THE DEVOPS 2.0 TOOLKIT

AUTOMATING THE CONTINUOUS DEPLOYMENT PIPELINE WITH CONTAINERIZED MICROSERVICES

VIKTOR FARCIC

The DevOps 2.0 Toolkit

Automating the Continuous Deployment Pipeline with Containerized Microservices

Viktor Farcic

This book is for sale at http://leanpub.com/the-devops-2-toolkit

This version was published on 2016-09-19

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Viktor Farcic by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought The DevOps 2.0 Toolkit by @vfarcic

The suggested hashtag for this book is #devops2book.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#devops2book

Contents

Preface

I started my career as a developer. During those early days, all I knew (and thought I should know) was to write code. I believed that a great software designer is a person that is proficient in writing code and that the path to the mastery of the craft was to know everything about a single programming language of choice. Later on, that changed and I started taking an interest in different programming languages. I switched from Pascal to Basic and then ASP. When Java and, later on, .Net came into existence, I learned benefits of object oriented programming. Python, Perl, Bash, HTML, JavaScript, Scala… Each programming language brought something new and taught me how to think differently and how to pick the right tool for the task at hand. With each new language I learned, I felt like I was closer to being an expert. All I wanted was to become a senior programmer. That desire changed with time. I learned that if I was to do my job well, I had to become a *software craftsman*. I had to learn much more than to type code. Testing became my obsession for some time, and now I consider it an integral part of development. Except in very special cases, each line of code I write is done with *test-driven development (TDD)*. It became an indispensable part of my tool-belt. I also learned that I had to be close to the customer and work with him side by side while defining what should be done. All that and many other things led me to *software architecture*. Understanding the "big picture" and trying to fit different pieces into one big system was the challenge that I learned to like.

Throughout all the years I've been working in the software industry, there was no single tool, framework or practice that I admired more than *continuous integration (CI)* and, later on, *continuous delivery (CD)*. The real meaning of that statement hides behind the scope of what CI/CD envelops. In the beginning, I thought that CI/CD means that I knew *Jenkins* and was able to write scripts. As the time passed I got more and more involved and learned that CI/CD relates to almost every aspect of software development. That knowledge came at a cost. I failed (more than once) to create a successful CI pipeline with applications I worked with at the time. Even though others considered the result a success, now I know that it was a failure because the approach I took was wrong. CI/CD cannot be done without making architectural decisions. Similar can be said for tests, configurations, environments, fail-over, and so on. To create a successful implementation of CI/CD, we need to make a lot of changes that, on the first look, do not seem to be directly related. We need to apply some patterns and practices from the very beginning. We have to think about architecture, testing, coupling, packaging, fault tolerance, and many other things. CI/CD requires us to influence almost every aspect of software development. That diversity is what made me fall in love with it. By practicing CI/CD we are influencing and improving almost every aspect of the software development life cycle.

To be truly proficient with CI/CD, we need to be much more than experts in operations. The DevOps movement was a significant improvement that combined traditional operations with advantages that development could bring. I think that is not enough. We need to know and influence architecture,

testing, development, operations and even customer negotiations if we want to gain all the benefits that CI/CD can bring. Even the name DevOps as the driving force behind the CI/CD is not suitable since it's not only about development and operations but everything related to software development. It should also include architects, testers, and even managers. DevOps was a vast improvement when compared to the traditional operations by combining them with development. The movement understood that manually running operations is not an option given current business demands and that there is no automation without development. I think that the time came to redefine DevOps by extending its scope. Since the name *DevOpsArchTestManageAndEverythingElse* is too cumbersome to remember and close to impossible to pronounce, I opt for **DevOps 2.0**. It's the next generation that should drop the heavy do-it-all products for smaller tools designed to do very specific tasks. It's the switch that should go back to the beginning and not only make sure that operations are automated but that the whole system is designed in a way that it can be automated, fast, scalable, fault-tolerant, with zero-downtime, easy to monitor, and so on. We cannot accomplish this by simply automating manual procedures and employing a single do-it-all tool. We need to go much deeper than that and start refactoring the whole system both on the technological as well as the procedural level.

Overview

This book is about different techniques that help us architect software in a better and more efficient way with *microservices* packed as *immutable containers*, *tested* and *deployed continuously* to servers that are *automatically provisioned* with *configuration management* tools. It's about fast, reliable and continuous deployments with *zero-downtime* and ability to *roll-back*. It's about *scaling* to any number of servers, design of *self-healing systems* capable of recuperation from both hardware and software failures and about *centralized logging and monitoring* of the cluster.

In other words, this book envelops the whole *microservices development and deployment lifecycle* using some of the latest and greatest practices and tools. We'll use *Docker, Kubernetes, Ansible, Ubuntu, Docker Swarm and Docker Compose, Consul, etcd, Registrator, confd, Jenkins*, and so on. We'll go through many practices and, even more, tools.

Finally, while there will be a lot of theory, this is a hands-on book. You won't be able to complete it by reading it in a metro on a way to work. You'll have to read this book while in front of a computer getting your hands dirty. Eventually, you might get stuck and in need of help. Or you might want to write a review or comment on the book's content. Please post your thoughts on the The DevOps 2.0 Toolkit channel in Disqus[1]. If you prefer one-on-one discussion, feel free to send me an email to *viktor@farcic.com*, or to contact me on HangOuts, and I'll give my best to help you out.

[1] https://disqus.com/home/channel/thedevops20toolkit/

Audience

This book is for professionals interested in the full microservices lifecycle combined with continuous deployment and containers. Due to the very broad scope, target audience could be *architects* who want to know how to design their systems around microservices. It could be *DevOps* wanting to know how to apply modern configuration management practices and continuously deploy applications packed in containers. It is for *developers* who would like to take the process back into their hands as well as for *managers* who would like to gain a better understanding of the process used to deliver software from the beginning to the end. We'll speak about scaling and monitoring systems. We'll even work on the design (and implementation) of self-healing systems capable of recuperation from failures (be it of hardware or software nature). We'll deploy our applications continuously directly to production without any downtime and with the ability to rollback at any time.

This book is for *everyone wanting to know more about the software development lifecycle* starting from requirements and design, through development and testing all the way until deployment and post-deployment phases. We'll create the processes taking into account best practices developed by and for some of the biggest companies.

About the Author

Viktor Farcic is a Senior Consultant at CloudBees[2].

He coded using a plethora of languages starting with Pascal (yes, he is old), Basic (before it got Visual prefix), ASP (before it got .Net suffix), C, C++, Perl, Python, ASP.Net, Visual Basic, C#, JavaScript, etc. He never worked with Fortran. His current favorites are **Scala** and **JavaScript** even though most of his office hours he spends with **Java**.

His big passions are Microservices, Continuous Deployment and Test-Driven Development (TDD).

He often speaks at community gatherings and conferences.

He wrote Test-Driven Java Development[3].

[2] https://www.cloudbees.com/

[3] https://www.packtpub.com/application-development/test-driven-java-development

The DevOps Ideal

Working on small greenfield projects is great. The last one I was involved with was during the summer of 2015 and, even though it had its share of problems, it was a real pleasure. Working with a small and relatively new set of products allowed us to choose technologies, practices, and frameworks we liked. Shall we use microservices? Yes, why not. Shall we try Polymer and GoLang? Sure! Not having baggage that holds you down is a wonderful feeling. A wrong decision would put us back for a week, but it would not put in danger years of work someone else did before us. Simply put, there was no legacy system to think about and be afraid of.

Most of my career was not like that. I had the opportunity, or a curse, to work on big inherited systems. I worked for companies that existed long before I joined them and, for better or worse, already had their systems in place. I had to balance the need for innovation and improvement with obvious requirement that existing business must continue operating uninterrupted. During all those years I was continuously trying to discover new ways to improve those systems. It pains me to admit, but many of those attempts were failures.

We'll explore those failures in order to understand better the motivations that lead to the advancements we'll discuss throughout this books.

Continuous Integration, Delivery, and Deployment

Discovering CI and, later on, CD, was one of the crucial points in my career. It all made perfect sense. The integration phase back in those days could last anything from days to weeks or even months. It was the period we all dreaded. After months of work performed by different teams working on different services or applications, the first day of the integration phase was the definition of hell on earth. If I didn't know better, I'd say that Dante was a developer and wrote Infierno during the integration phase.

On the dreaded first day of the integration phase, we would all come to the office with grim faces. Only whispers could be heard while the integration engineer would announce that the whole system was set up, and the "game" could begin. He would turn it on and, sometimes, the result would be a blank screen. Months of work in isolation would prove, one more time, to be a disaster. Services and applications could not be integrated, and the long process of fixing problems would begin. In some cases, we would need to redo weeks of work. Requirements defined in advance were, as always, subject to different interpretations and those differences are nowhere more noticeable than in the integration phase.

Then *eXtreme Programming (XP)* practices came into existence and, with them, *continuous integration (CI)*. The idea that integration should be done continuously today sounds like something

obvious. Duh! Of course, you should not wait until the last moment to integrate! Back then, in the waterfall era, such a thing was not so obvious as today. We implemented a continuous integration pipeline and started checking out every commit, running static analysis, unit and functional tests, packaging, deploying and running integration tests. If any of those phases failed, we would abandon what we were doing and made fixing the problem detected by the pipeline our priority. The pipeline itself was fast. Minutes after someone would make a commit to the repository we would get a notification if something failed. Later on, *continuous delivery (CD)* started to take ground, and we would have confidence that every commit that passed the whole pipeline could be deployed to production. We could do even better and not only attest that each build is production ready, but apply *continuous deployment* and deploy every build without waiting for (manual) confirmation from anyone. And the best part of all that was that everything was fully automated.

It was a dream come true. Literally! It was a dream. It wasn't something we managed to turn into reality. Why was that? We made mistakes. We thought that CI/CD is a task for the operations department (today we'd call them *DevOPS*). We thought that we could create a process that wraps around applications and services. We thought that CI tools and frameworks are ready. We thought that architecture, testing, business negotiations and other tasks were the job for someone else. We were wrong. I was wrong.

Today I know that successful CI/CD means that no stone can be left unturned. We need to influence everything; from architecture through testing, development and operations all the way until management and business expectations. But let us go back again. What went wrong in those failures of mine?

Architecture

Trying to fit a monolithic application developed by many people throughout the years, without tests, with tight coupling and outdated technology is like an attempt to make an eighty-year-old lady look young again. We can improve her looks, but the best we can do is make her look a bit less old, not young. Some systems are, simply put, too old to be worth the "modernization" effort. I tried it, many times, and the result was never as expected. Sometimes, the effort in making it "young again" is not cost effective. On the other hand, I could not go to the client of, let's say, a bank, and say "we're going to rewrite your whole system." Risks are too big to rewrite everything and, be it as it might, due to its tight coupling, age, and outdated technology, changing parts of it is not worth the effort. The commonly taken option was to start building the new system and, in parallel, maintain the old one until everything was done. That was always a disaster. It can take years to finish such a project, and we all know what happens with things planned for such a long term. That's not even the waterfall approach. That's like standing at the bottom of Niagara Falls wondering why you get wet. Even doing trivial things like updating the JDK was quite a feat. And those were the cases when I would consider myself lucky. What would you do with, for example, codebase done in Fortran or Cobol?

Then I heard about microservices. It was like music to my ears. The idea that we can build many small independent services that can be maintained by small teams, have codebase that can be understood in no time, being able to change framework, programming language or a database without affecting

the rest of the system and being able to deploy it independently from the rest of the system was too good to be true. We could, finally, start taking parts of the monolithic application out without putting the whole system at (significant) risk. It sounded too good to be true. And it was. Benefits came with downsides. Deploying and maintaining a vast number of services turned out to be a heavy burden. We had to compromise and start standardizing services (killing innovation), we created shared libraries (coupling again), we were deploying them in groups (slowing everything), and so on. In other words, we had to remove the benefits microservices were supposed to bring. And let's not even speak of configurations and the mess they created inside servers. Those were the times I try to forget. We had enough problems like that with monoliths. Microservices only multiplied them. It was a failure. However, I was not yet ready to give up. Call me a masochist.

I had to face problems one at a time, and one of the crucial ones was deployments.

Deployments

You know the process. Assemble some artifacts (JAR, WAR, DLL, or whatever is the result of your programming language), deploy it to the server that is already polluted with… I cannot even finish the sentence because, in many cases, we did not even know what was on the servers. With enough time, any server maintained manually becomes full of "things". Libraries, executables, configurations, gremlins and trolls. It would start to develop its own personality. Old and grumpy, fast but unreliable, demanding, and so on. The only thing all the servers had in common was that they were all different, and no one could be sure that software tested in, let's say, pre-production environment would behave the same when deployed to production. It was a lottery. You might get lucky, but most likely you won't. Hope dies last.

You might, rightfully, wonder why we didn't use virtual machines in those days. Well, there are two answers to that question, and they depend on the definition of "those days". One answer is that in "those days" we didn't have virtual machines, or they were so new that management was too scared to approve their usage. The other answer is that later on we did use VMs, and that was the real improvement. We could copy production environment and use it as, let's say testing environment. Except that there was still a lot of work to update configurations, networking, and so on. Besides, we still did not know what was accumulated on those machines throughout the years. We just knew how to duplicate them. That still did not solve the problem that configurations were different from one VM to another as well as that a copy is the same as the original only for a short period. Do the deployment, change some configuration, bada bing, bada boom, you go back to the problem of testing something that is not the same as it will be in production. Differences accumulate with time unless you have a repeatable and reliable automated process instead of manual human interventions. If such a thing would exist, we could create immutable servers. Instead of deploying applications to existing servers and go down the path of accumulating differences, we could create a new VM as part of the CI/CD pipeline. So, instead of creating JARs, WAR, DLLs, and so on, we started creating VMs. Every time there was a new release it would come as a complete server built from scratch. That way we would know that what was tested is what goes into production. Create new VM with software deployed, test it and switch your production router to point from the old to the new one. It was awesome, except that it was slow and resource demanding. Having a separate VM for each

service is overkill. Still, armed with patience, immutable servers were a good idea, but the way we used that approach and the tools required to support it were not good enough.

Orchestration

The orchestration was the key. *Puppet* and *Chef* proved to be a big help. Programming everything related to servers setup and deployment was a huge improvement. Not only that the time needed to setup servers and deploy software dropped drastically, but we could, finally, accomplish a more reliable process. Having humans (read operations department) manually running those types of tasks was a recipe for disaster. Finally a story with a happy ending? Not really. You probably started noticing a pattern. As soon as one improvement was accomplished, it turned out that it, often, comes with a high price. Given enough time, Puppet and Chef scripts and configurations turn into an enormous pile of **** (I was told not to use certain words so please fill in the blanks with your imagination). Maintaining them tends to become a nightmare in itself. Still, with orchestration tools, we could drastically reduce the time it took to create immutable VMs. Something is better than nothing.

The Light at the End of the Deployment Pipeline

I could go on and on describing problems we faced. Don't take me wrong. All those initiatives were improvements and have their place in software history. But history is the past, and we live in the present trying to look into the future. Many, if not all of the problems we had before are now solved. *Ansible* proved that orchestration does not need to be complicated to set up nor hard to maintain. With the appearance of *Docker*, containers are slowly replacing VMs as the preferable way to create immutable deployments. New operating systems are emerging and fully embracing containers as first class citizens. Tools for service discovery are showing us new horizons. *Swarm, Kubernetes and Mesos/DCOS* are opening doors into areas that were hard to imagine only a few years ago.

Microservices are slowly becoming the preferred way to build big, easy to maintain and highly scalable systems thanks to tools like *Docker, CoreOS, etcd, Consul, Fleet, Mesos, Rocket*, and others. The idea was always great, but we did not have the tools to make it work properly. Now we do! That does not mean that all our problems are gone. It means that when we solve one problem, the bar moves higher up, and new issues emerge.

I started by complaining about the past. That will not happen again. This book is for readers who do not want to live in the past but present. This book is about preparations for the future. This book is about stepping through the looking glass, about venturing into new areas and about looking at things from a new angle.

This is your last chance. After this, there is no turning back. You take the blue pill - the story ends, you wake up in your bed and believe whatever you want to believe. You take the red pill - you stay in Wonderland and I show you how deep the rabbit-hole goes.

– Morpheus (Matrix)

If you took the blue pill, I hope that you didn't buy this book and got this far by reading the free sample. There are no hard feelings. We all have different aspirations and goals. If, on the other hand, you chose the red one, you are in for a ride. It will be like a roller coaster, and we are yet to discover what awaits us at the end of the ride.

The Implementation Breakthrough: Continuous Deployment, Microservices, and Containers

On the first look *continuous deployment (CD)*, *microservices (MS)* and *containers* might seem like three unrelated subjects. After all, *DevOps* movement does not stipulate that microservices are necessary for continuous deployment, nor microservices need to be packaged into containers. However, when those three are combined, new doors open waiting for us to step through. Recent developments in the area of containers and the concept of immutable deployments enable us to overcome many of the problems microservices had before. They, on the other hand, allow us to gain flexibility and speed without which CD is not possible or cost effective.

