same service from any client. Another way to reuse code is by creating common libraries. Multiple clients can use the same library, but then each client should be responsible for maintaining its libraries. It can sometimes become challenging when you create too many libraries and each client is maintaining a different version. In that case, you may have to include multiple versions of same library, and the build process may become difficult due to backward compatibility and similar concerns. Depending on your needs, you can go either way as long as you can control the number of libraries and versions by clients and put a tight process around them. This will certainly save you from lot of code duplication.

- **Tagging.** Given the sheer number of microservices, debugging a problem may become difficult, so you need to do some kind of instrumentation at this stage. One of the best practices is to tag each request with a unique request ID and log each one of them. This unique ID will identify the originating request and should be passed by each service to any downstream requests. When you see issues, you can clearly track back through logs and identify the problematic service. This solution will be most effective if you establish a centralized logging system. All the services should log in all the messages to this shared system in a standardized format so that teams can replay the events as required all from one place, from infrastructure to application. A shared library for centralized logging is worth looking into, as we previously discussed. There are several log management and aggregation tools out there in the market, such as ELK (Elasticsearch, Logstash, Kibana) and Splunk, that are ideal.

Deployment

Automation is the key during deployment. Without it, success with a microservices paradigm would be almost impossible. As we discussed, there may be hundreds to thousands of microservices, and for the agile delivery, automation is a *must*.

Think of deploying thousands of microservices and maintaining them. What happens when one of the microservices goes down? How do you know which machine has enough resources to run your microservices? It becomes very complicated to manage this without automation in place. Various tools, such as Kubernetes and Docker Swarm, can be used to automate the deployment process. Given the importance of this topic, a whole chapter, Chapter 9, "Container Orchestration," is dedicated to deployment.

Operations

The operations part of the process needs to be automated as well. Again, we are talking about hundreds to thousands of microservices—organizational capabilities need to mature enough to handle this level of complexity. You'll need a support system, including the following:

- **Monitoring.** From infrastructure to application APIs to last-mile performance, everything should be monitored, and automatic alerts with proper thresholds should be put in place. Consider building live dashboards with data and alerts that pop up during issues.
- **On-demand scalability.** With microservices, scalability is the simplest task. Provision another instance of your microservice you want to scale and just put it behind the existing load balancer and you are all set. But in a scaled environment, this also needs to be automated. As we will discuss later, it is a matter of setting up an integer value to tell the number of instances you want to run for a particular microservice.
- **API exposure.** In most cases, you will want to expose the APIs externally for external users to consume. This is best done by using an edge server, which can handle all the external requests. It can utilize the API gateway and discovery service to do its job, and you can use one edge server per device type (e.g., mobile, browser) or use case. An open source application created by Netflix, called Zuul, can be utilized for this function and beyond.
- **Circuit breaker.** Sending a request to a failed service is pointless. Hence, a circuit breaker can be built that tracks the success and failure of every request made to every service. In the case of multiple failures, all the requests to that particular service should be blocked (break the circuit) for a set time. After the set time expires, another attempt should be made, and so on. Once the response is successful, reconnect the circuit. This should be done at the service instance level. Netflix's Hystrix provides an open source circuit-breaker implementation.

Migrating a Monolithic Application to Microservices

While most of the best practices for building a new microservices-based application apply to migrating from an existing monolithic application as well, there are some additional guidelines that, if followed, will make the migration simpler and more efficient.

Although it may sound correct to convert the whole monolithic application to a completely microservices-based application, it may not be efficient or may be very costly in some cases to convert every function or capability into microservices. You might end up writing the application from scratch, after all. The right way to migrate may require a stepwise approach, as shown in Figure 4.4.

