



A PRACTICAL GUIDE TO

CONTINUOUS DELIVERY

EBERHARD WOLFF


About This E-Book

EPUB is an open, industry-standard format for e-books. However, support for EPUB and its many features varies across reading devices and applications. Use your device or app settings to customize the presentation to your liking. Settings that you can customize often include font, font size, single or double column, landscape or portrait mode, and figures that you can click or tap to enlarge. For additional information about the settings and features on your reading device or app, visit the device manufacturer's Web site.

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the e-book in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a “Click here to view code image” link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

A Practical Guide to Continuous Delivery

Eberhard Wolff

 **Addison-Wesley**

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

*To my family and friends for their support.
And to the computing community for all the fun it
has provided to me.*

Preface

P.1 Overview of Continuous Delivery and the Book

Continuous Delivery makes it possible to bring software much faster and with substantially higher reliability into production than before. The basis for these improvements is a Continuous Delivery pipeline that automates the software rollout to a large degree and thus represents a reproducible, low-risk process for rolling out new releases.

Where does the term Continuous Delivery originate from?

The Agile Manifesto (<http://agilemanifesto.org>) defines as its most important objective:

“Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.”

Therefore, Continuous Delivery is a technique from the Agile field.

This book explains how such a pipeline can be built in practice and which technologies can be employed for building it. The focus is not only on compiling and installing the software, but also on the different tests needed to ensure high software quality.

In addition, the book demonstrates how Continuous Delivery affects the interplay between development and operations in the context of DevOps. Moreover, the book describes the effects of Continuous Delivery on software architecture. Further, it discusses not only the theory behind Continuous Delivery, but also introduces a possible technology stack that covers build, continuous integration, load tests, acceptance tests, and monitoring. For the internal components of the technology stack there is always an example project provided that helps the reader to gain practical experience. While the book offers a first introduction into the technology stack, it also highlights ways to gain more comprehensive knowledge on the different topics. By providing suggestions for experiments and for how to try things out on their own it invites the readers to broaden their practical knowledge. In this way the readers receive guidance about how to continue studying the presented topics and how to develop hands-on experience with them. For instance, the example projects can serve as bases for individual

experiments or even for the buildup of an individual Continuous Delivery pipeline.

The website <http://continuous-delivery-book.com> contains further information, errata, and links to the examples can be found.

P.2 Why Continuous Delivery?

Why should Continuous Delivery be used at all? A small story will serve to answer this question—whether the story is true is another question.

P.2.1 A Small Story

The marketing department of an enterprise—let us call it Big Money Online Commerce Inc.—decided to revise the registration process of its e-commerce website. This was meant to attract more customers to increase the sales volume. So a team of developers got down to work. A short while later the team had finished the task.

First of all, the modifications had to be tested. For testing the team of Big Money Online Commerce Inc. had built a test environment in a laborious process. In this test environment the software had to be manually tested. Unfortunately, the testing indeed revealed errors. However, meanwhile the developers were already working on the next project and first had to refamiliarize themselves with the old project before being able to fix the errors. In addition, because of the manual testing some “errors” turned out to be caused by the fact that the testers had not tested correctly, and other “errors” were just not reproducible for some reason.

Next the code had to be brought into production. The required process was very elaborate since the e-commerce website of Big Money Online Commerce Inc. had grown over the years and had therefore become very complex. Delivery of a single feature did not justify the effort associated with a deployment. Therefore, a deployment was only scheduled once per month. After all, the modification of the registration process could be brought into production together with the other changes made during the last month. Accordingly, one night was reserved for the deployment of all the modifications. Unfortunately, an error occurred during the rollout. The team started to analyze the problem. However, this turned out to be very difficult, so the system was not available the next morning. By then the

developers were tired out and under a lot of pressure—each minute that the system was down cost money. Still, there was no way back to the old version since some modifications in the deployment could not be easily undone. Only over the course of the day, after a comprehensive error analysis, was a task force able to eliminate the problem so that the website became available again. In the end the problem had been a configuration change that had been performed in the test environment, but forgotten during the deployment.

For the moment everything seemed fine; however, there was another error, which was initially overlooked. This error should have been discovered by the manual tests. The test that would have been able to spot the error had even been run successfully. However, during the testing phase a number of errors were repaired, and this specific test was only performed before the fixes were introduced. But the error in question only arose due to one of the fixes, and since the manual test was not repeated after the fixing, this error made it into production.

Therefore, the next day it was discovered by chance that the registration for the website of Big Money Online Commerce Inc. did not work at all anymore. Nobody had noticed that before. Only when the first potential customer called the hotline to complain about the problem was it finally recognized. Unfortunately, at this point it was impossible to say how many registrations had been lost due to the failure—the necessary statistics about the website were lacking. How fast the optimized registration process would be able to compensate for these lost registrations remained doubtful. It was quite possible that the modifications had not resulted in more registrations overall, as originally planned, but rather in fewer. Besides all this, the new release was substantially slower—a circumstance nobody had anticipated either.