Before we move forward with this line of thinking, we'll try to define correctly each of those terms.

Continuous Integration

To understand *continuous deployment* we should first define its predecessors; *continuous integration* and *continuous delivery*.

Integration phase of a project development tended to be one of the most painful stages in software development life-cycle. We would spend weeks, months or even years working in separate teams dedicated to separate applications and services. Each of those teams would have their set of requirements and tried their best to meet them. While it wasn't hard to periodically verify each of those applications and services in isolation, we all dreaded the moment when team leads would decide that the time has come to integrate them into a unique delivery. Armed with the experience from previous projects, we knew that integration will be problematic. We knew that we will discover problems, unmet dependencies, interfaces that do not communicate with each others correctly and that managers will get disappointed, frustrated, and nervous. It was not uncommon to spend weeks or even months in this phase. The worse part of all that was that a bug found during the integration phase could mean going back and redoing days or weeks worth of work. If someone asked me how a feel about integration I'd say that it was closest I could get to becoming permanently depressed. Those were different times. We thought that was the "right" way to develop applications.

A lot changed since then. *Extreme Programming (XP)* and other agile methodologies become familiar, automated testing become frequent, and continuous integration started to take ground. Today we know that the way we developed software back then was wrong. The industry moved a long way since then.

Continuous integration (CI) usually refers to integrating, building, and testing code within the development environment. It requires developers to integrate code into a shared repository often. How often is often can be interpreted in many ways and it depends on the size of the team, the size of the project and the number of hours we dedicate to coding. In most cases it means that coders either push directly to the shared repository or merge their code with it. No matter whether we're pushing or merging, those actions should, in most cases, be done at least a couple of times a day. Getting code to the shared repository is not enough and we need to have a pipeline that, as a minimum, checks out the code and runs all the tests related, directly or indirectly, to the code corresponding to the repository. The result of the execution of the pipeline can be either *red* or *green*. Something failed, or everything was run without any problems. In the former case, minimum action would be to notify the person who committed the code.

The continuous integration pipeline should run on every commit or push. Unlike continuous delivery, continuous integration does not have a clearly defined goal of that pipeline. Saying that one application integrates with others does not tell us a lot about its production readiness. We do not know how much more work is required to get to the stage when the code can be delivered to production. All we are truly striving for is the knowledge that a commit did not break any of the existing tests. Never the less, CI is a huge improvement when done right. In many cases, it is a very hard practice to implement, but once everyone is comfortable with it, the results are often very impressive.

Integration tests need to be committed together with the implementation code, if not before. To gain maximum benefits, we should write tests in *test-driven development (TDD)* fashion. That way, not only that tests are ready for commit together with implementation, but we know that they are not faulty and would not pass no matter what we do. There are many other benefits TDD brings to the table and, if you haven't already, I strongly recommend to adopt it. You might want to consult the Test-Driven Development[4] section of the Technology Conversations[5] blog.

Tests are not the only CI prerequisite. One of the most important rules is that when the pipeline fails, fixing the problem has higher priority than any other task. If this action is postponed, next executions of the pipeline will fail as well. People will start ignoring the failure notifications and, slowly, CI process will begin losing its purpose. The sooner we fix the problem discovered during the execution of the CI pipeline, the better we are. If corrective action is taken immediately, knowledge about the potential cause of the problem is still fresh (after all, it's been only a few minutes between the commit and the failure notification) and fixing it should be trivial.

So how does it work? Details depend on tools, programming language, project, and many other factors. The most common flow is the following.

- Pushing to the code repository
- Static analysis
- Pre-deployment testing

[4] http://technologyconversations.com/category/test-driven-development/

[5] http://technologyconversations.com/

- Packaging and deployment to the test environment
- Post-deployment testing

Pushing to the Code Repository

Developers work on features in separate branches. Once they feel comfortable that their work is stable, the branch they've been working on is merged with the mainline (or trunk). More advanced teams may skip feature branches altogether and commit directly to the mainline. The crucial point is that the mainline branch (or trunk) needs to receive commits often (either through merges or direct pushes). If days or weeks pass, changes accumulate and benefits of using continuous integration diminish. In that case, there is no fast feedback since the integration with other people's code is postponed. On the other hand, CI tools (we'll talk about them later) are monitoring the code repository, and whenever a commit is detected, the code is checked out (or cloned) and the CI pipeline is run. The pipeline itself consists of a set of automated tasks run in parallel or sequentially. The result of the pipeline is either a failure in one of its steps or a promotion. As a minimum, failure should result in some form of a notification sent to the developer that pushed the commit that resulted in a failed pipeline. It should be his responsibility to fix the problem (after all, he knows best how to fix a problem created by him only minutes ago) and do another commit to the repository that, in turn, will trigger another execution of the pipeline. This developer should consider fixing the problem his highest priority task so that the pipeline continues being "green" and avoid failures that would be produced by commits from other developers. Try to keep a number of people who receive the failure notification to a minimum. The whole process from detecting a problem until it is fixed should be as fast as possible. The more people are involved, the more administrative work tends to happen and the more time is spent until the fix is committed. If, on the other hand, the pipeline runs successfully throughout all its tasks, the package produced throughout the process is promoted to the next stage and, in most cases, given to testers for manual verifications. Due to the difference in speed between the pipeline (minutes) and manual testing (hours or days), not every pipeline execution is taken by QAs.

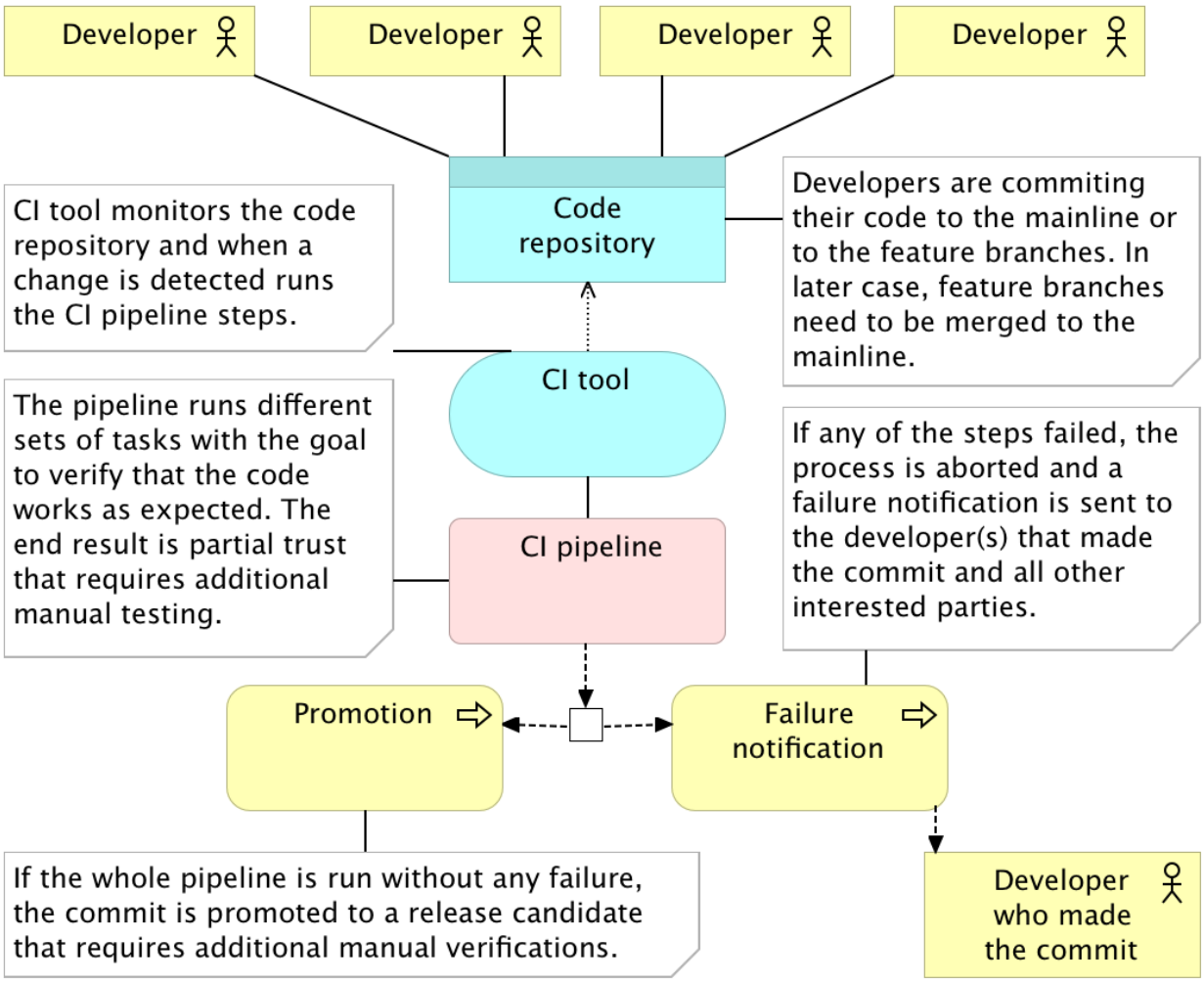

Figure 2-1: Continuous integration process

The first step in the continuous integration pipeline is often static analysis.

Static Analysis

Static analysis is the analysis of computer software that is performed without actually executing programs. Like its opposite, the analysis performed while executing programs is known as *dynamic analysis*.

The *static analysis* goals vary from highlighting possible coding errors to making sure that agreed formatting is followed. While benefits of using static analysis are questionable, the effort required to implement it is so small that there is no real reason not to use it.

I won't provide a comprehensive list of tools since they vary from one programming language to

another. CheckStyle[6] and FindBugs[7] for Java, JSLint[8] and JSHint[9] for JavaScript, and PMD[10] for a variety of languages, are only a few examples.

Static analysis is often the first step in the pipeline for the simple reason that its execution tends to be very fast and in most cases faster than any other step we have in the pipeline. All we have to do is choose the tools and often spend a little up-front time in setting up the rules we want them to use. From there on, the cost of the maintenance effort is close to nothing. Since it should not take more than few seconds to run this step, the cost in time is also negligible.

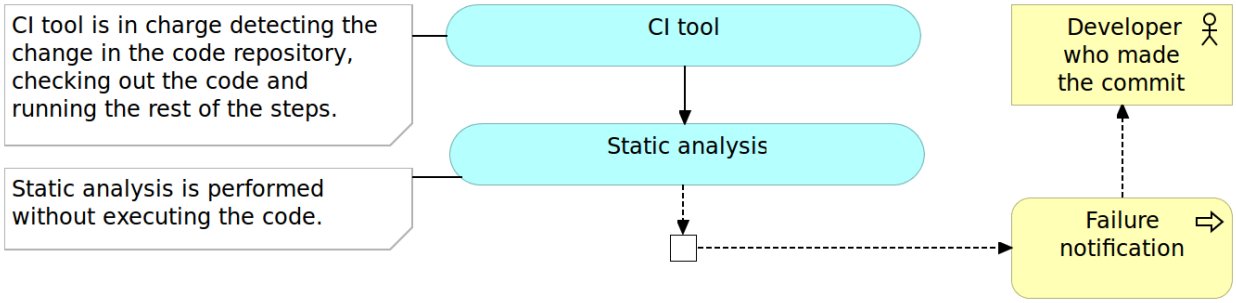

Figure 2-2: Continuous integration pipeline: static analysis

With the static analysis set up, our pipeline just started, and we can move to pre-deployment testing.

Pre-Deployment Testing

Unlike (optional) static analysis, *pre-deployment* tests should be mandatory. I intentionally avoided more specific name for those tests because it depends on the architecture, programming language, and frameworks. As a rule of thumb, all types of tests that do not require code to be deployed to a server should be run in this phase. *Unit tests* always fall into this category and with few others that might be run as well. If, for example, you can execute *functional tests* without deploying the code, run them now.

Pre-deployment testing is probably the most critical phase in *continuous integration pipeline*. While it does not provide all the certainty that we need, and it does not substitute *post-deployment testing*, tests run in this phase are relatively easy to write, should be very fast to execute and they tend to provide much bigger code coverage than other types of tests (for example integration and performance).

[6] http://checkstyle.sourceforge.net/
[7] http://findbugs.sourceforge.net/
[8] http://www.jslint.com/
[9] http://jshint.com/
[10] https://pmd.github.io/

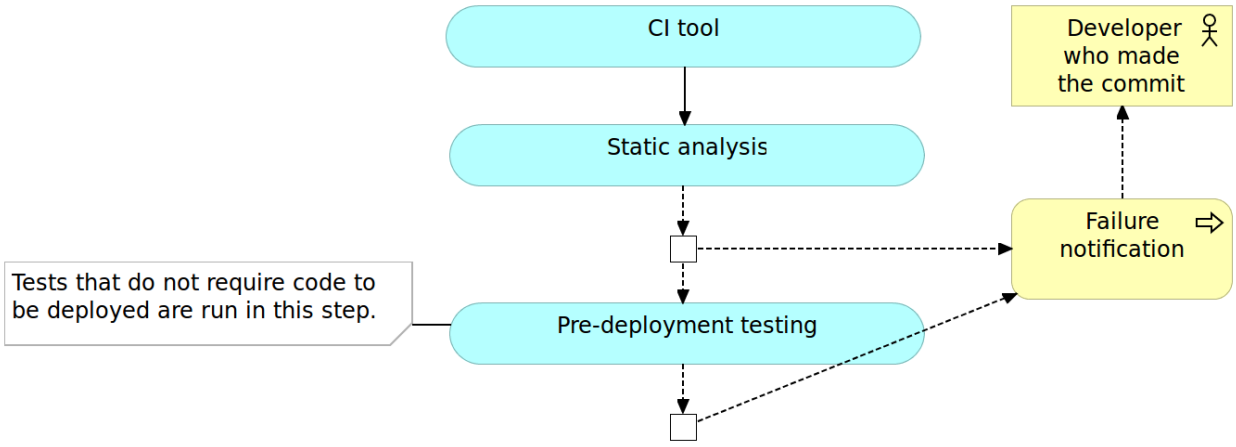

Figure 2-3: Continuous integration pipeline: pre-deployment testing

Packaging and Deployment to the Test Environment

Once we did all types of verifications that could be done without actually deploying the application, it is time to package it. The method to do it would depend on framework and programming language. In the Java world we would create JAR or WAR files, for JavaScript we would minimize the code and maybe send it to the CDN server, and so on and so forth. Some programming languages do not require us to do anything in this phase except possibly compress all the files into a ZIP or TAR a file for easier transfer to servers. An optional, but in the case of this book mandatory, step is to create a container that contains not only the package but also all other dependencies our application might need like libraries, runtime environment, application server, and so on.

Once the deployment package is created, we can proceed to deploy it to a test environment. Depending on the capacity of the servers you might need to deploy to multiple boxes with, for example, one being dedicated to performance testing and the other for all the rest of tests that require deployment.

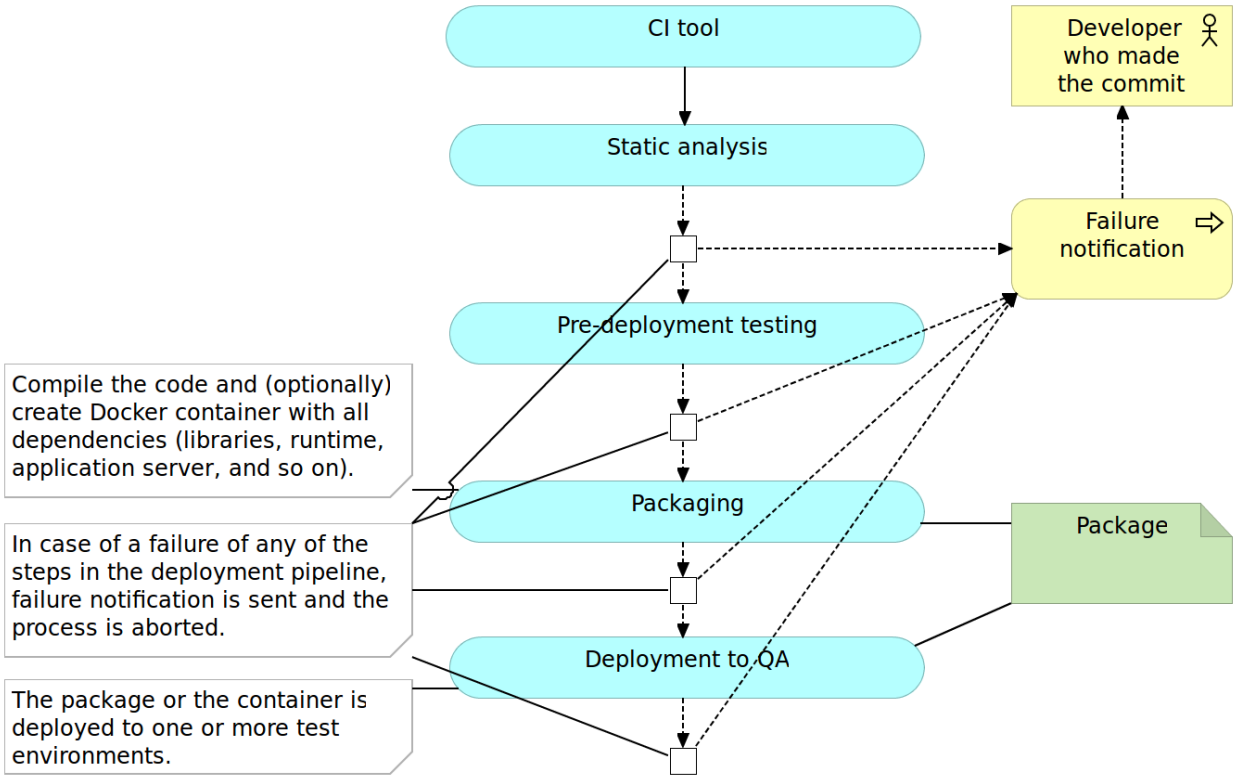

Figure 2-4: The continuous integration pipeline: packaging and deployment

Post-Deployment Testing

Once deployed to a test environment, we can execute the rest of the tests; those that could not be run without deploying the application or a service as well as those that prove that the integration was successful. Again, types of tests that can be run in this phase depend on frameworks and programming language but, as a general rule, they include functional, integration and performance tests.