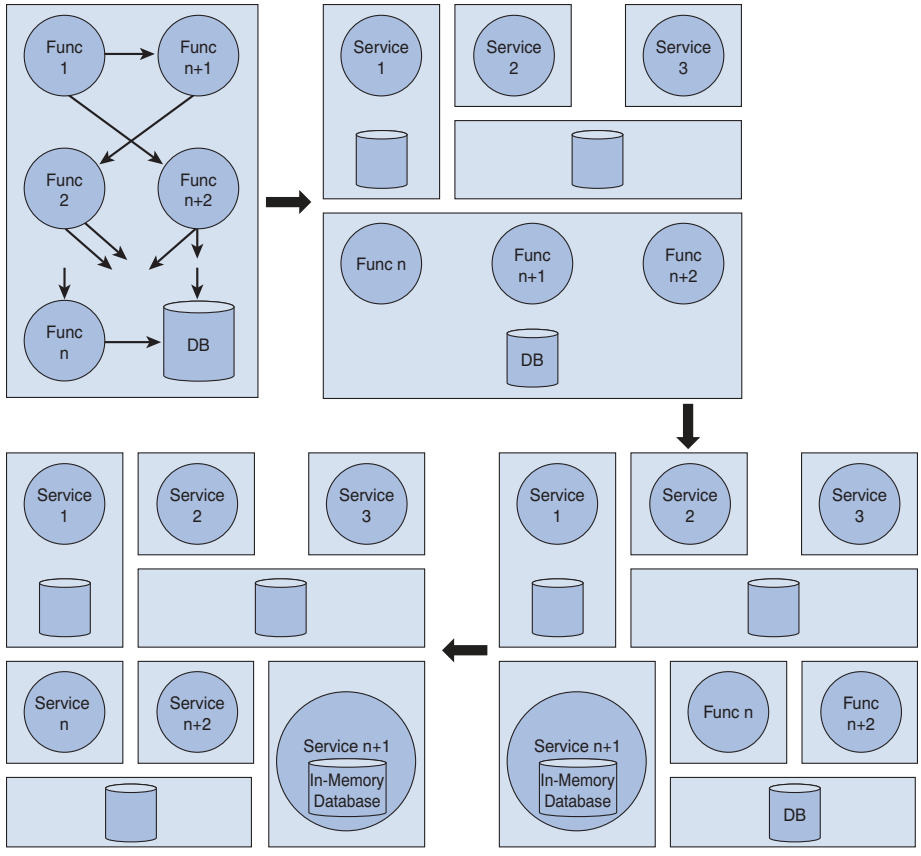


Figure 4.4 *Basic migration steps, monolithic to microservices*

The next question is, Where do we start with the current monolithic application? If the application is really old and it would be time consuming and difficult to take pieces out (i.e., if there is very high level of cohesiveness), then it is probably better to start from scratch. In other cases where parts of the code can be disabled quickly and the technology architecture is not completely outdated, it is better to start with rebuilding the components as microservices and replace the old code.

Microservices Criteria

The question then becomes what components should be migrated first or even migrated at all. That brings us to what I call the "microservices criteria," which outline one of the possible ways to select and prioritize the functions that should be migrated to microservices. They are a set of rules you establish that either qualifies or disqualifies the conversion of your existing monolithic application's components to microservices given the organization's needs at that time.

That "time" is very important here because with time the needs of the organization may change, and you may have to come back and convert more components to microservices later. In other words, with changing needs, additional components of your monolithic application may qualify for the conversion.

Here are best practices that can be considered as microservices criteria during the conversion process:

- **Scale.** You need to determine which functions are highly used. Convert the highly used services or application functionality as microservices first. Recall, a microservice performs only one well-defined service. Keep the principle in mind and divide the application accordingly.
- **Performance.** There likely are components that are not performing well, and other alternatives are readily available. It may be there is open source plugin available, or you may want to build a service from scratch. One of the key things to keep in mind is the boundary of a microservice. As long as you design your microservice in such a way that it does one and only one thing well, it is good. Determining the boundary is often going to be hard, and you will find it easier to do this with practice. Another way to look at the microservice boundary is that you should be able to rewrite the whole microservice in a few weeks' time (if/when required) as opposed to taking few months to rewrite the service.
- **Better technology alternatives or polyglot programming.** Domain-specific languages can be employed to help with problem domains. This is particularly applicable to components for which you received many enhancement requests in the past and you expect that to continue. If you think not only that such a component's implementation can be simplified using a new language or capability in the market but also that future maintenance and updates would become easier, then now is the right time to address such changes. In other cases, you may find another language provides easier abstractions for concurrency than the current one used. You can leverage the new language for a given microservice

while the rest of the application can still be using a different language. Likewise, you may want some microservices to be extremely fast and may decide to write them in C to get the maximum gains rather than writing in another high-level language. The bottom line is to take advantage of this flexibility.