Therefore, Big Money Online Commerce Inc. started to implement the next round of optimizations and features in order to rollout a new update of the website in the next month. What might be the chances that this deployment would work any better?

P.2.2 Continuous Delivery Helps

Continuous Delivery solves such problems by different measures:

- Deployment takes place more frequently—up to several times per day. This decreases the time until a new feature can be used.
- Frequent deployments also result in faster feedback to new features and code modifications. The developers do not have to think back to remember what they implemented a month ago.
- To be able to deploy faster the buildup of test environments and the tests themselves have to be largely automated since otherwise the necessary effort is too high.
- Automation results in reproducibility: Once the test environment has successfully been built, the same automation can be used to build up the production—with the same configuration, in fact. Therefore problems caused by a misconfiguration of the production environment do not occur.
- In addition, automatization leads to more flexibility. Test environments can be built on demand. For example, in the case of a redesign of the user interface there can be a separate test environment for marketing for a limited time. Further, in the case of comprehensive load tests, additional environments can be generated in order to have an environment that is close to production, and can be destroyed again after testing so that no permanent investments into hardware are necessary (for instance, when a cloud is used).
- Automated tests make it easier to reproduce errors. Since the identical steps are carried out during each test, there are no errors associated with running the test.
- When tests are automated, they can be run more frequently without extra effort. Therefore, any fix undergoes the entire testing process so that the error in the registration process of the Big Money Online Commerce Inc. would not have been detected only in production.
- The risk associated with a new release is further reduced by setting up the deployment to production in such a manner that there is an easy way back to the old version if need arises. This prevents the production failure described in the story.
- Finally, applications are supposed to have domain-based monitoring so that processes like registration cannot break down without anybody noticing.

In summary, Continuous Delivery provides the business with faster availability of new features and with more reliable IT systems. The increased reliability is also an asset for the IT employees themselves since it is no fun to roll out new releases and fix emerging errors under high pressure at night or on weekends. Additionally, it is certainly better for IT and the enterprise when errors are detected during tests and not only in production.

There are numerous technologies and techniques to implement Continuous Delivery. Continuous Delivery has diverse effects, even on the architecture of an application. These are the topics this book deals with. The objective is to create a fast and reliable process to bring software into production.

P.3 For Whom Is the Book Meant?

The book is meant for managers, architects, developers, and administrators who want to introduce Continuous Delivery as a method and/or DevOps as an organization:

- In the theoretical part of the book, managers get to know the processes behind Continuous Delivery as well as its requirements and benefits for their enterprise. In addition, they learn to evaluate the technical consequences of Continuous Delivery.
- Developers and administrators are provided with a comprehensive introduction into the technical aspects and can acquire the necessary skills to implement Continuous Delivery and to build up a Continuous Delivery pipeline.
- Architects can, in addition to the technical aspects, also learn about the effects of Continuous Delivery on the software architecture—see [Chapter 11](#).

The book introduces different technologies for the implementation of Continuous Delivery. A Java project serves as an example. For some technological areas—for example, for the implementation of acceptance tests—other technologies are established for other programming languages. The book mentions alternatives in these contexts but focuses on Java. Technologies for the automated allocation of infrastructures are independent of the programming language used. The book is especially well suited for readers who are active in the Java field—for other technologies the approaches have to be transferred to some extent by the readers themselves.

P.4 Overview of the Chapters

The book comprises three parts. The first part establishes the fundamentals for understanding Continuous Delivery:

- [Chapter 1](#), “[Continuous Delivery: What and How?](#)” introduces the term Continuous Delivery and explains which problems Continuous

Delivery solves and how. A first introduction into the Continuous Delivery pipeline is given.

- Continuous Delivery requires the automated deployment of infrastructure—software has to be installed on servers. [Chapter 2](#), “[Providing Infrastructure](#),” introduces some approaches for this. Chef can be used to automate installations. Vagrant can be employed to set up test environments on developer machines. Docker is not only a very efficient solution for virtualization, but can also serve for the automated installation of software. At the end of the first part an overview of the use of PaaS (Platform as a Service) cloud solutions for Continuous Delivery is provided.

The second part comprises chapters that describe the different components of a Continuous Delivery pipeline in detail. After a conceptual introduction, concrete technologies are presented as examples that can be used to implement the discussed part of the pipeline:

- The focus of [Chapter 3](#), “[Build Automation and Continuous Integration](#),” by Bastian Spanneberg, is on what happens during the commit of a new software version. Build tools like Gradle or Maven are introduced, an overview of unit tests is given, and Continuous Integration with Jenkins is discussed. These topics are followed by static code reviews with SonarQube and repositories like Nexus or Artifactory.
- [Chapter 4](#), “[Acceptance Tests](#),” introduces JBehave and Selenium as an approach for automated GUI-based acceptance tests and for textual acceptance tests.
- Performance is covered in [Chapter 5](#), “[Capacity Tests](#),” by just that—capacity tests. Gatling is used as example technology.
- Exploratory testing, which is discussed in [Chapter 6](#), “[Exploratory Testing](#),” serves to manually check new features and general problems in the applications.