Exact tools and technologies used to write and run those tests will depend on many aspects. My personal preference is to use *behavior-driven development* for all functional tests that, at the same time, act as acceptance criteria and Gatling[11] for performance tests.

Once the execution of post-deployment tests is finished successfully, the continuous integration pipeline is typically completed as well. Packages or artifacts we generated during the *packaging* and *deployment to test environment* are waiting for further, usually manual, verifications. Later on, one of the builds of the pipeline will be elected to be deployed to production. Means and details of additional checks and deployment to production are not part of continuous integration. Every build that passed the whole pipeline is considered integrated and ready for whatever comes next.

[11] http://gatling.io/

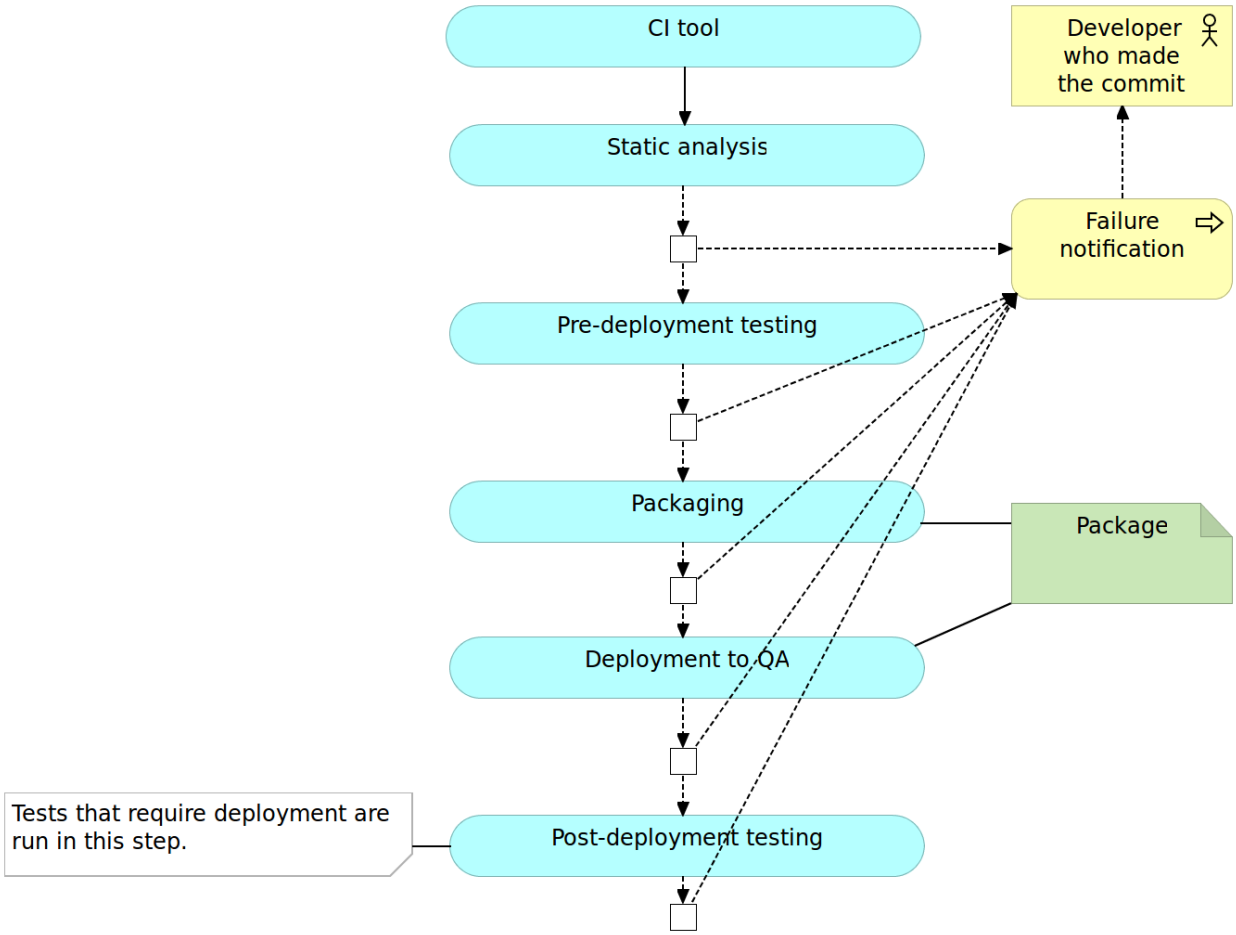

Figure 2-5: Continuous integration pipeline: post-deployment testing

Many other things could be done in the pipeline. The pipeline presented here is a very general one and often varies from case to case. For example, you might choose to measure code coverage and fail when a certain percentage is not reached.

We're not going into details right now but trying to get a general overview of the process so let us move into continuous delivery and deployment.

Continuous Delivery and Deployment

The *continuous delivery* pipeline is in most cases the same as the one we would use for CI. The major difference is in the confidence we have in the process and lack of actions to be taken after the execution of the pipeline. While CI assumes that there are (mostly manual) validations to be performed afterward, successful implementation of the CD pipeline results in packages or artifacts being ready to be deployed to production. In other words, every successful run of the pipeline *can be deployed to production*, no questions asked. Whether it will be deployed or not depends more on political than technical decisions. The marketing department might want to wait until a certain date,

or they might want to go live with a group of features deployed together. No matter the decision which build to deploy and when, from the technical perspective, the code of every successful build is fully finished. The only difference between the continuous integration and continuous delivery processes is that the latter does not have the manual testing phase that is performed after the package is promoted through the pipeline. Simply put, the pipeline itself provides enough confidence that there is no need for manual actions. With it, we are technically capable of deploying every promoted build. Which one of those will be deployed to production is a decision often based on business or marketing criteria where the company decides the right time to release a set of features.

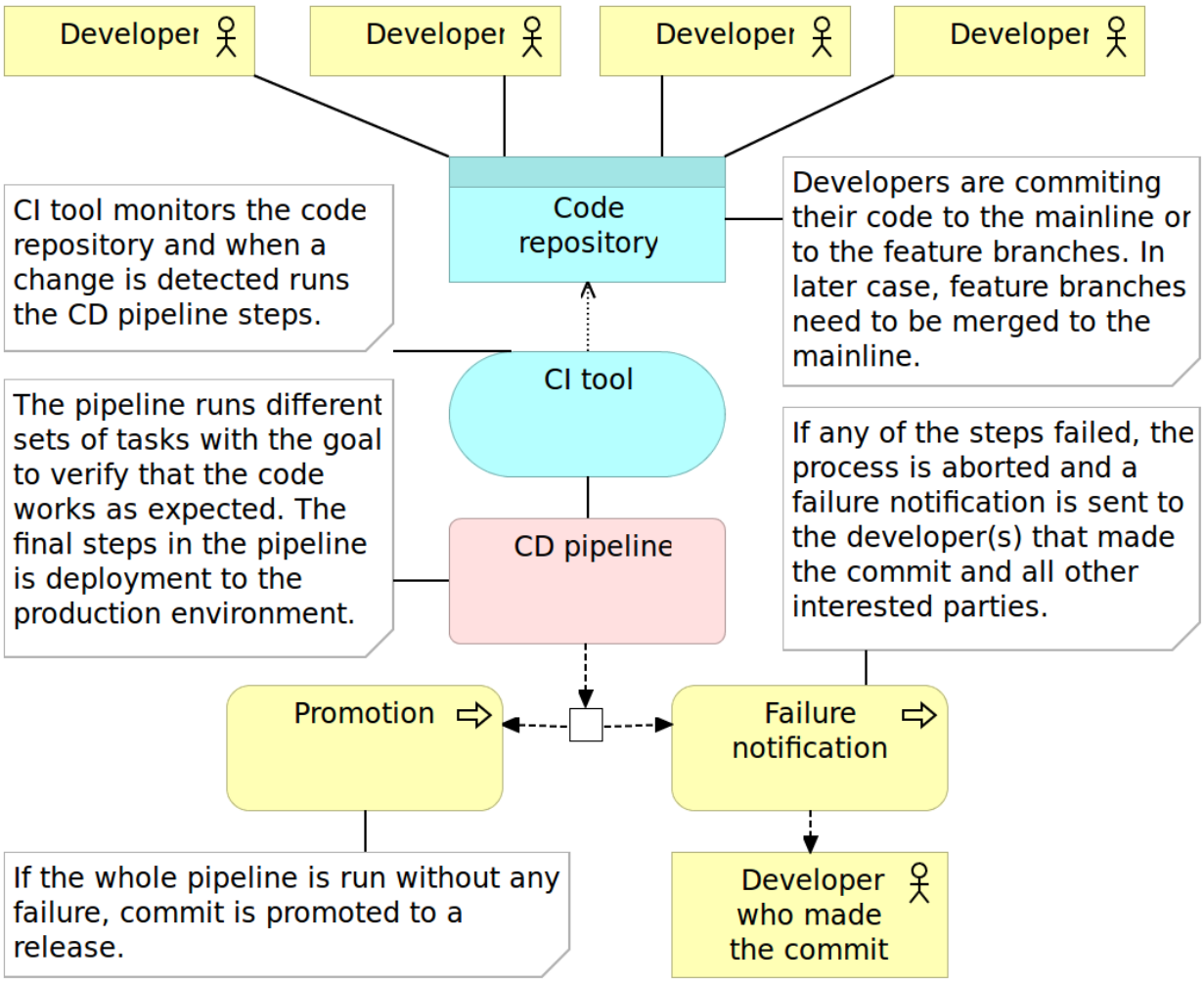

Figure 2-6: Continuous delivery process

Keep in mind that we continued using *CI tool* in the continuous delivery process diagram. The reason for this is a lack of any substantial difference between CI and CD tools. This does not mean that there are no products being marketed as CD tools - there are many. However, in my experience, this is more of a marketing stunt as both processes are almost the same assuming that processes rely on a high level of automation.

Regarding the pipeline process, there is also no substantial difference between continuous integration and continuous delivery. Both go through the same phases. The real difference is in the confidence we have in the process. As a result, the continuous delivery process does not have the manual QA phase. It's up to us to make a decision which one of the promoted packages will be deployed to production.

The *Continuous deployment* pipeline goes a step further and automatically deploys every build that passed all verifications. It is a fully automated process that starts with a commit to the code repository and ends with the application or the service being *deployed to production*. There is no human intervention, nothing to decide and nothing to do but to start coding the next feature while results of your work are finding their way to the users. In cases when packages are deployed to QA server before being deployed to production, post-deployment testing is done twice (or as many times are the number of servers we deploy to). In such a case, we might choose to run different subsets of post-deployment tests. For example, we might run all of them on the software deployed to QA server and only integration tests after deploying to production. Depending on the result of post-deployment tests, we might choose to roll-back or enable the release to the general public. When a proxy service is used to make a new release visible to the public, there is usually no need to roll-back since the newly released application was not made visible before the problem was detected.

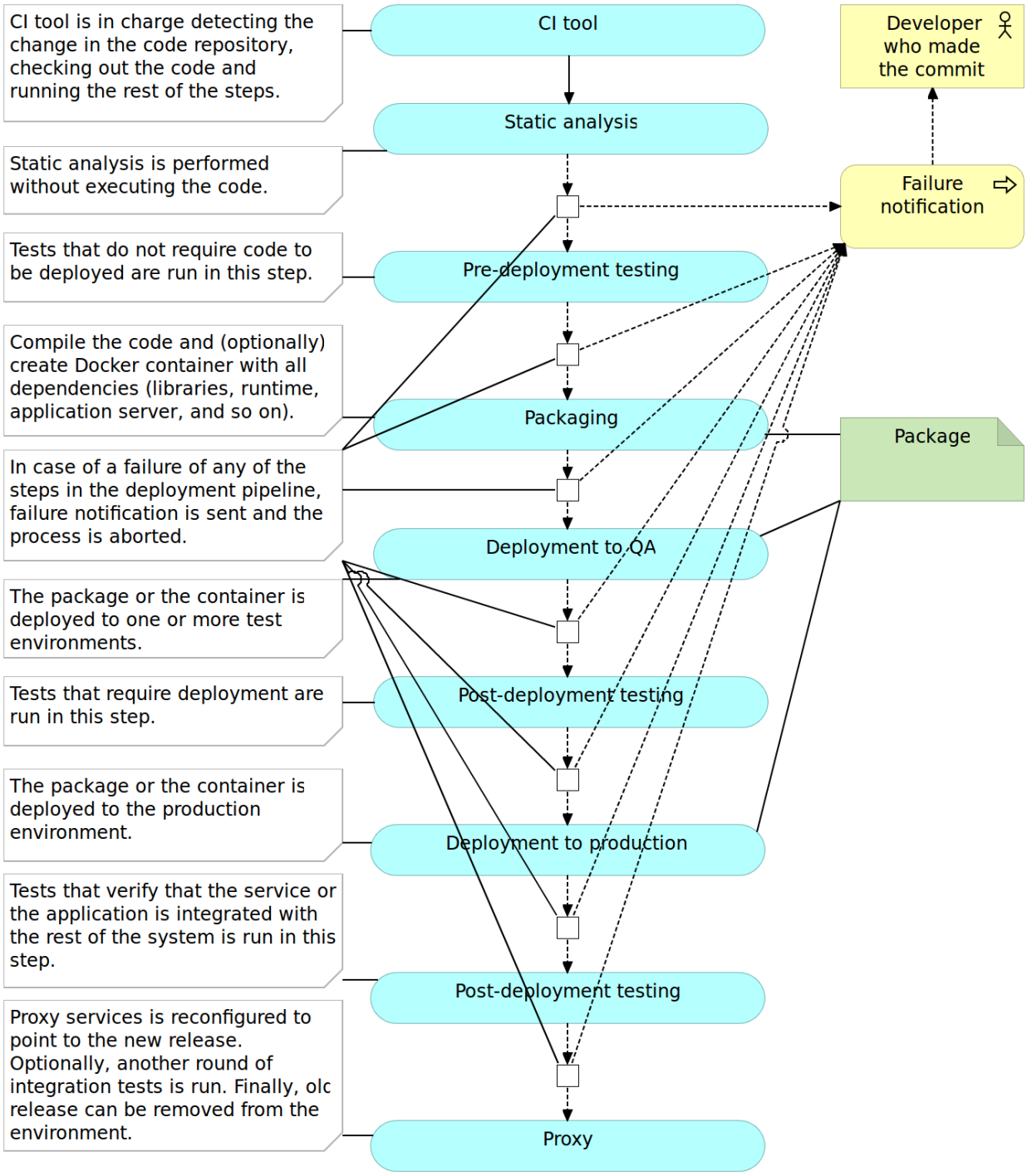

Figure 2-7: Continuous deployment pipeline

We need to pay particular attention to databases (especially when they are relational) and ensure that changes we are making from one release to another are backward compatible and can work with both releases (at least for some time).

While continuous integration welcomes, but does not necessarily require, deployed software to be tested in production, continuous delivery and deployment have production (mostly integration) testing as an absolute necessity and, in the case of continuous deployment, part of the fully automated pipeline. Since there are no manual verifications, we need to be as sure as possible that whatever was deployed to production is working as expected. That does not mean that all the automated tests need to be repeated. It means that we need to run tests that prove that the deployed software is integrated with the rest of the system. The fact that we run, possibly same, integration tests in other environments does not mean that due to some differences, software deployed to production continues to "play nicely" with the rest of the system.