- **Storage alternatives or polyglot persistence.** With the rise of big data, some components of the application may provide value by using NoSQL databases rather than relational databases. If any such component in the application may benefit from this alternative, then it may be right time to make the switch to NoSQL.

  These are the key aspects you should consider for each service or feature within your monolithic application, and you need to prioritize the conversion of such items first. Once you have derived the value from high-priority items, you can then apply other rules.

- **Modification requests.** One important thing to track in any software lifecycle is the new enhancements requests or changes. Features that have a higher number of change requests may be suitable for microservices because of the build and deployment time. Separating such services reduces the build and deployment time, as you will not have to build the entire application, just the changed microservice, which may also increase availability time for the rest of the application.

- **Deployment.** There are always some parts of the application that add deployment complexity. In a monolithic application, even if a particular feature is untouched, you still must go through the complete build and deployment process. If such cases exist, it is beneficial to cut out such pieces and replace them with microservices so your overall deployment time is reduced for the rest of the monolithic application. We talk more about this after we learn about containers.

- **Helper services.** In most applications, the core or main service depends on some of the helper services. The unavailability of such helper functions may impact the availability of the core service. For example, in the helpdesk application, ticketing depends on the product catalog service. If the product catalog service is not available, the user will be unable to submit a ticket. If such cases exist, helper services should be converted to microservices and appropriately made highly available so they can better serve core services. These are also called *circuit-breaker services*.

Depending on the application, this criteria may require most of the services to be converted to microservices, and that is okay. The intention here is to simplify the conversion process so that you can prioritize and define the roadmap for your migration to a microservices-based architecture.

Rearchitecting the Services

Once you have identified the functions to be migrated as microservices, it's time to start rearchitecting the selected services following the best practices from the earlier scenario. Here are the aspects to keep in mind:

- **Microservices definition.** For each function, define the appropriate microservices, which should include communication mechanism (API), technology definition, and so on. Consider the data your existing function uses, or create and plan accordingly the data strategy for microservices. If the function was on heavy databases such as Oracle, would it make sense to move to MySQL? Determine how you are going to manage the data relationship. Finally, run each microservices as a separate application.
- **Refactor code.** You may reuse some of the code if you are not changing the programming language. Think about the storage/database layer—shared vs. dedicated, in-memory vs. external. The goal here is not to add new functionality unless required but to repackage the existing code and expose the required APIs.
- **Versioning.** Before you begin coding, decide on the source control and versioning mechanism, and make sure these standards are followed. Each microservice is to be a separate project and deployed as a separate application.
- **Data migration.** If you decide to create a new database, you will have to migrate the legacy data also. This is usually handled by writing simple SQL scripts depending on your source and destination.
- **Monolithic code.** Initially, leave the existing code in place in the monolithic application in case you have to roll back. You can either update the rest of the code to use the new microservices or, better, split your application traffic, if possible, to utilize both the monolithic and microservices version. This provides you the opportunity to test and keep an eye on performance. Once confident, you can move all the traffic to microservices and disable/get rid of old code.