These chapters deal with the start of the Continuous Delivery pipeline. These phases influence primarily software development. The following

chapters mainly introduce technologies and techniques that are useful for the areas of Continuous Delivery that are close to production:

- [Chapter 7, “Deploy—The Rollout in Production,”](#) describes approaches that are useful for minimizing risk during the rollout of software in production.
- During operations of an application various data can be collected to obtain feedback. [Chapter 8, “Operations,”](#) introduces technologies that facilitate the gathering and analysis of this data: ELK stack (Elasticsearch-Logstash-Kibana) for log file analysis and Graphite for monitoring.

For the different technologies described in these chapters there are examples provided so that the readers can try them out and experiment with them on their own computers. This allows the readers to gain hands-on experience. Thanks to infrastructure automation these examples are easy to run on their own computers.

Finally the question arises: how can Continuous Delivery be introduced and what effects does it have? This is discussed in the third part of the book:

- [Chapter 9, “Introducing Continuous Delivery into Your Enterprise,”](#) shows how Continuous Delivery can be introduced in an organization.
- [Chapter 10, “Continuous Delivery and DevOps,”](#) describes the merging of Development (Dev) and operations (Ops) into one organizational unit (DevOps).
- Continuous Delivery also has effects on the architecture of applications. The associated challenges are discussed in [Chapter 11, “Continuous Delivery, DevOps, and Software Architecture.”](#)
- The book ends with a conclusion in [Chapter 12, “Conclusion: What Are the Benefits?”](#)

P.5 Paths Through the Book

This section describes the possible reading paths through the book for the different audiences—that is, which chapters should be read in which order ([Figure P.1](#)). The introduction in [Chapter 1](#) is interesting for all readers—the

chapter clarifies the basic terms and discusses the motivation for Continuous Delivery.

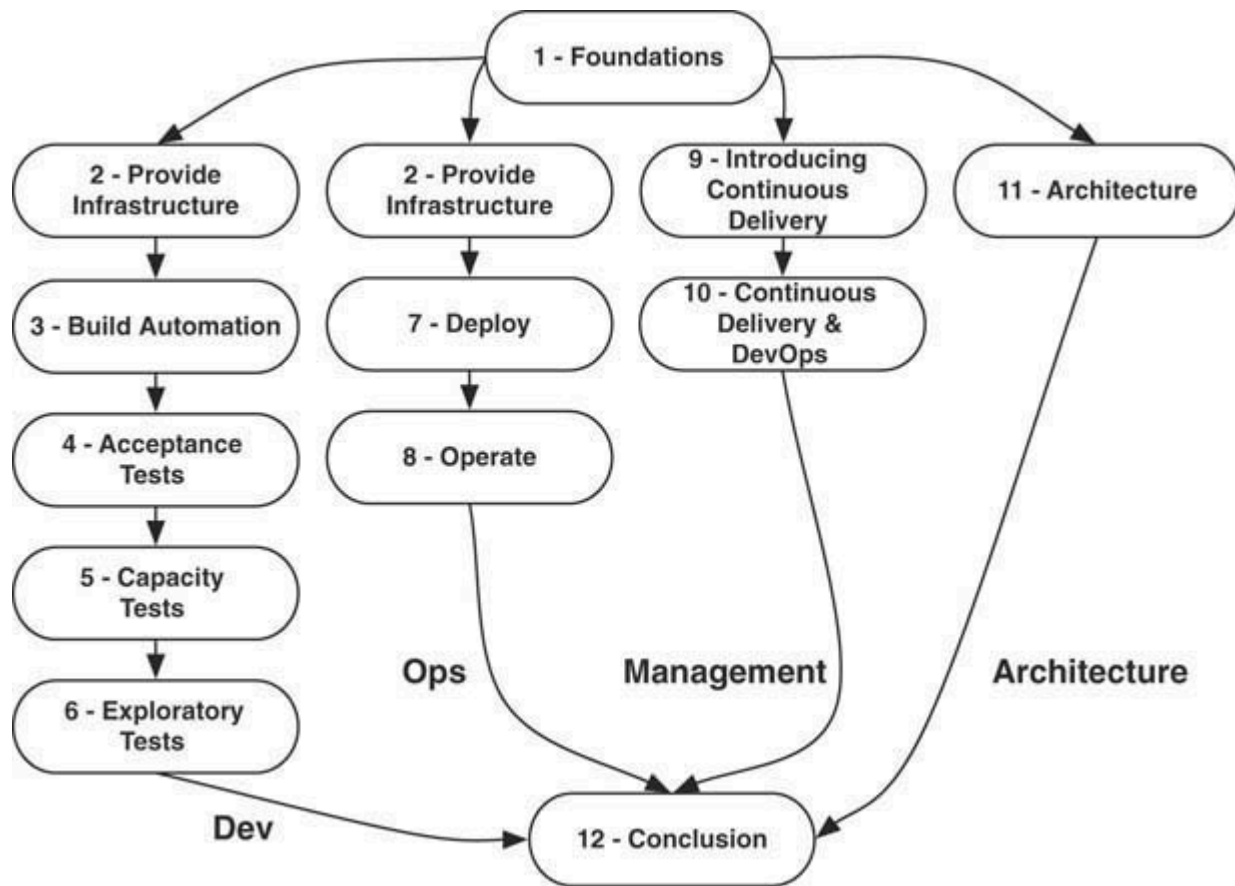


Figure P.1 *Paths through the book*

The ensuing chapters have different focuses:

- For technicians, who are primarily interested in development, the chapters dealing with commit and tests are most interesting. These chapters discuss, in addition to development, quality assurance and build, and show how these tasks are influenced by Continuous Delivery. In addition, they provide concrete code and technology examples taken from the Java field.
- Administrators and people working in operations particularly need to know about topics like deployment, allocation of infrastructure, and operations, which are likewise influenced by Continuous Delivery.
- From a management perspective information about the introduction of Continuous Delivery and the relationship between DevOps and

Continuous Delivery are most relevant. These two chapters show the effects of Continuous Delivery on the organization.

- Architects have to take a broad perspective. [Chapter 11](#) dealing with architecture is certainly the main focus for them; however, since they are usually also interested in technical details and the management perspective, they might also want to read at least selected chapters dealing with these topics.
- Finally, an extra chapter is devoted to the conclusion.

Note

Register your copy of *A Practical Guide to Continuous Delivery* at informit.com for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN 9780134691473 and click Submit. Once the process is complete, you will find any available bonus content under “Registered Products.”

Acknowledgments

I would like to thank Bastian Spanneberg for his contribution to this book. In addition, my thanks go to the reviewers, who substantially influenced this book with their insightful comments: Marcel Birkner, Lars Gentsch, Halil-Cem Gürsoy, Felix Müller, Sascha Möllering, and Alexander Papaspyrou. I also would like to say thanks for the many discussions with friends and colleagues—many thoughts and ideas are the product of such discussions.

I would also like to thank my employer, innoQ.

Finally, I would also like to say thanks to my friends, parents, and relatives, whom I often neglected while working on this book, and especially to my wife to whom I am indebted for the English translation of this book.

Of course my thanks go also to all those people who helped to create the technologies described in this book and thus made Continuous Delivery possible.

Last but not least I would like to thank the dpunkt.verlag and René Schönfeldt who supported me very professionally during the creation of the German version of this book.

Addison-Wesley gave me the opportunity to publish the English version. Chris Zahn, Chris Guzikowski, Lori Lyons, Dhayanidhi Karunanidhi, and Larry Sulky provided excellent support for that process.

About the Author

Eberhard Wolff, a Fellow at innoQ in Germany, has more than 15 years of experience as an architect and consultant working at the intersection of business and technology. He has given talks and keynote addresses at several international conferences, served on multiple conference program committees, and written more than 100 articles and books. His technological focus is on modern architectures—often involving cloud, continuous delivery, DevOps, microservices, and NoSQL.

PART I

Foundations

In this part [Chapter 1](#) provides the basics to understand Continuous Delivery. [Chapter 2](#) explains the technical foundations of Continuous Delivery: It discusses the automated deployment of infrastructure and the automated installation of software without which Continuous Delivery would be impossible.

Chapter 1. Continuous Delivery: What and How?

1.1 Introduction: What Is Continuous Delivery?

This question is not so easy to answer. The inventors of the term do not provide a real definition.¹ Martin Fowler focuses in his discussion² of Continuous Delivery on the fact that software can be brought into production at any time. This requires an automation of the processes necessary for the installation of software and feedback about software quality. Wikipedia³ on the other hand defines Continuous Delivery as an optimization and automation of the software release process.

In the end, the main objective of Continuous Delivery is to analyze and optimize the process leading up to the release of software. Exactly speaking this process is often blended out during development.

1.2 Why Software Releases are So Complicated

Software releases are a challenge—very likely every IT department has already worked during a weekend to bring a software release into production. Such events often end with bringing the software somehow into production—because from a certain point the path back to the old version is even more dangerous and difficult than the path ahead. However, the installation of the release is then often followed by a long phase in which the release has to be stabilized.

1.2.1 Continuous Integration Creates Hope

Nowadays it is the release into production that represents a challenge. Not so long ago the problems started much earlier: Individual teams worked independently on their modules, and prior to the release the different versions first had to be integrated. When the modules were put together for the first time, the system frequently did not even compile. Often it took days or even weeks until all changes were integrated and compiled successfully. Only then could the deployments commence. These problems have mostly been solved by now: All teams work on a shared version of the code that is permanently automatically integrated, compiled, and tested. This approach is called Continuous Integration. The required infrastructure for Continuous

Integration is detailed in [Chapter 3, “Build Automation and Continuous Integration.”](#) The fact that the former problems associated with this phase have been solved raises hopes that there will also be solutions for the problems arising during the other phases leading up to production.