Another very useful technique in the context of continuous deployment is *feature toggles*. Since every build is deployed to production, we can use them to disable some features temporarily. For example, we might have the login screen fully developed but without the registration. It would not make sense to let the visitors know about a feature that requires another still not deployed feature. Continuous delivery solves that problem by manually approving which build is deployed to production and would choose to wait. Since, in the case of continuous deployment that decision-making it not available, feature toggles are a must or we would need to delay merging with the mainline until all related features are finished. However, we already discussed the importance of constant merging with the mainline and such delays are against the logic behind CI/CD. While there are other ways to solve this problem, I find feature toggles to be indispensable to all those who choose to apply continuous deployment. We won't go into feature toggles details. For those interested obtaining more info, please visit the Feature Toggles (Feature Switches or Feature Flags) vs Feature Branches[12] article.

Most teams start with continuous integration and slowly move towards delivery and deployment since former are prerequisites for later. In this book, we'll practice continuous deployment. Don't be scared. Everything we'll do can be easily modified so that there are pauses and manual interventions. For example, we will be deploying containers directly to production (actually to VMs that imitate production) without passing through test environments. When applying techniques from this book, you can easily choose to add a testing environment in between.

The important thing to note is that the pipeline phases that we discussed are performed in particular order. That order is not only logical (for example, we cannot deploy before compiling) but also in order of the execution time. Things that take less to run are run first. For example, as a general rule, pre-deployment tests tend to run much faster than those we'll run as post-deployment. The same rule should be followed within each phase. If, for example, you have different types of tests within the pre-deployment phase, run those that are faster first. The reason for this quest for speed is time until we get feedback. The sooner we find out that there is something wrong with the commit, the better. Ideally, we should get that feedback before we move to the next development task. Do the commit, have a quick coffee, check your inbox and if there is no angry email stating that something failed, move to the next task.

Later on, throughout this book, you'll see that some of the phases and details of the presented pipeline are a bit different due to advantages brought by microservices and containers. For example,

[12] http://technologyconversations.com/2014/08/26/feature-toggles-feature-switches-or-feature-flags-vs-feature-branches/

packaging will finish with immutable (unchangeable) containers, deployment to a test environment might not be required at all, we might choose to perform testing directly to the production environment using the blue/green technique, and so on. However, I am ahead of myself. Everything will come in due time.

With CI/CD out of the way (for now), it is time to discuss microservices.

Microservices

We already spoke about speed in the context of continuous deployment. This speed refers to the time from conception of the idea for new functionality until it is fully operational and deployed to production. We want to be able to move fast and provide the shortest possible time to market. If a new functionality can be delivered in a matter of hours or days, business will start seeing benefits much faster than if it takes weeks or months.

Speed can be accomplished in multiple ways. For example, we want the pipeline to be as fast as possible both in order to provide quick feedback in case of a failure as well as to liberate resources for other queued jobs. We should aim at spending minutes instead of hours from checking out the code to having it deployed to production. Microservices can help accomplishing this timing. Running the whole pipeline for a huge monolithic application is often slow. Same applies to testing, packaging, and deployment. On the other hand, microservices are much faster for the simple reason that they are far smaller. There is less code to test, less code to package and less code to deploy.

We would not be switching to microservices if that were be the only reason. Later on, there will be a whole chapter dedicated to a much deeper examination of microservices. For now, the important thing to note is that due to the goals today's competition sets in front of us (flexibility, speed, and so on), microservices are probably the best type of architecture we can apply.

Containers

Before containers became common, microservices were painful to deploy. In comparison, monolithic applications are relatively simple to handle. We would, for example, create a single artifact (JAR, WAR, DLL, and so on), deploy it to the server and make sure that all required executables and libraries (for example JDKs) are present. This process was most of the time standardized, and had relatively few things to think about. One microservice is equally simple, but when their number multiplies with ten, hundred or even thousand, things start getting complicated. They might use different versions of dependencies, different frameworks, various application servers, and so on. The number of stuff we have to think about starts rising exponentially. After all, one of the reasons behind microservices is the ability to choose the best tool for the job. One might be better off if it's written in GoLang while the other would be a better fit for NodeJS. One could use JDK 7, while the other might need JDK 8. Installing and maintaining all that might quickly turn servers into garbage cans and make people in charge of them go crazy. The most common solution applied back then was standardizing as much as possible. Everyone must use only JDK 7 for the back-end. All front-end

has to be done with JSP. The common code should be placed in shared libraries. In other words, people tried to solve problems related to microservices deployment applying the same logic they learned during years of development, maintenance, and deployment of monolithic applications. Kill the innovation for the sake of standardization. And we could not blame them. The only alternative were immutable VMs and that only changed one set of problems for another. That is, until containers become popular and, more importantly, accessible to masses.

Docker[13] made it possible to work with containers without suffering in the process. They made containers accessible and easy to use to everyone.

What are containers? The definition of the word container is "an object for holding or transporting something". Most people associate containers with *shipping containers*. They should have strength suitable to withstand shipment, storage, and handling. You can see them being transported in a variety of ways, most common one of them being by ship. In big shipyards, you can find hundreds or even thousands of them stacked one besides the other and one on top of the other. Almost all merchandise is shipped through containers for a reason. They are standardized, easy to stack and hard to damage. Most involved with shipping do not know what's inside them. Nobody cares (except customs) because what is inside is irrelevant. The only important thing is to know where to pick them and where to deliver them. It is a clear separation of concerns. We know how to handle them from outside while their content is known only to those who packed them in the first place.

The idea behind "software" containers is similar. They are *isolated* and *immutable* images that provide designed functionality in most cases accessible only through their APIs. They are a solution to make our software run reliably and on (almost) any environment. No matter where they are running (developer's laptop, testing or production server, data center, and so on), the result should always be the same. Finally, we can avoid conversations like the following.

QA: There is a problem with the login screen.

Developer: It works on my computer!

The reason such a conversation is obsolete with containers is that they behave in the same way no matter the environment they're running on.

The way for containers to accomplish this feat is through *self-sufficiency* and *immutability*. Traditional deployments would put an artifact into an existing node expecting that everything else is in place; the application server, configuration files, dependencies, and so on. Containers, on the other hand, contain everything our software needs. The result is a set of images stacked into a container that contains everything from binaries, application server and configurations all the way down to runtime dependencies and OS packages. This description leads to the question about differences between a container and a VM. After all, all that we described by now is equally valid for both.

For example, a physical server running five virtual machines would have five operating systems in addition to a *hypervisor* that is more resource demanding than *lxc*. Five containers, on the other hand, share the operating system of the physical server and, where appropriate, binaries and libraries. As a result, containers are much more lightweight than VMs. With monolithic applications this is not

[13] https://www.docker.com/

so big of a difference, especially in cases when a single one would occupy the whole server. With microservices however, this gain in resource utilization is critical considering that we might have tens or hundreds of them on a single physical server. Put in other words, a single physical server can host more containers than virtual machines.

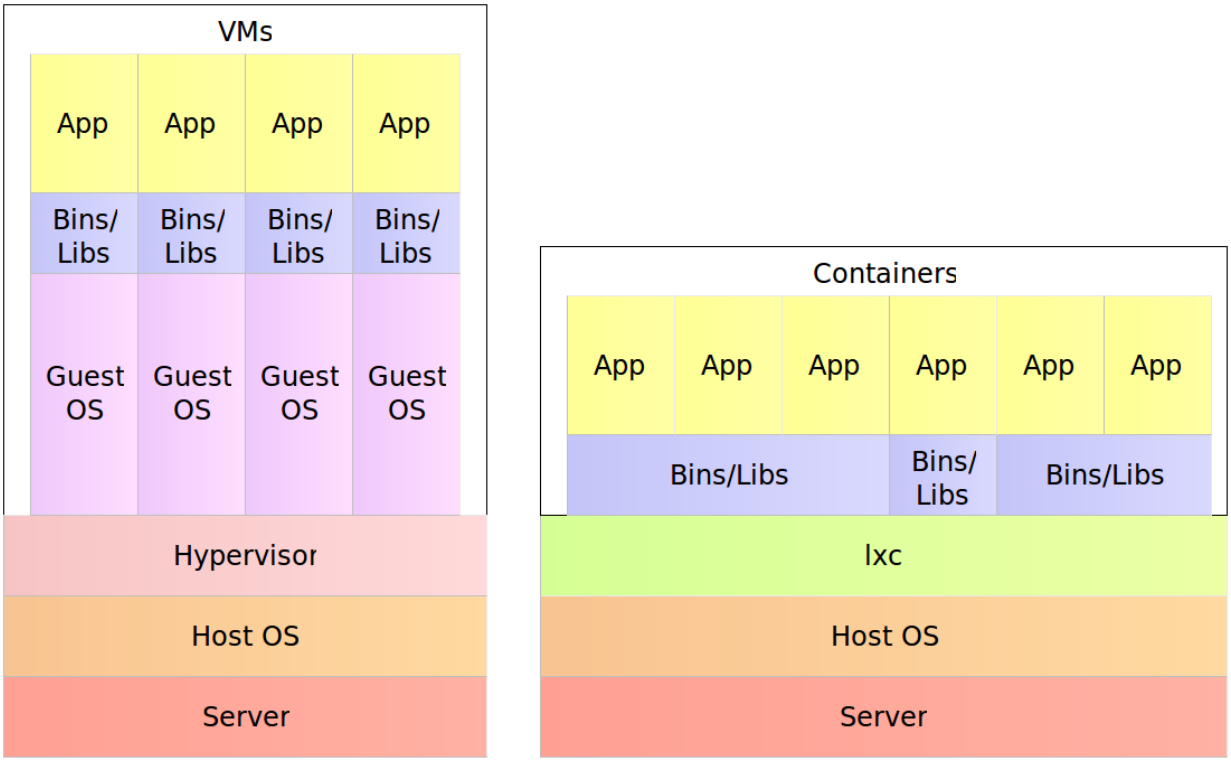

Figure 2-8: Virtual machines and containers resources utilization comparison

The Three Musketeers: Synergy of Continuous Deployment, Microservices, and Containers

Continuous deployment, microservices, and containers are a match made in heaven. They are like the three musketeers, each capable of great deeds but when joined, capable of so much more.

With continuous deployment, we can provide continuous and automatic feedback of our applications readiness and deployment to production, thus increasing the quality of what we deliver and decreasing the time to reach the market.

Microservices provide us with more freedom to make better decisions, faster development and, as we'll see very soon, easier scaling of our services.

Finally, containers provide the solution to many of deployment problems; in general and especially when working with microservices. They also increase reliability due to their immutability.

Together, they can combine all that and do so much more. Throughout this book, we'll be on a quest to *deploy often and fast, be fully automatic, accomplish zero-downtime, have the ability to rollback, provide constant reliability across environments, be able to scale effortlessly, and create self-healing systems able to recuperate from failures.* Any of those goals is worth a lot. Can we accomplish all of them? Yes! Practices and tools we have at our disposal can provide all that, and we just need to combine them correctly. The journey ahead is long but exciting. There are a lot of things to cover and explore and we need to start from the beginning; we'll discuss the architecture of the system we are about to start building.

Knowing is not enough; we must apply. Willing is not enough; we must do.

– Johann Wolfgang von Goethe

System Architecture

From here on, the whole book will be one big project. We'll go through all the stages starting from development all the way until production deployment and monitoring. Each phase will begin with a discussion about different paths we can take to accomplish the goal. We'll choose the best given our needs and implement it. The objective is to learn techniques that you can apply to your projects so please feel free to adapt instructions to fit your needs.

As most other projects, this one will start with high-level requirements. Our goal is to create an online shop. The complete plan is still not available, but we do know that selling books has priority. We should design services and a Web application in a way that it can easily be extended. We do not have the whole set of requirements in front of us, so we need to be prepared for the unknown. Besides books, we'll be selling other types of goods, and there will be other kinds of functionality like a shopping cart, registration and login, and so on. Our job is to develop bookstore and be able to respond to the future requirements in a fast manner. Since it is a new endeavor, not much traffic is expected at the beginning, but we should be prepared to scale easily and quickly if the service becomes successful. We want to ship new features as fast as possible without any downtime and to be able to recuperate from failures.

Let us start working on the architecture. It is clear that requirements are very general and do not provide many details. That means that we should be prepared for very likely changes in the future as well as requests for new features. At the same time, business requires us to build something small but be prepared to grow. How should we solve the problems given to us?

The first thing we should decide is how to define the architecture of the application we're about to build. Which approach will allow us possible changes of the direction, additional (but at this moment unknown) requirements and the need to be ready to scale? We should start by examining two most common approaches to applications architecture; monoliths and microservices.

Monolithic Applications

Monolithic applications are developed and deployed as a single unit. In the case of Java, the result is often a single WAR or JAR file. Similar statement is true for C++, .Net, Scala and many other programming languages.

Most of the short history of software development is marked by a continuous increment in size of the applications we are developing. As time passes, we're adding more and more to our applications continuously increasing their complexity and size and decreasing our development, testing and deployment speed.

We started dividing our applications into layers: presentation layer, business layer, data access layer, and so on. This separation is more logical than physical, and each of those layers tends to be in

charge of one particular type of operations. This kind of architecture often provided immediate benefits since it made clear the responsibility of each layer. We got separation of concerns on a high level. Life was good. Productivity increased, time-to-market decreased and overall clarity of the code base was better. Everybody seemed to be happy, for a while.

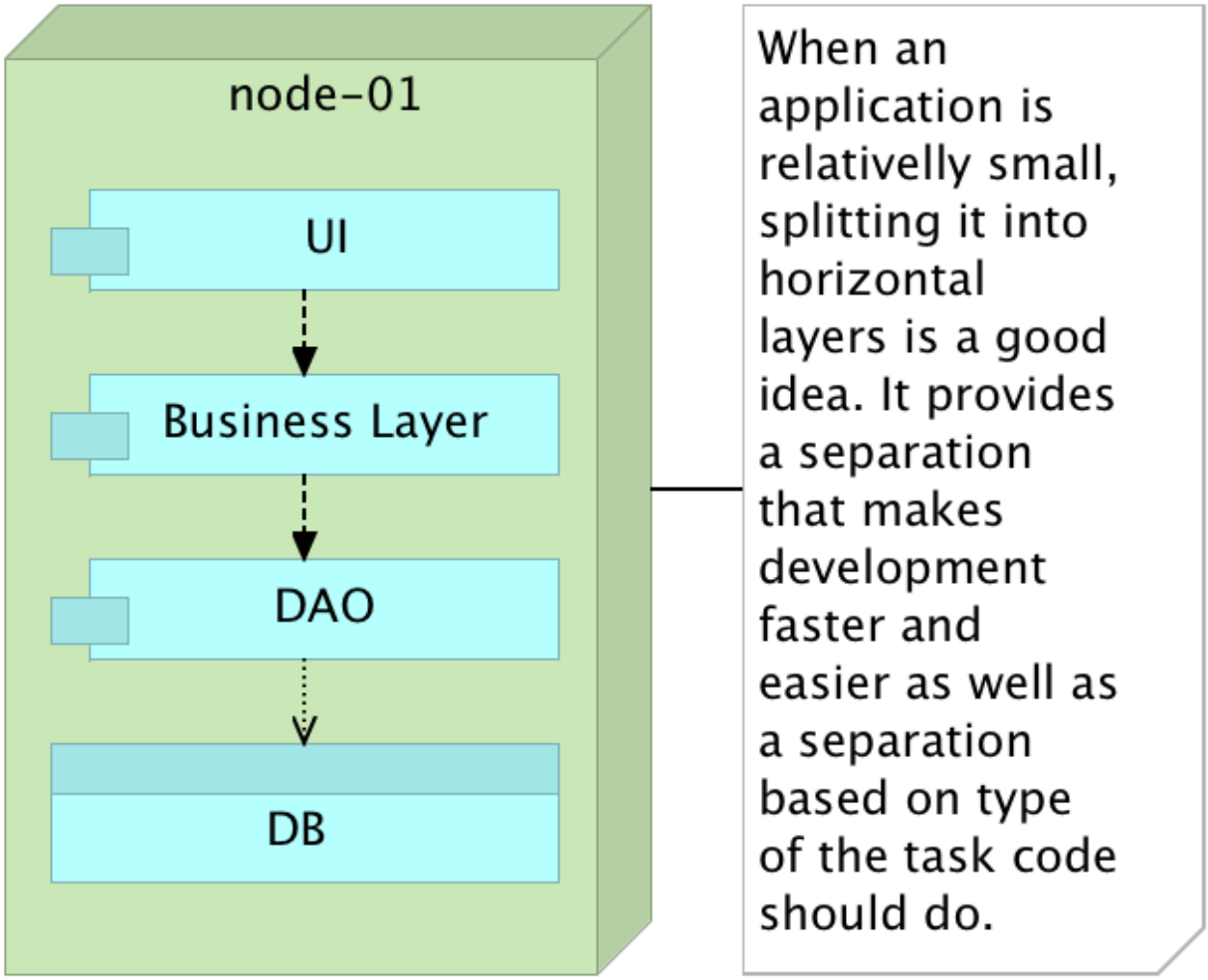

Figure 3-1: Monolithic application

With time, the number of features our application was required to support was increasing and with that comes increased complexity. One feature on UI level would need to speak with multiple business rules that in turn require multiple DAO classes that access many different database tables. No matter how hard we try, the sub-division within each layer and communication between them gets ever more complicated and, given enough time, developers start straying from the initial path. After all, a design made initially often does not pass the test of time. As a result, modifications to any given sub-section of a layer tends to be more complicated, time demanding and risky since they might affect many different parts of the system with often unforeseen effects.