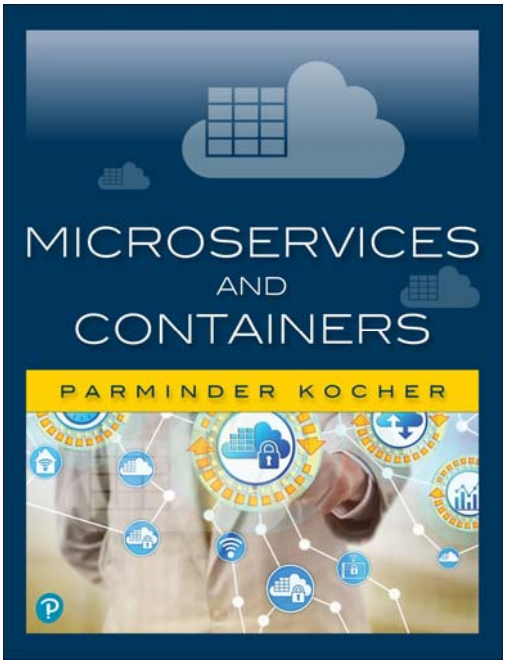
- **Independent build, deploy, and manage.** Build and deploy each microservice independently. As you roll out new versions of microservices, you can again split the traffic between the old and the new version for some time. This means that you may have two or more versions of the same microservice running in the production environment. Some of the user traffic can be routed to the new microservice version to make sure the service works and performs right. If the new version is not performing optimally or as expected, it would be easy to roll back all the traffic to the previous version and send the new version back to development. The key here is to set up the repeatable automated deployment process and move toward continuous delivery.
- **Old code removal.** You can remove your temporary code and delete the data from the old storage location only after you have verified that everything is migrated correctly and operating as expected. Be sure to make backups along the way.

A Hybrid Approach

When writing a brand-new application, developers can directly follow the microservices architecture principles and blueprint to build the software application, as we have discussed. Developers sometimes follow a kind of hybrid approach of microservices and monolithic. In this case, they can develop part of their application as microservices and the rest following standard SOA/MVC practices based on certain criteria. The idea is that not all the components of the application may qualify as microservices.

As we discussed in Chapter 3, microservices offer lot of flexibility, but this flexibility comes at some cost. The hybrid approach is to balance the flexibility and cost aspects with the understanding that, over time, components can be pulled out of the monolithic part and converted to microservices on an as-needed basis. The key is to keep both approaches in mind, along with microservices criteria, during this transition.

Transition to Microservices and DevOps to Transform your Software Effectiveness



Available March 30, 2018

ORDER & SAVE

SAVE 35% WHEN YOU ORDER

from **informit.com/kocher** and enter the code **INFOQ** during checkout

FREE US SHIPPING on print books

Major eBook Formats

Only InformIT offers PDF, EPUB, & MOBI together for one price

OTHER AVAILABILITY

Through O'Reilly Safari subscription service

Booksellers and online retailers including Amazon/Kindle store and Barnes and Noble/bn.com

Together, microservices and Docker containers can bring unprecedented agility and scalability to application development and deployment—especially in large, complex projects where speed is crucial but small errors can be disastrous. In **Microservices and Containers**, Parminder Singh Kocher demonstrates why and how these technologies can help you build, deploy, manage, and scale industrial-strength applications.

Learn how to leverage microservices and Docker to drive exponential improvements in DevOps effectiveness, on-demand scalability, application performance, time-to-market, code reuse, and application reliability. Kocher also offers detailed guidance and a complete roadmap for transitioning from monolithic architectures, and an in-depth case study walking through the migration of an enterprise-class SOA system.

- Understand how microservices enable you to organize applications into standalone components that are easier to manage, update, and scale
- Decide whether microservices and containers are worth your investment, and manage the organizational learning curve associated with them
- Apply best practices for interprocess communication among microservices
- Migrate monolithic systems in an orderly fashion
- Understand Docker containers, installation, and interfaces
- Network, orchestrate, and manage Docker containers effectively
- Use Docker to maximize scalability in microservices-based applications

Parminder Kocher is a lifelong technology learner with two decades of hands-on experience in building enterprise-grade software systems. He started at Cisco Systems in 2005 managing the Remote Management Service (RMS) platform, and has since worked as an innovation evangelist leading multiple software groups. Currently, he is engineering director for Cisco Networking Academy platform, where he leads the engineering teams responsible for developing the Academy's next-gen platform.

Pearson Addison-Wesley

informIT the trusted technology learning source