1.2.2 Slow and Risky Processes

The processes in the later phases are often highly complex and elaborate. In addition, manual steps render them very tedious and error-prone. This is true for the release into production, but also for the preceding phases—for example, during testing. Especially during a manual process, which, to make things worse, is only performed a few times per year, errors are likely to occur. This of course contributes to the risk associated with the overall procedure.

Because of the high risk and complexity, releases are not very frequently brought into production. In the end this causes the processes to take even longer due to lack of practice. In addition, this makes it difficult to optimize the processes.

1.2.3 It’s Possible to be Fast

On the other hand, there are always possibilities to bring a release rapidly into production in an emergency—for instance, when an error has to be urgently repaired. However, in such a case all the tests and therefore all the safety nets, which are an integral part of the standard process, are omitted. This is of course a pretty high risk—there are good reasons to normally run those tests.

Therefore, the normal path into production is slow and risky—and in emergencies the path can be faster, but at the expense of even more risk.

1.3 Values of Continuous Delivery

Using the motivation and the approaches of Continuous Integration, we want to optimize the way for releases into production.

A fundamental principle of Continuous Integration is: “If it hurts, do it more often and bring the pain forward.” What sounds like masochism is in reality an approach for problem solving. Instead of avoiding problems with releases by bringing as few releases as possible into production, these processes should be performed as often and as early as possible in order to

optimize them as quickly as possible—with regards to speed and with regards to reliability. Consequently, Continuous Delivery forces the organization to change and to adopt a new way of working.

In the end this approach is not really surprising: As mentioned, every IT organization is able to rapidly bring a fix into production—and in such a scenario it is common practice to perform only a fraction of the usual tests and security checks. This is possible because the change is small and therefore represents only a small risk. Here, another approach for minimizing risk becomes apparent: Instead of trying to safeguard against failures via complex processes and rare releases, it is also possible to more frequently bring small changes into production. This approach is in essence identical to the Continuous Integration strategy: Continuous Integration means that even small software changes by the individual developers and the team are permanently integrated instead of having the teams and developers work independently for days or weeks and integrating all the accumulated changes only at the end—a strategy that frequently causes substantial problems; in some cases the problems are so large that the software cannot be compiled at all.

However, Continuous Delivery is more than just “fast and small.” Continuous Delivery is based on different values. From these values, concrete technical measures can be deduced.

1.3.1 Regularity

Regularity means to execute processes more frequently. All processes that are necessary to bring software into production should regularly be performed—and not only when a release has to be brought into production. For example, it is necessary to build test and staging environments. The test environments can be used for acceptance or technical tests. The staging environments can be used by the final customer for testing and evaluating the features of a new release. By providing these environments, the process for the generation of an environment can turn into a regular process that is not merely performed when the production environment has to be created. To generate this multitude of environments without too much effort the processes have to become largely automated. Regularity usually leads to automation. Similar rules apply to tests: It does not make sense to postpone the necessary tests until right before the release—instead they should rather be performed regularly. Also in this case this approach basically forces

automation in order to limit the necessary effort. Regularity also leads to a high degree of reliability—processes that are frequently performed can be reliably repeated and executed.

1.3.2 Traceability/Confirmability

All changes to the software that is supposed to be brought into production and to the infrastructure that is required for the release have to be traceable. It has to be possible to reconstruct each state of the software and of the infrastructure. This leads to versioning that does not only comprise the software, but also the necessary environments. Ideally, it is possible to generate each state of the software together with the environment required for the operation in the right configuration. Thereby all changes to the software and the environments can be traced. Likewise it is very easy to generate a suitable system for error analyses. And finally, changes can be documented or audited in this manner.

One possible solution for the problem is that production and staging environments are only accessible for certain persons. This is supposed to avoid “quick fixes” that are not documented and cannot be traced anymore. Besides, security requirements and data security argue against accessing production environments.

With Continuous Delivery, interventions into an environment are only possible when an installation script is changed. The changes to the scripts are traceable when they are deposited in a version control system. The developers of the scripts also do not have access to the production data so that there are also no problems with data security.

1.3.3 Regression

To minimize the risk associated with bringing software into production, the software has to be tested. Of course, the correct functioning of new features has to be assured during the testing. However, a lot of effort arises from the attempt to avoid regressions—that is, errors that are introduced by modifications in already tested software parts. This would in effect require that all tests be rerun in case of a modification since in the end a modification at one site of the system might cause an error somewhere else. This necessitates automated tests. Otherwise the required effort for the execution becomes much too high. Should an error nevertheless make its way into production, it can still be discovered by monitoring. Ideally, there is

the possibility to install as simply as possible an older version on the production system without the error (rollback) or to bring a fix quickly into production (roll forward). In the end the idea is to have a kind of early warning system that takes measures throughout different phases of the project, like test and production, to discover and solve regressions.