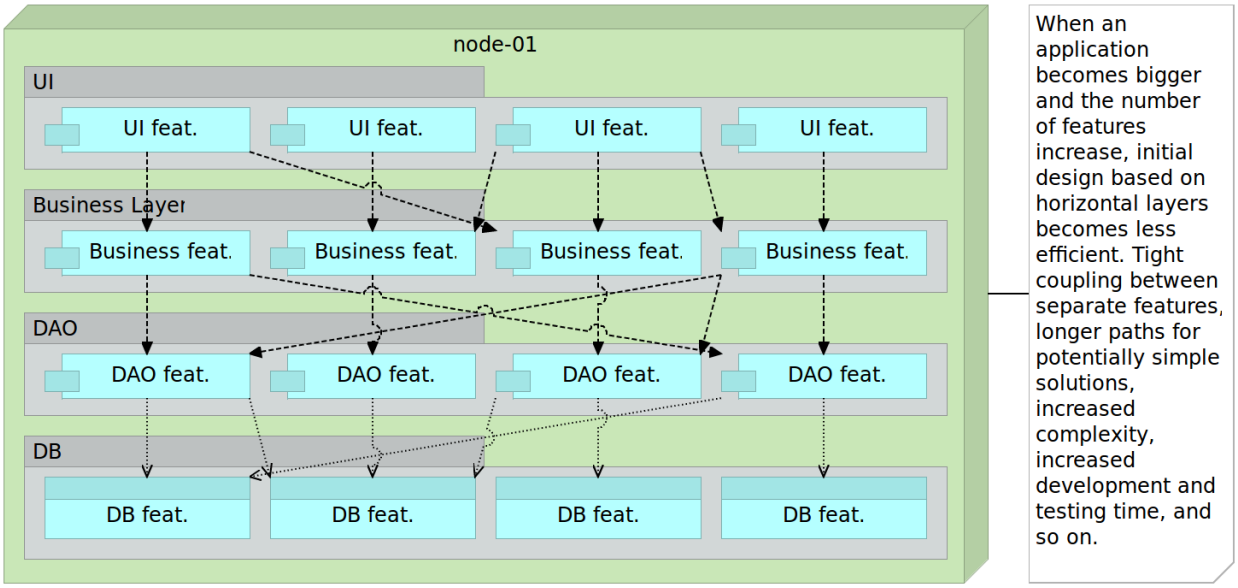

Figure 3-2: Monolithic application with increased number of features

As time passes, things start getting worse. In many cases, the number of layers increases. We might decide to add a layer with a rules engine, API layer, and so on. As things usually go, the flow between layers is in many cases mandatory. That results in situations where we might need to develop a simple feature that under different circumstances would require only a few lines of code but, due to the architecture we have, those few lines turn up to be hundreds or even thousands because all layers need to be passed through.

The development was not the only area that suffered from monolithic architecture. We still needed to test and deploy everything every time there was a change or a release. It is not uncommon in enterprise environments to have applications that take hours to test, build and deploy. Testing, especially regression, tends to be a nightmare that in some cases last for months. As time passes, our ability to make changes that affect only one module is decreasing. The primary objective of layers is to make them in a way that they can be easily replaced or upgraded. That promise is almost never actually fulfilled. Replacing something in big monolithic applications is hardly ever easy and without risks.

Scaling monoliths often mean scaling the entire application thus producing very unbalanced utilization of resources. If we need more resources, we are forced to duplicate everything on a new server even if a bottleneck is one module. In that case, we often end up with a monolith replicated across multiple nodes with a load balancer on top. This setup is sub-optimum at best.

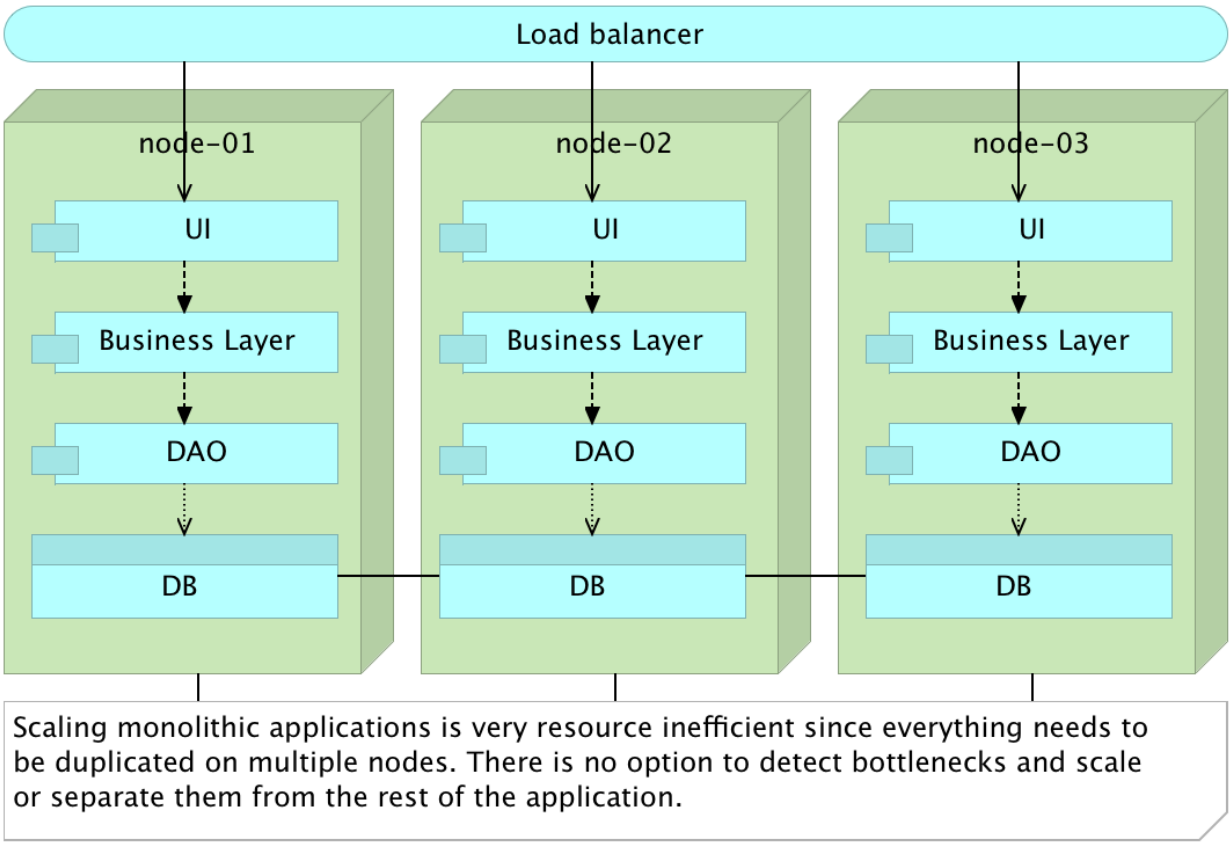

Figure 3-3: Scaling monolithic application

Services Split Horizontally

Service-oriented architecture (SOA) was created as a way to solve problems created by, often tightly coupled, monolithic applications. The approach is based on four main concepts we should implement.

- Boundaries are explicit
- Services are autonomous
- Services share schema and contract but not class
- Services compatibility is based on policy

SOA was such a big hit that many software providers jumped right in and created products that should help us in the transition. The most used type born out of SOA movement is *Enterprise Service Bus (ESB)*. At the same time, companies that experienced problems with monolithic applications and big systems jumped into the train and started the SOA transition with ESB as the locomotive.

However, the common problem with this move is the way we are used working that often resulted in an intention to artificially apply SOA architecture into the existing model.

We continued having the same layers as we had before, but this time physically separated from each other. There is an apparent benefit from this approach in that we can, at least, develop and deploy each layer independently from others. Another improvement is scaling. With the physical separation between what used to be layers, we are allowed to scale better. That approach was often combined with acquisitions of one of the enterprise service bus (ESB) products. In between services we would put ESB that would be in charge of transformation and redirection of requests from one service to another. ESB and similar products are beasts of their own and we often end up with another monolithic application that is as big or even bigger than the one we tried to split. What we needed was to *break services by bounded contexts* and separate them physically with each running in their own process and with clearly defined communication between them. Thus, microservices were born.

Microservices

Microservices are an approach to architecture and development of a single application composed of small services. The key to understanding microservices is their independence. Each is developed, tested and deployed separately from each other. Each service runs as a separate process. The only relation between different microservices is data exchange accomplished through APIs they are exposing. They inherit, in a way, the idea of small programs and pipes used in Unix/Linux. Most Linux programs are small and produce some output. That output can be passed as input to other programs. When chained, those programs can perform very complex operations. It is complexity born from a combination of many simple units.

In a way, microservices use the concepts defined by SOA. Then why are they called differently? SOA implementations went astray. That is especially true with the emergence of ESB products that themselves become big and complex enterprise applications. In many cases, after adopting one of the ESB products, the business went as usual with one more layer sitting on top of what we had before. Microservices movement is, in a way, reaction to misinterpretation of SOA and the intention to go back to where it all started. The main difference between SOA and microservices is that the latter should be self-sufficient and deployable independently of each other while SOA tends to be implemented as a monolith.

Let's see what Gartner has to say about microservices. While I'm not a big fan of their predictions, they do strike the important aspect of the market by appealing to big enterprise environments. Their evaluations of market tendencies usually mean that we passed the adoption by greenfield projects, and the technology is ready for the big enterprises. Here's what Gary Olliffe said about microservices at the beginning of 2015.

Microservice architectures promise to deliver flexibility and scalability to the development and deployment of service-based applications. But how is that promise delivered? In short, by adopting

an architecture that allows individual services to be built and deployed independently and dynamically; an architecture that embraces DevOps practices.

Microservices are simpler, developers get more productive and systems can be scaled quickly and precisely, rather than in large monolithic globs. And I haven't even mentioned the potential for polyglot coding and data persistence.

Key aspects of microservices are as follows.

- They do one thing or are responsible for one functionality.
- Each microservice can be built by any set of tools or languages since each is independent of others.
- They are truly loosely coupled since each microservice is physically separated from others.
- Relative independence between different teams developing different microservices (assuming that APIs they expose are defined in advance).
- Easier testing and continuous delivery or deployment.

One of the problems with microservices is the decision when to use them. In the beginning, while the application is still small, problems that microservices are trying to solve do not exist. However, once the application grows and the case for microservices can be made, the cost of switching to a different architecture style might be too big. Experienced teams tend to use microservices from the very start knowing that technical debt they might have to pay later will be more expensive than working with microservices from the very beginning. Often, as it was the case with Netflix, eBay, and Amazon, monolithic applications start evolving towards microservices gradually. New modules are developed as microservices and integrated with the rest of the system. Once they prove their worth, parts of the existing monolithic application gets refactored into microservices as well.

One of the things that often gets most critique from developers of enterprise applications is decentralization of data storage. While microservices can work (with few adjustments) using centralized data storage, the option to decentralize that part as well should, at least, be explored. The option to store data related to some service in a separate (decentralized) storage and pack it together into the same container or as a separate one and link them together is something that in many cases could be a better option than storing that data in a centralized database. I am not proposing always to use decentralized storage but to have that option in account when designing microservices.

Finally, we often employ some kind of a lightweight proxy server that is in charge of the orchestration of all requests no matter whether they come from outside or from one microservice to another.

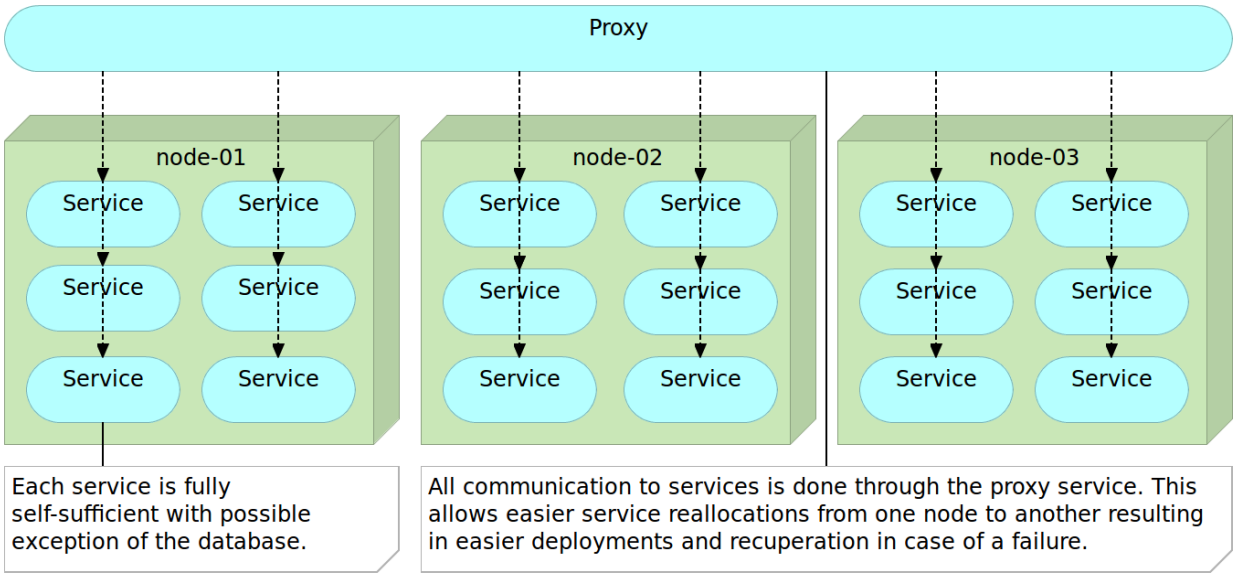

Figure 3-4: Microservices with a proxy service

Armed with a basic knowledge about monoliths and microservices, let us compare the two and evaluate their strengths and weaknesses.

Monolithic Applications and Microservices Compared

From what we learned by now, seems that microservices are a better option than monoliths. Indeed, in many (but far from all) cases they are. However, there is no such thing as a free lunch. Microservices have their set of disadvantages with *increased operational and deployment complexity*, and *remote process calls* being the most common.

Operational and Deployment Complexity

The primary argument against microservices is *increased operational and deployment complexity*. This argument is correct, but thanks to relatively new tools it can be mitigated. *Configuration Management (CM)* tools can handle environment setups and deployments with relative ease. Utilization of *containers with Docker* significantly reduces deployment pains that microservices can cause. CM tools together with containers allow us to deploy and scale microservices quickly.

In my opinion, increased deployment complexity argument usually does not take into account advances we saw during last years and is greatly exaggerated. That does not mean that part of the work is not shifted from development to DevOps. It is. However, benefits are in many cases bigger than the inconvenience that change produces.

Remote Process Calls

Another argument for monolithic applications is *reduced performance* produced by microservices' *remote process calls*. Internal calls through classes and methods are faster and this problem cannot be removed. How much that loss of performance affects a system depends on case to case basis. The important factor is how we split our system. If we take it towards the extreme with very small microservices (some propose that they should not have more than 10-100 lines of code), this impact might be considerable. I like to create microservices organized around bounded contexts or functionality like users, shopping cart, products, and so on. That reduces the number of remote process calls but still keep services organization within healthy boundaries. Also, it's important to note that if calls from one microservice to another are going through a fast internal LAN, the negative impact is relatively small.

So, what are the advantages microservices have over monoliths? The following list is by no means final nor it represents advantages only available with microservices. While many of them are valid for other types of architecture, they are more prominent with microservices.

Scaling

Scaling microservices is much easier than monolithic applications. With monoliths, we duplicate the whole application into a new machine. On the other hand, with microservices, we duplicate only those that need scaling. Not only that *we can scale what needs to be scaled* but we can distribute things better. We can, for example, put a service that has heavy utilization of CPU together with another one that uses a lot of RAM while moving the other CPU demanding service to a different hardware.

Innovation

Monolithic applications, once the initial architecture is made, do not leave much space for innovation. I'd go even further and claim that *monoliths are innovation killers*. Due to their nature, changing things takes time, and experimentation is perilous since it potentially affects everything. One cannot, for example, change Apache Tomcat for NodeJS just because it would better suit one particular module.