1.4 Benefits of Continuous Delivery

Continuous Delivery offers numerous benefits. Depending on the scenario the different advantages can be of varying importance—consequently this will influence how Continuous Delivery is implemented.

1.4.1 Continuous Delivery for Time to Market

Continuous Delivery decreases the time required for bringing changes into production. This generates a substantial benefit on the business end: It becomes much easier to react to changes of the market.

However, the advantages extend beyond a faster time to market: Modern approaches like Lean Startup⁴ advocate a strategy that benefits even more from the increased speed. The focus of Lean Startup is to position products on the market and to evaluate their chances at the market while investing as little effort as possible in doing so. Just like with scientific experiments, it is defined beforehand how the success of a product on the market can be measured. Then the experiment is performed, and afterwards the success or failure is measured.

1.4.2 One Example

Let us look at a concrete example. In a web shop a new feature is supposed to be created: Orders can be delivered on a defined day. As a first experiment the new feature can be advertised. Here the number of clicks on a link within the advertisement can be used as an indication for the success of this experiment. At this point no software has been developed yet—that is, the feature is not yet implemented. If the experiment did not lead to a promising result, the feature does not appear to be beneficial and other features can be prioritized instead—without much effort having been invested.

1.4.3 Implementing a Feature and Bringing It into Production

If the experiment was successful, the feature will be implemented and brought into production. Even this step can be conducted like an experiment: Metrics can help to control the success of the feature. For example, the number of orders with a fixed delivery date can be measured.

1.4.4 On to the Next Feature

The analysis of the metrics reveals that the number of orders is high enough—interestingly most orders are not sent directly to the customer, but to a third person. Additional measurements show that the ordered items are obviously birthday presents. Based on this information the feature can be expanded—for example with a birthday calendar and recommendations for suitable presents. This requires of course that such additional features are designed, implemented, brought into production and finally evaluated with regards to their success. Alternatively, there might also be options to evaluate the potential market success of these features without any implementation—via advertisements, customer interviews, surveys, or other approaches.

1.4.5 Continuous Delivery Generates Competitive Advantages

Continuous Delivery makes it possible to more rapidly bring required software changes into production. This allows enterprises to more quickly test different ideas and to develop their business model further. This creates a competitive advantage: Since more ideas can be evaluated, it is easier to filter out the right ones—and this is not based on subjective estimations of market chances, but on the basis of objectively measured data ([Figure 1.1](#)).

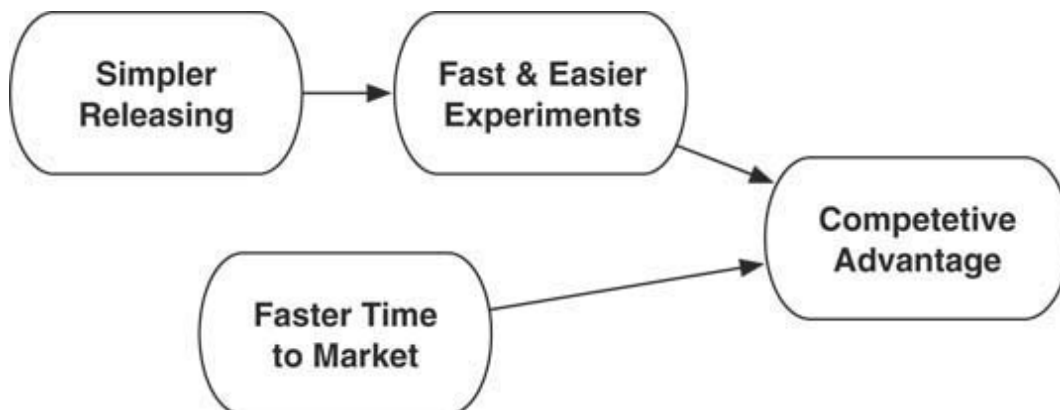


Figure 1.1 *Reasons for Continuous Delivery in a startup*

1.4.6 Without Continuous Delivery

Without Continuous Delivery the feature for the fixed delivery dates would have been planned and brought into production during the next release—this would likely have taken a number of months. Before the release, marketing would hardly have dared to advertise the feature since the long time up to the next release would render such advertisements futile. If the feature had not proven successful in the end, there would have been high costs caused by its implementation without creating any benefit. Evaluating the success of a new feature would certainly also be possible in the classical approach; however, the reaction would be drastically slower. Further developments such as the features supporting the buying of birthday presents would reach the market much later since they would require that the software be brought into production again and the time-consuming release process be run through a second time. Besides, it remains doubtful whether the success of the feature would have been analyzed in enough detail to recognize the potential for additional features supporting the shopping for birthday presents.

1.4.7 Continuous Delivery and Lean Startup

Therefore, optimization cycles can be passed through much faster thanks to Continuous Delivery because each feature can be brought into production practically at any time. This makes approaches like Lean Startup possible. This influences how the business end is working: It has to more rapidly define new features and does not have to focus on long range planning anymore, but can immediately react to the outcome of the current experiments. This is especially easy in startups, but such structures can also be built in classical organizations. The Lean Startup approach has, unfortunately, a misleading name: It is an approach where new products are positioned on the market via a series of experiments, and this approach can of course also be implemented in classical enterprises, not only in startups. It can also be used when products have to be delivered classically—for instance, on media such as CDs, with other complex installation procedures, or as part of another product such as a machine. In such a case the installation of the software has to be simplified or ideally automated. Besides a range of customers has to be identified who would like to test new software versions and be willing to provide feedback on them—that is, classical beta testers or power users.

1.4.8 Effects on the Development Process

Continuous Delivery influences the software development process: When individual features are supposed to be brought into production, the process has to support this. Some processes use iterations of one or several weeks' length. At the end of each iteration a new release with several features is brought into production. This is not an ideal approach for Continuous Delivery because in this way individual features cannot pass through the pipeline on their own. This also poses obstacles for the Lean Startup approach: When several features are rolled out at the same time, it is not obvious which change influences the measured values. Let us assume that the option for delivery on a fixed date is introduced in parallel with a change of the shipment costs—it will not be possible to distinguish which of the two changes had a greater influence on the higher number of sold items.

Therefore, processes like Scrum, XP (Extreme Programming), and of course the waterfall are impedimentary since they always bring several features together into production. Kanban,⁵ on the other hand, focuses on bringing a single feature through the different phases into production. This fits ideally with Continuous Delivery. Of course, the other processes can also be modified in ways that allow them to support the delivery of individual features. However, in such a case the processes have been adapted and are not implemented according to the textbook anymore. Another possibility is to initially deactivate the additional features in order to bring several features together in one release into production, but still be able to measure their effects separately.

In the end this approach also means that teams include multiple different roles. In addition to development and operation of the features, business roles such as marketing are conceivable. Thanks to the decreased organizational hurdles, the feedback from the business end can be translated into experiments even faster.

Try and Experiment

- Gather information about Lean Startup and Kanban. Where did Kanban come from originally?

Choose a project you know or a feature in a project:

- What could a minimal product look like? The minimal product should give an idea about the market chances of the planned complete product.
- Is it also possible to evaluate the product without software? Is it, for instance, possible to advertise it? Are interviews of potential users an option?
- How can the success of the feature be measured? Is there, for instance, an influence on sales, a number of clicks, or another value that could be measured?
- How much time in advance do marketing and sales typically have for planning a product or feature? How does that fit to the Lean Startup idea?

1.4.9 Continuous Delivery to Minimize Risk

The use of Continuous Delivery as described in the last section goes together with a certain business model. However, for classical enterprises the business often depends on long-range planning. In such a case an approach like Lean Startup cannot be implemented. In addition, there are many enterprises for which time to market is not a decisive factor. Not all markets are very competitive in this regard. This can of course change when such companies are suddenly confronted with competitors that are able to enter the market with a Lean Startup model.

In many scenarios time to market cannot motivate the introduction of Continuous Delivery. Still the techniques can be useful since Continuous Delivery offers additional benefits:

- Manual release processes require a lot of effort. It is no rare event that entire IT departments are blocked for a whole weekend for a release. And after a release there is frequently still extensive follow-up work to do.
- Also the risk is high: The software rollout depends on many manual modifications, which easily leads to mistakes. If the errors are not

discovered and fixed in time, this can have far-reaching consequences for the enterprise.

The sufferers are found in the IT departments: Developers and system administrators who work through weekends and nights to bring releases into production and to fix errors. In addition to working long hours they are subjected to high stress because of the high risk. And the risks should not be underestimated: Knight Capital, for instance, lost \$440M because of a failed software rollout.⁶ As a consequence the company went into insolvency. A number of questions⁷ arise from such scenarios—in particular why the problem occurred, why it wasn't noticed in a timely manner, and ultimately how such events can be prevented in other environments.

Continuous Delivery can be a solution for such situations: Fundamental aspects of Continuous Delivery are the higher reliability and the quality of the release process. This allows developers and system administrators to sleep calmly in the true sense of the word. Different factors are relevant for this:

- Due to the higher level of automation of the release processes, the results become easier to reproduce. Thus, when the software has been deployed and tested in a test or staging environment, the exact same result will be obtained in production because the environment is completely identical. This allows largely eliminating sources of error, and consequently the risk decreases.
- In addition, testing software becomes much easier since the tests are largely automated. This increases the quality further as the tests can be performed more frequently.
- When there are more frequent deployments, the risk decreases likewise since fewer changes are brought into production per deployment. Fewer changes translate into a smaller risk that an error has crept in.

In a way, the situation is paradoxical: The classical IT tries to bring releases as seldom as possible into production since they are associated with a high risk. During each release an error with potentially disastrous consequences can creep in. Fewer releases should therefore result in fewer problems.

is the main motivation for Continuous Delivery, it is essential that the pipeline include production.