I'm not suggesting that we should change programming language, server, persistence, and other architecture aspects for each module. However, monolithic servers tend to go to an opposite extreme where changes are risky if not unwelcome. With microservices, we can choose what we think is the best solution for each service separately. One might use Apache Tomcat while the other could use NodeJS. One can be written in Java and the other in Scala. I'm not advocating that each service is different from the rest but that each can be made in a way we think is best suited for the goal at hand. On top of that, changes and experiments are much easier to do. After all, whatever we do affects only one out of many microservices and not the system as a whole as long as the API is respected.

Size

Since *microservices are small*, they are much easier to understand. There is much less code to go through to see what one microservice is doing. That in itself greatly simplifies development, especially when newcomers join the project. On top of that, everything else tends to be much faster. IDEs work faster with a small project when compared to big ones used in monolithic applications. They start faster since there are no huge servers nor an enormous number of libraries to load.

Deployment, Rollback and Fault Isolation

Deployment is much faster and easier with microservices. Deploying something small is always quicker (if not easier) than deploying something big. In case we realized that there is a problem, that problem has potentially limited effect and can be rolled back much easier. Until we roll back, the fault is isolated to a small part of the system. Continuous delivery or deployment can be done with speed and frequencies that would not be possible with big applications.

Commitment Term

One of the common problems with monolithic applications is commitment. We are often forced to choose from the start the architecture and the technologies that will last for a long time. After all, we're building something big that should last for a long time. With microservices that *need for a long-term commitment is much smaller*. Change the programming language in one microservice and if it turns out to be a good choice, apply it to others. If the experiment failed or is not the optimum, there's only one small part of the system that needs to be redone. Same applies to frameworks, libraries, servers, and so on. We can even use different databases. If some lightweight NoSQL seems like the best fit for a particular microservice, why not use it and pack it into the container?

Let us go one step back and look at this subject from the prism of deployment. How do those two architectural approaches differ when the time comes to deploy our applications.

Deployment Strategies

We already discussed that continuous delivery and deployment strategies require us to rethink all aspects of the application lifecycle. That is nowhere more noticeable than at the very beginning when we are faced with architectural choices. We won't go into details of every possible deployment strategy we could face but limit the scope to two major decisions that we should make. First one is architecturally related to the choice between monolithic applications and microservices. The second one is related to how we package the artifacts that should be deployed. More precisely, whether we should perform mutable or immutable deployments.

Mutable Monster Server

Today, the most common way to build and deploy applications is as a *mutable monster server*. We create a web server that has the whole application and update it every time there is a new release. Changes can be in *configuration* (properties file, XMLs, DB tables, and so on), *code artifacts* (JARs, WARs, DLLs, static files, and so on) and *database schemas and data*. Since we are changing it on every release, it is mutable.

With mutable servers, we cannot know for sure that development, test, and production environments are the same. Even different nodes in the production might have undesirable differences. Code, configuration or static files might not have been updated in all instances.

It is a *monster server* since it contains everything we need as a single instance. Back-end, front-end, APIs, and so on. Moreover, it grows over time. It is not uncommon that after some time no one is sure what is the exact configuration of all pieces in production and the only way to accurately reproduce it somewhere else (new production node, test environment, and so on) is to copy the VM where it resides and start fiddling with configurations (IPs, host file, DB connections, and so on). We just keep adding to it until we lose the track of what it has. Given enough time, your "perfect" design and impressive architecture will become something different. New layers will be added, the code will be coupled, patches on top of patches will be created and people will start losing themselves in the maze the code start looking like. Your beautiful little project will become a big monster. The pride you have will become a subject people talk about on coffee breaks. People will start saying that the best thing they could do is to throw it to trash and start over. But, the monster is already too big to start over. Too much is invested. Too much time would be needed to rewrite it. Too much is at stake. Our monolith might continue existing for a long time.

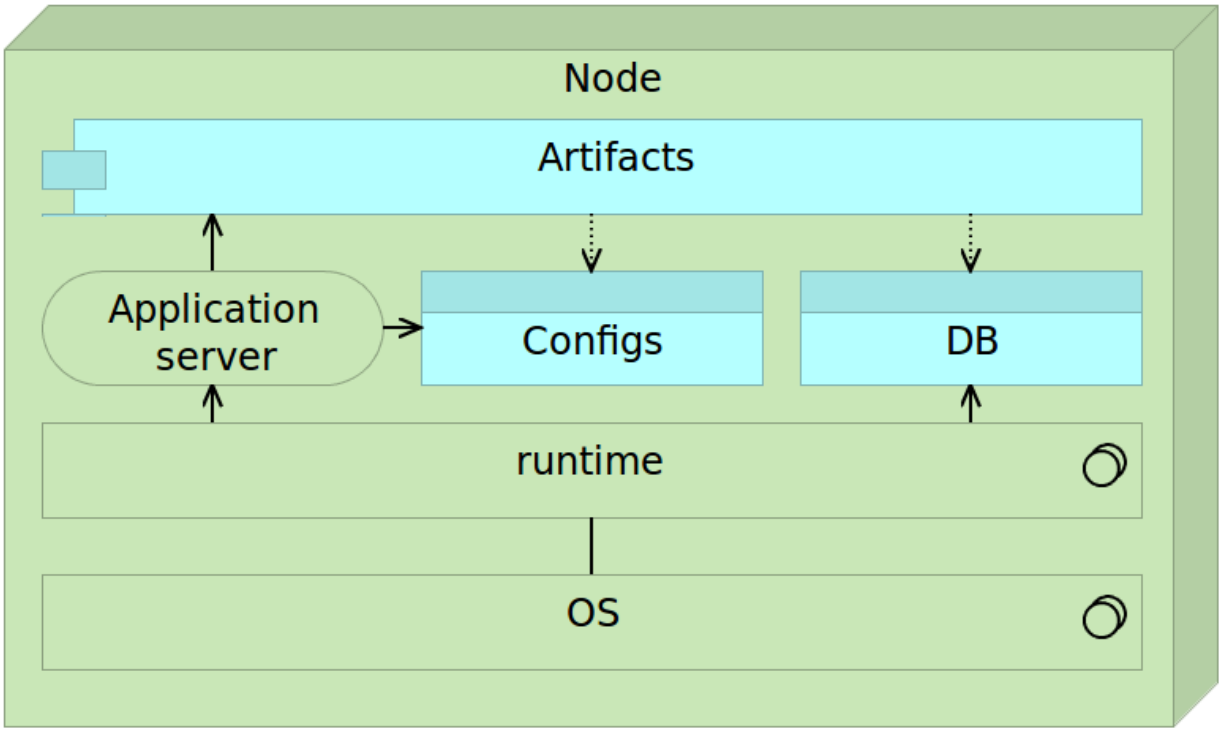

Figure 3-5: Mutable application server as initially designed

Mutable deployments might look simple, but they are usually not. By coupling everything into one place, we are hiding complexity thus increasing the chance of discrepancies between different instances.

Time to restart such a server when it receives a new release can be considerable. During that time server is usually not operational. Downtime that the new release provokes is a loss of money and trust. Today's business expects us to operate 24/7 without any downtime, and it is not uncommon that a release to production means night work of the team during which our services are not available. Given such a situation, applying continuous deployment is a dream out of the reach. It is a dream that can not become a reality.

Testing is also a problem. No matter how much we tested the release on development and test environments, the first time it will be tried in production is when we deploy it and make it available not only to our testers but also to all our users.

Moreover, fast rollback of such a server is close to impossible. Since it is mutable, there is no "photo" of the previous version unless we create a snapshot of a whole virtual machine that brings up a whole new set of problems.

By having architecture like this, we cannot fulfill all, if any, of the requirements described earlier. We cannot deploy often, due to inability to produce zero-downtime and easily rollback. Full automation is risky due to mutable nature of its architecture thus preventing us to be fast.

By not deploying often we are accumulating changes that will be released and, in that way, we are

increasing the probability of a failure.

To solve those problems, deployments should be immutable and composed of small, independent, and self-sufficient applications. Remember, our goals are to deploy often, have zero-downtime, be able to rollback any release, be automated and be fast. Moreover, we should be able to test the release on production environment before users see it.

Immutable Server and Reverse Proxy

Each "traditional" deployment introduces a risk tied with changes that need to be performed on the server. If we change our architecture to immutable deployments, we gain immediate benefits. Provisioning of environments becomes much simpler since there is no need to think about applications (they are unchangeable). Whenever we deploy an image or a container to the production server, we know that it is precisely the same as the one we built and tested. Immutable deployments reduce the risk tied to unknown. We know that each deployed instance is exactly the same as the other. Unlike mutable deployment, when a package is immutable and contains everything (application server, configurations, and artifacts) we stop caring about all those things. They were packaged for us throughout the deployment pipeline and all we have to do is make sure that the immutable package is sent to the destination server. It is the same package as the one we already tested in other environments and inconsistencies that could be introduced by mutable deployments are gone.

A reverse proxy can be used to accomplish zero-downtime. Immutable servers together with a reverse proxy in a simplified form can be as follows.

First we start with a reverse proxy that points to our fully self-sufficient immutable application package. This package could be a virtual machine or a container. We'll refer to this application as application image to establish a clear distinction from mutable applications. On top of the application is a proxy service that routes all the traffic towards the final destination instead of exposing the server directly.

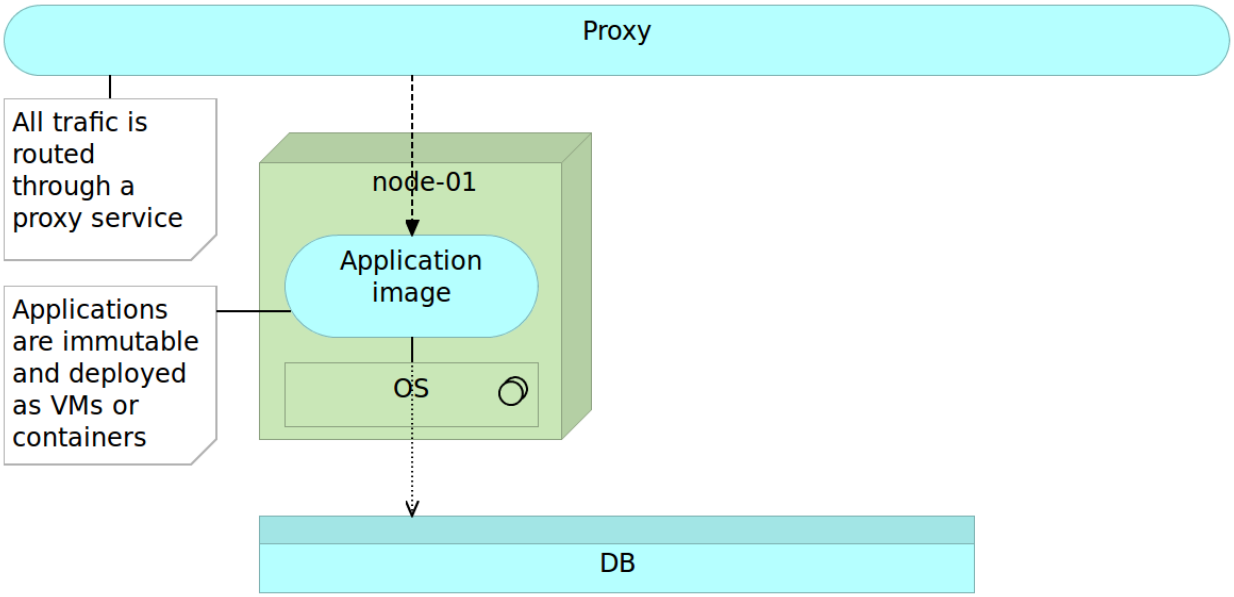

Figure 3-6: Immutable application server deployed as an image (a virtual machine or a container)

Once we decide to deploy a new version, we do it by deploying a separate image to a separate server. While in some cases we could deploy this image to the same server, more often than not, monolithic applications are very resource demanding and we cannot have both on the same node without affecting the performance. At this moment, we have two instances. One old (previous release) and one new (latest release). All traffic still goes to the old server through the reverse proxy so users of our application still do not notice any change. For them, we're still running the old and proven software. This is the right moment to execute the final set of tests. Preferably those tests are automatic and part of the deployment process but manual verification is not excluded. For example, if changes were done to front-end, we might want to do the final round of user experience tests. No matter what types of tests are performed, they should all "attack" the new release bypassing the reverse proxy. The good thing about those tests is that we are working with the future production version of the software that resides on production hardware. We are testing production software and hardware without affecting our users (they are still being redirected to the old version). We could even enable our new release only to a limited number of users in the form of A/B testing.

To summarize, at this stage we have two instances of the server, one (the previous release) used by our users and the other (the latest release) used for testing.

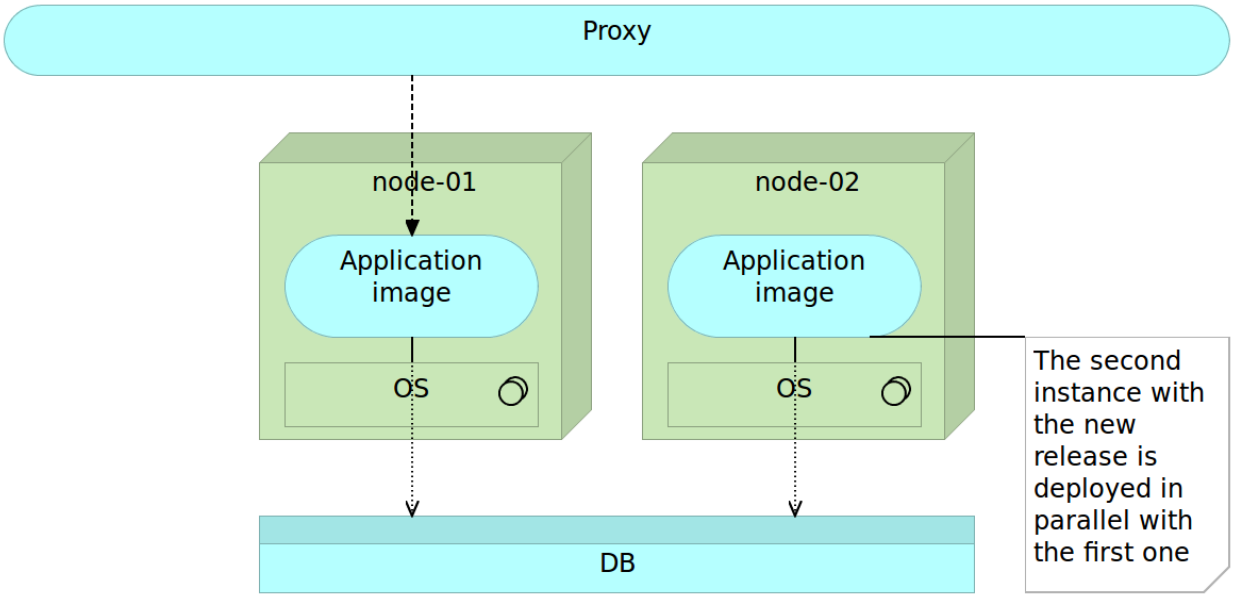

Figure 3-7: New release of the immutable application deployed to a separate node

Once we are finished with tests and are confident that the new release works as expected, all we have to do is change the reverse proxy to point to the new release. The old one can stay for a while in case we need to rollback the changes. However, for our users, it does not exist. All traffic is routed to the new release. Since the latest release was up-and-running before we changed the routing, the switch itself will not interrupt our service (unlike, for example, if we would need to restart the server in case of mutable deployments). When the route is changed we need to reload our reverse proxy. As an example, *nginx* maintains old connections until all of them are switched to the new route.

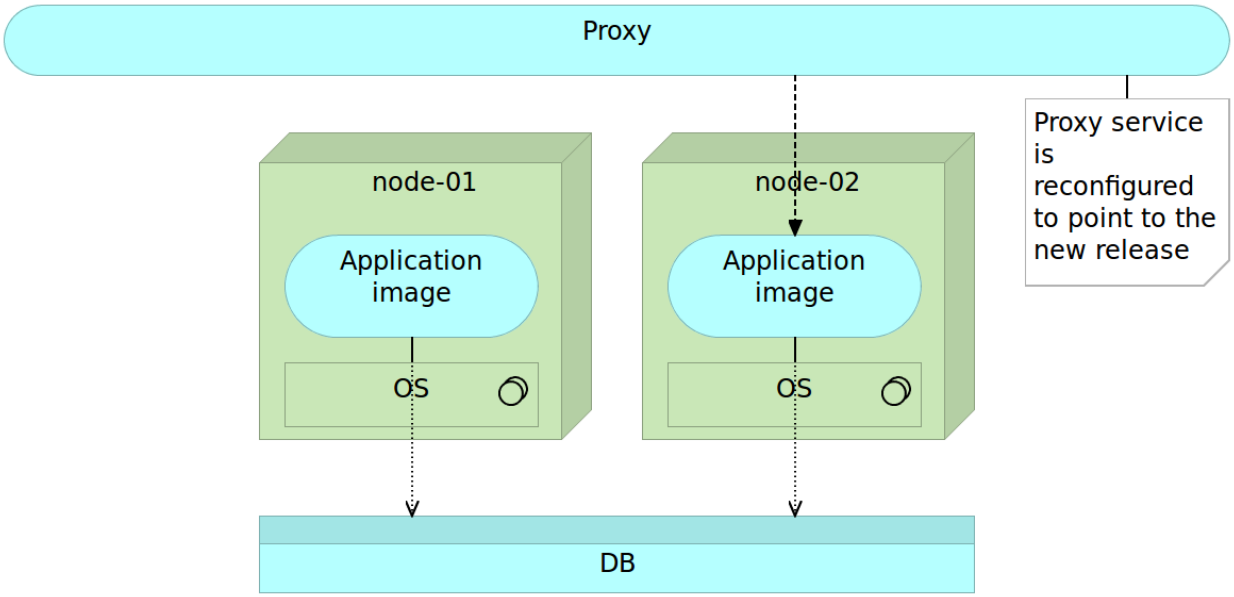

Figure 3-8: Poxy is rerouted to point to the new release

Finally, when we do not need the old version, we can remove it. Even better, we can let the next release remove it for us. In the latter case, when the time comes, release process will remove the older release and start the process all over again.

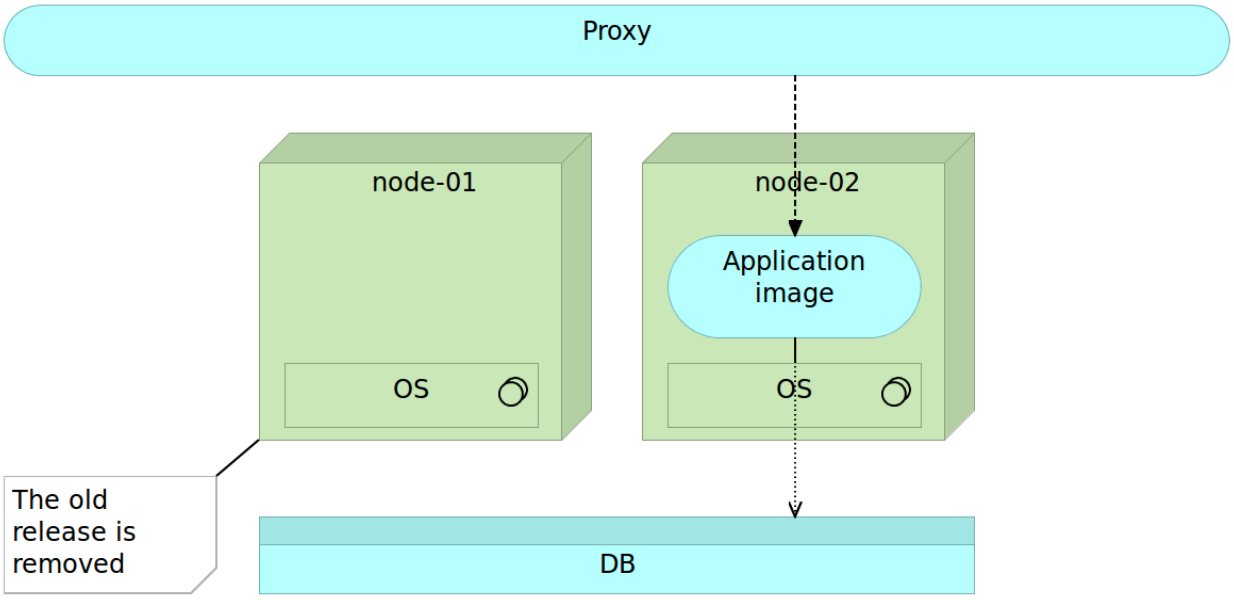

Figure 3-9: The old release is removed

The technique described above is called blue-green deployment and has been in use for a long time. We'll be practicing it later on when we reach the Docker packaging and deployment examples.

Immutable Microservices

We can do even better than this. With immutable deployments, we can easily accomplish automatism of the process. Reverse proxy gives us zero-downtime and, having two releases up and running allows us to rollback easily. However, since we're still dealing with one big application, deployment and tests might take a long time to run. That in itself might prevent us from being fast and thus from deploying as often as needed. Moreover, having everything as one big server increases development, testing and deployment complexity. If things could be split into smaller pieces, we might divide complexity into easily manageable chunks. As a bonus, having small independent services would allow us to scale more easily. They can be deployed to the same machine, scaled out across the network or multiplied if the performance of one of them becomes the bottleneck. Microservices to the rescue!

With "monster" applications we tend to have decoupled layers. Front-end code should be separated from the back-end, business layer from data access layer, and so on. With microservices, we should start thinking in a different direction. Instead of having the business layer separated from the data access layer, we would separate services. For example, users management could be split from the sales service. Another difference is physical. While traditional architecture separates on a level of packages and classes but still deploys everything together, microservices are split physically; they might not even reside on the same physical machine.

Deployment of microservices follows the same pattern as previously described.

We deploy our microservice immutable image as any other software.

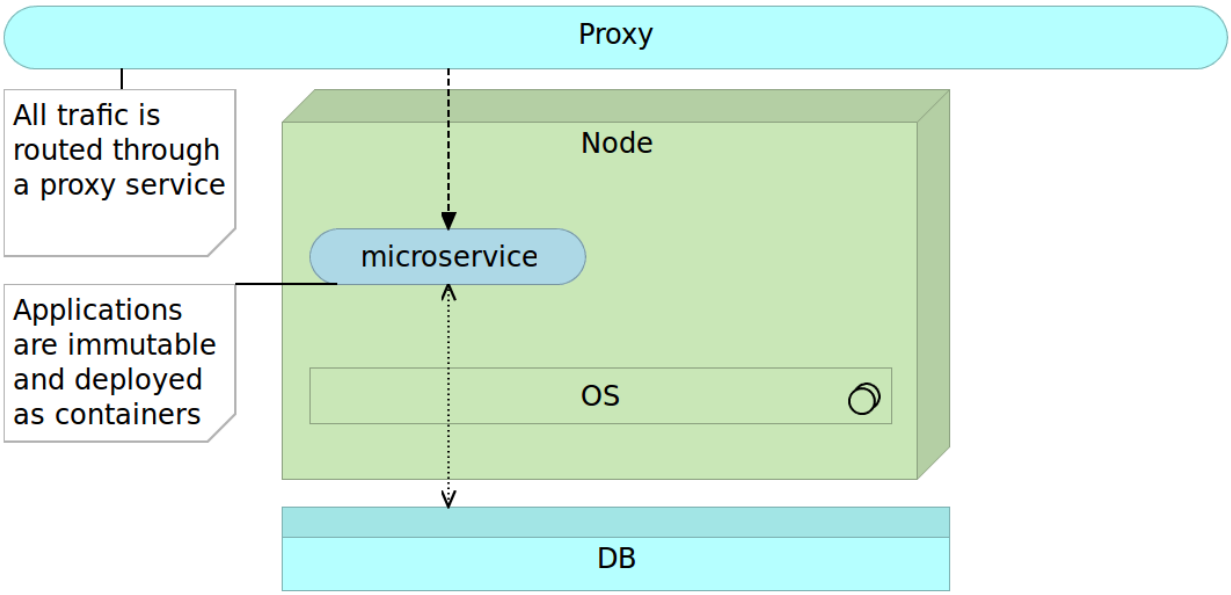

Figure 3-10: Immutable microservice deployed as an image (a virtual machine or a container)

When the time comes to release a new version of some microservice we deploy it alongside the older version.

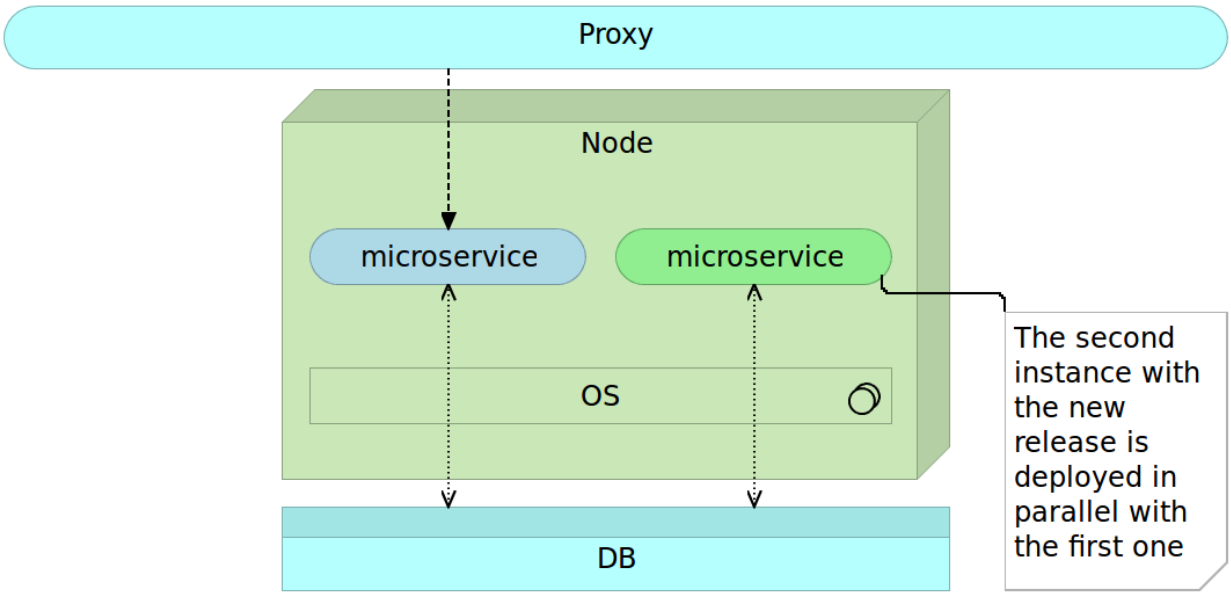

Figure 3-11: New release of the immutable microservice deployed alongside the old release

When that microservice release is properly tested we change the proxy route.

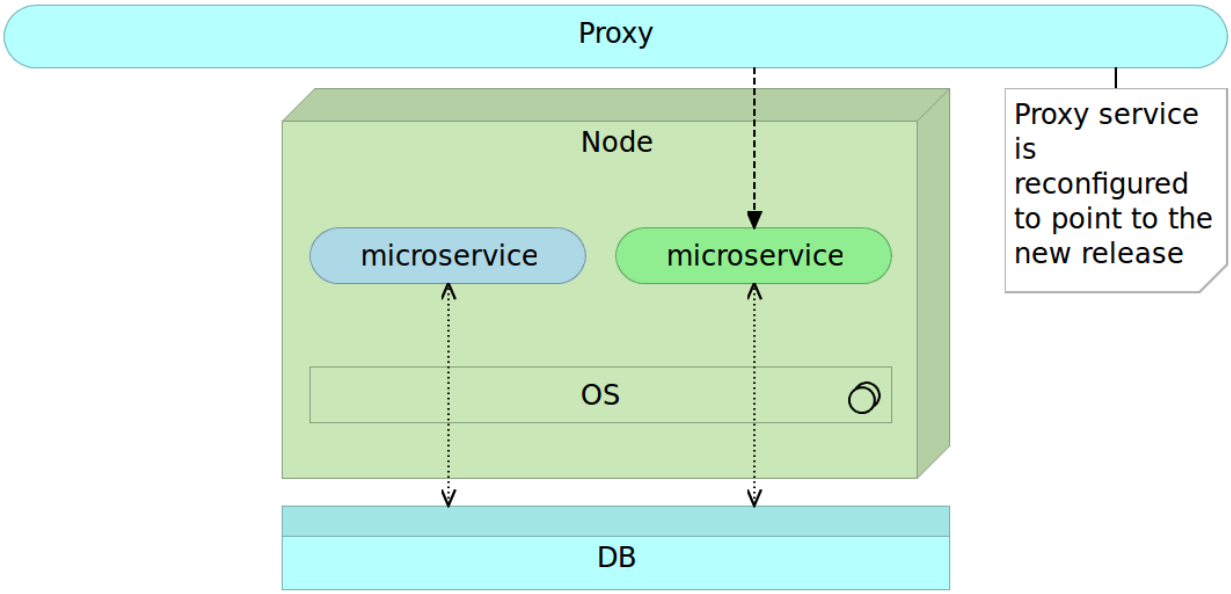

Figure 3-12: Poxy is re-configured to point to the new release

Finally, we remove the older version of the microservice.

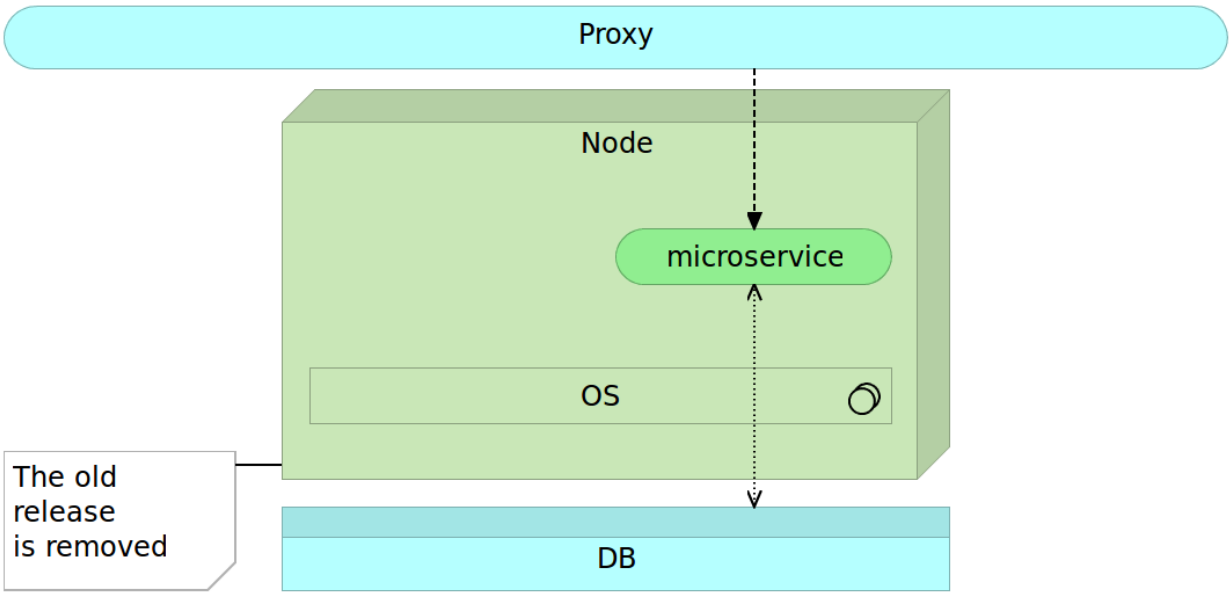

Figure 3-13: The old release is removed

The only significant difference is that due to the size of microservices, we often do not need a separate server to deploy the new release in parallel with the old one. Now we can truly deploy often automatically, be fast with zero-downtime and rollback in case something goes wrong.

Technologically, this architecture might pose particular problems that will be the subject of the next chapters. For now, let's just say that those problems are easy to solve with the tools and processes we have at our disposal.

Given our requirements that are poor at best and the advantages microservices bring over monoliths, the choice is clear. We will be building our application using immutable microservices approach. That decision calls for a discussion about the best practices we should follow.

Microservices Best Practices

Most of the following best practices can be applied to services oriented architecture in general. However, with microservices, they become even more significant or beneficial. Following is a very brief description that will be extended later on throughout the book when the time comes to apply them.

Containers

Dealing with many microservices can quickly become a very complex endeavor. Each can be written in a different programming language, can require a different (hopefully light) application server or can use a different set of libraries. If each service is packed as a container, most of those problems will

go away. All we have to do is run the container with, for example, Docker and trust that everything needed is inside it.

Containers are self-sufficient bundles that contain everything we need (with the exception of the kernel), run in an isolated process and are immutable. Being self-sufficient means that a container commonly has the following components.

- Runtime libraries (JDK, Python, or any other library required for the application to run)
- Application server (Tomcat, nginx, and so on)
- Database (preferably lightweight)
- Artifact (JAR, WAR, static files, and so on)

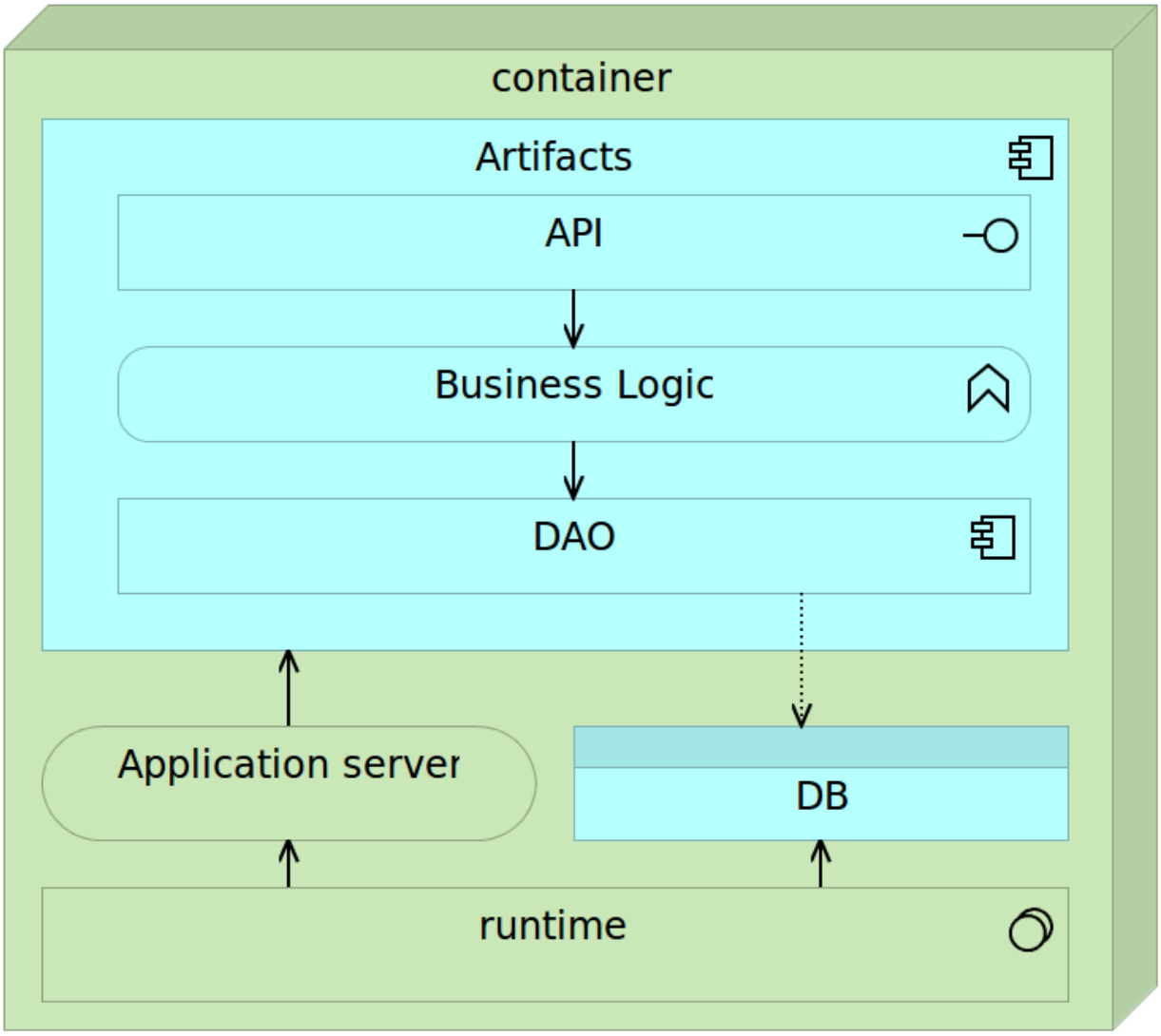

Figure 3-14: Self-sufficient microservice inside a container

Fully self-sufficient containers are the easiest way to deploy services but pose a few problems with scaling. If we'd like to scale such a container on multiple nodes in a cluster, we'd need to make sure that databases embedded into those containers are synchronized or that their data volumes are located on a shared drive. The first option often introduces unnecessary complexity while shared volumes might have a negative impact on performance. Alternative is to make containers almost self-sufficient by externalizing database into a separate container. In such a setting there would be two different containers per each service. One for the application and the other for the database. They would be linked (preferably through a proxy service). While such a combination slightly increases deployment complexity, it provides greater freedom when scaling. We can deploy multiple instances of the application container or several instances of the database depending performance testing results or increase in traffic. Finally, nothing prevents us to scale both if such a need arises.

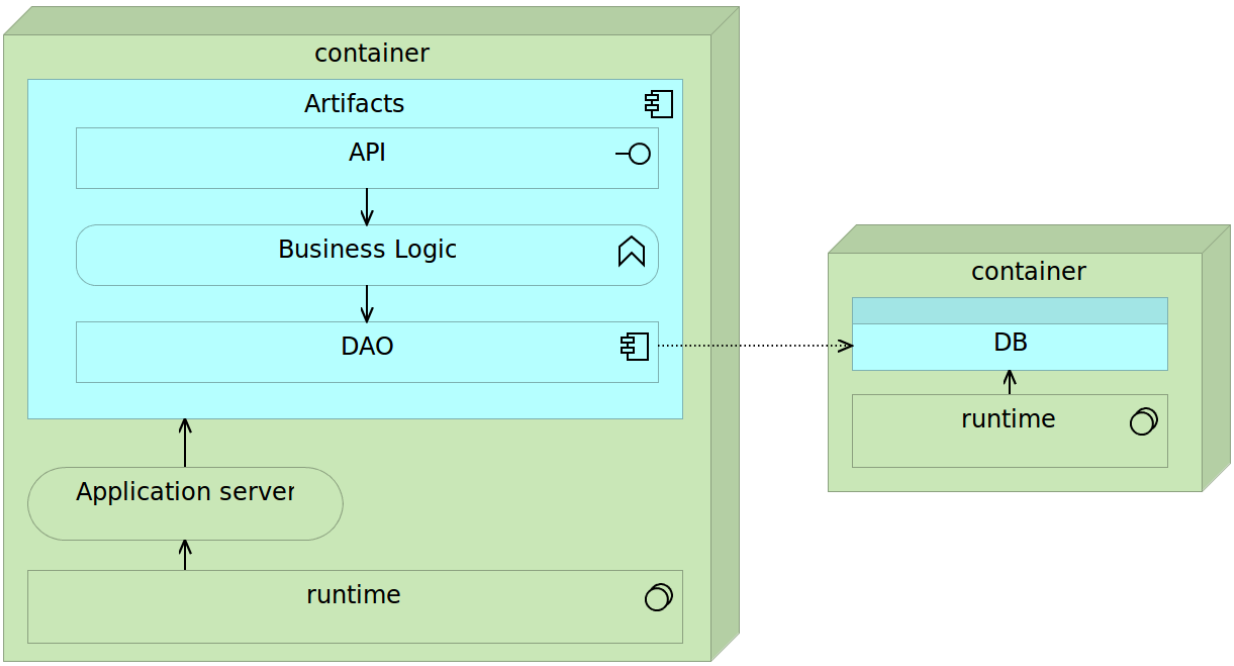

Figure 3-15: Microservice inside a container with the separate database

Being self-sufficient and immutable allows us to move containers across different environments (development, testing, production, and so on) and always expect the same results. Those same characteristics combined with microservices approach of building small applications allows us to deploy and scale containers with very little effort and much lower risk than other methods would allow us.

However, there is a third commonly used combination when dealing with legacy systems. Even though we might decide to gradually move from monolithic applications towards microservices, databases tend to be the last parts of the system to be approved for refactoring. While this is far from the optimal way to perform the transition, the reality, especially in big enterprises is that data is the most valuable asset. Rewriting an application poses much lower risk than the one we'd be facing if we decide to restructure data. It's often understandable that management is very skeptical

of such proposals. In such a case we might opt for a shared database (probably without containers). While such a decision would be partly against what we're trying to accomplish with microservices, the pattern that works best is to share the database but make sure that each schema or a group of tables is exclusively accessed by a single service. The other services that would require that data would need to go through the API of the service assigned to it. While in such a combination we do not accomplish clear separation (after all, there is no clearer more apparent than physical), we can at least control who accesses the data subset and have a clear relation between them and the data. Actually, that is very similar to what is commonly the idea behind horizontal layers. In practice, as the monolithic application grows (and with it the number of layers) this approach tends to get abused and ignored. Vertical separation (even if a database is shared), helps us keep much clearer bounded context each service is in charge of.

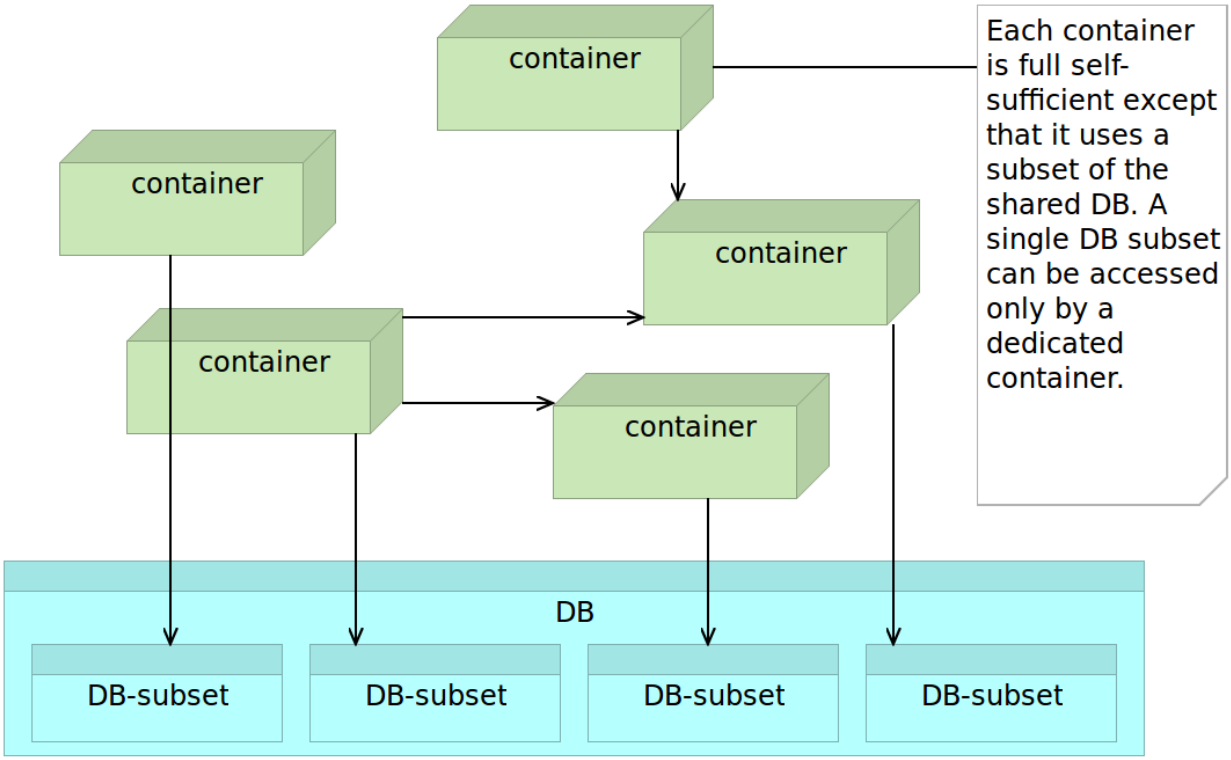

Figure 3-16: Microservices inside containers accessing the shared database

Proxy Microservices or API Gateway

Big enterprise front-ends might need to invoke tens or even hundreds of HTTP requests (as is the case with Amazon.com). Requests often take more time to be invoked than to receive response data. Proxy microservices might help in that case. Their goal is to invoke different microservices and return an aggregated service. They should not contain any logic but only group several responses together and respond with aggregated data to the consumer.

Reverse Proxy

Never expose microservice API directly. If there isn't some orchestration, the dependency between the consumer and the microservices becomes so big that it might remove freedom that microservices are supposed to give us. Lightweight servers like *nginx*, *Apache Tomcat*, and *HAProxy* are excellent at performing reverse proxy tasks and can easily be employed with very little overhead.

Minimalist Approach

Microservices should contain only packages, libraries, and frameworks that they truly need. The smaller they are, the better. That is quite in contrast to the approach used with monolithic applications. While previously we might have used JEE servers like JBoss that packed all the tools that we might or might not need, microservices work best with much more minimalistic solutions. Having hundreds of microservices with each of them having a full JBoss server becomes overkill. *Apache Tomcat*, for example, is a much better option. I tend to go for even smaller solutions with, for instance, *Spray* as a very lightweight RESTful API server. Don't pack what you don't need.

The same approach should be applied to OS level as well. If we're deploying microservices as *Docker* containers, *CoreOS* might be a better solution than, for example, *Red Hat* or *Ubuntu*. It's free from things we do not need allowing us to obtain better utilization of resources. However, as we'll see later, choosing OS is not always that simple.

Configuration Management

As the number of microservices grows, the need for *Configuration Management (CM)* increases. Deploying many microservices without tools like *Puppet*, *Chef* or *Ansible* (just to name few) quickly becomes a nightmare. Actually, not using CM tools for any but simplest solutions is a waste, with or without microservices.

Cross-Functional Teams

While no rule dictates what kinds of teams are utilized, microservices are done best when the team working on one is multifunctional. A single team should be responsible for it from the start (design) until the finish (deployment and maintenance). They are too small to be handled from one team to another (architecture/design, development, testing, deployment and maintenance teams). Preference is to have a team that is in charge of the full lifecycle of a microservice. In many cases, one team might be in charge of multiple microservices, but multiple teams should not be in charge of one.

API Versioning

Versioning should be applied to any API, and this holds true for microservices as well. If some change breaks the API format, it should be released as a separate version. In the case of public APIs as well as those used by other internal services, we cannot be sure who is using them and, therefore, must maintain backward compatibility or, at least, give consumers enough time to adapt.

Final Thoughts

Microservices as a concept existed for a long time. Take a look at the following example:

```
1 ps aux | grep jav[a] | awk '{print $2}' | xargs kill
```

The command listed above is an example of the usage of *pipes* in Unix/Linux. It consists of four programs. Each of them is expecting an input (*stdin*) and/or an output (*stdout*). Each of them is highly specialized and performs one or very few functions. While simple by themselves, when combined those programs are capable performing some very complex operations. Same holds true for most of the programs found in today's Unix/Linux distributions. In this particular case, we're running `ps aux` that retrieves the list of all running processes and passing the output to the next in line. That output is used by `grep jav[a]` to limit the results to only Java processes. Again, the output is passed to whoever needs it. In this particular example, next in line is `awk '{print $2}'` that does, even more, filtering and returns only the second column that happens to be the process ID. Finally, `xargs kill` takes the output of *awk* as input and kills all processes that match IDs we retrieved previously.

Those not used to Unix/Linux might think that the command we just examined is an overkill. However, after a bit of practice, those working with Linux commands find this approach very flexible and useful. Instead of having "big" programs that need to contemplate all possible use cases, we have a lot of small programs that can be combined to fulfill almost any task we might need. It is a power born out of utmost simplicity. Each program is small and created to achieve a very specific objective. More importantly, they all accept clearly defined input and produce well-documented output.

Unix is, as far as I know, the oldest example of microservices still in use. A lot of small, specific, easy to reason with services with well-defined interfaces.

Even though microservices exist for a long time, it is not a chance that they become popular only recently. Many things needed to mature and be available for microservices to be useful to all but selected few. Some of the concepts that made microservices widely used are *domain-driven design*, *continuous delivery*, *containers*, *small autonomous teams*, *scalable systems*, and so on. Only when all those are combined into a single framework microservices start to shine truly.

Microservices are used to create complex systems composed of small and autonomous services that exchange data through their APIs and limit their scope to a very specific bounded context. From a certain point of view, microservices are what object-oriented programming was initially designed to be. When you read thoughts of some of the leaders of our industry and, especially, object-oriented programming, their descriptions of best practices when absorbed for their logic and not the way authors implemented them initially, are the reminiscence of what microservices are today. The following quotes correctly describe some of the aspects of microservices.

The big idea is 'messaging'. The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.

– Alan Kay

> Gather together those things that change for the same reason, and separate those things that change for different reasons - Robert C. Martin

When implementing microservices, we tend to organize them to do only one thing or perform only one function. This allows us to pick the best tools for each of the jobs. For example, we can code them in a language that best suits the objective. Microservices are truly loosely coupled due to their physical separation and provide a great level of independence between different teams as long as APIs are clearly defined in advance. On top of that, with microservices, we have much faster and easier testing and continuous delivery or deployment due to their decentralized nature. When concepts we discussed are combined with the emergence of new tools, especially *Docker*, we can see microservices in a new light and remove part of the problems their development and deployment was creating earlier.

Still, do not take bits of advice from this book as something that should be applied to all cases. Microservices are not an answer to all our problems. Nothing is. They are not the way all applications should be created and no single solution fits all cases. With microservices, we are trying to solve very specific problems and not to change the way all applications are designed.

Armed with the decision to develop our application around microservices, it is time to do something practical. There is no coding without development environment so that will be our first goal. We'll create a development environment for our "fancy" books store service.

We had enough theory and the time is ripe to put this book in front of a computer. From now on, most of the book will be a hands-on experience.