Try and Experiment

Look at your current project:

- Where do problems typically arise during installation?
- Could these problems be solved by automation?
- Where should the current approaches be simplified in order to facilitate automation and optimization? Evaluate the required effort and the expected benefit.
- How are production systems and test systems currently built? By the same team? Would it be conceivable to apply automation to both areas or only to one of the two?
- For which systems would automation be useful? How often are the systems built?

1.4.10 Faster Feedback and Lean

When a developer modifies the code, she receives feedback from her own tests, integration tests, performance tests, and finally from production. If changes are brought into production only once per quarter, several months can pass between the code modifications and the feedback from production. The same can hold true for acceptance or performance tests. If an error occurs then, the developer has to think back to what it was she had implemented months ago and what the problem might be.

With Continuous Delivery the feedback cycles become much faster: Every time the code passes through the pipeline the developer and his/her entire team receive feedback. Automated acceptance and capacity tests can be run after each change. This enables the developer and the development team to recognize and fix errors much more rapidly. The speed of feedback can be further increased by preferring fast tests, such as unit tests, and by first testing broadly and only afterwards testing deeply. This ensures from the start that all features function at least for easy cases—the so-called “happy path.” This makes spotting basic errors easier and faster. In addition, tests

that are known from experience to fail more often should be executed at the start.

Continuous Delivery is also in line with Lean thinking. Lean regards everything that is not paid for by the customer as waste. Any change to the code is waste until it is brought into production since only then will the customer be willing to pay for the modifications. Besides, Continuous Delivery implements shorter cycle times for faster feedback—another Lean concept.

Try and Experiment

Have a look at your current project:

- How much time passes between a code change and
 - feedback from a Continuous Integration server?
 - feedback from an acceptance test?
 - feedback from a performance/capacity test?
 - bringing it into production?

1.5 Generations and Structure of a Continuous Delivery Pipeline

As already mentioned, Continuous Delivery extends the approach of Continuous Integration to additional phases. [Figure 1.3](#) offers an overview of the phases.

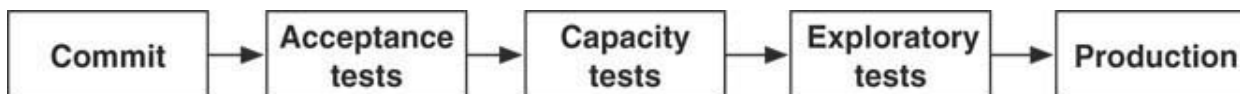


Figure 1.3 *Phases of a Continuous Delivery pipeline*

This section introduces the structure of a Continuous Delivery environment. It is oriented along Humble et al. (see footnote 1) and consists of the following phases:

- The commit phase comprises the activities that are typically covered by a Continuous Integration infrastructure such as the build process, unit

tests, and static code analysis. [Chapter 3](#) discusses this part of the pipeline in detail.

- The next step is acceptance testing, which is the focus of [Chapter 4](#), “[Acceptance Tests](#).” Strictly speaking, the topic is automated tests: Either the interactions with the GUI are automated to test the system or the requirements are described in natural language in a manner that allows them to be used as automated tests. From this phase on, if not before it is necessary to generate environments on which the applications can run. Therefore, [Chapter 2](#), “[Providing Infrastructure](#),” deals with the question of how such environments can be generated.
- Capacity tests ([Chapter 5](#), “[Capacity Tests](#)”) ensure that the software can cope with the expected load. For this purpose an automated test should be used that unambiguously indicates whether the software is sufficiently fast. The relevant point is not only performance, but also scalability. Therefore, the test can also take place in an environment that does not correspond to the production environment. However, the environment has to be able to deliver reliable results about the behavior in production. Depending on the concrete use case other non-functional requirements, such as security, can also be tested in an automated fashion.
- During explorative tests ([Chapter 6](#), “[Exploratory Testing](#)”) the application is not examined based on a strict test plan. Instead, domain experts test the application with a focus on new features and unanticipated behaviors. Thus, even in Continuous Delivery not all tests have to be automated. In fact, by having a large number of automated tests, capacity is freed for explorative testing since routine tests do not have to be manually worked off anymore.
- The deployment into production ([Chapter 7](#), “[Deploy—The Rollout in Production](#)”) merely comprises the installation of the application in another environment and is therefore relatively low risk. There are different approaches to further minimize the risks associated with the introduction into production.
- During operation of the application, challenges arise—especially in the areas of monitoring and surveillance of log files. These challenges are discussed in [Chapter 8](#), “[Operations](#).”

In principle, several releases can be processed in the pipeline in parallel. Of course this requires that the pipeline support multiple releases in parallel—if the tests are running in fixed environments, this is not possible since the environment will be occupied by a test so that a parallel test of a second release cannot run at the same time.

However, it is very rare that releases are processed in parallel by Continuous Delivery. A project should have exactly one state in the version administration, which is then promoted through the pipeline. At the most it might happen that modifications to the software occur with such a speed that a new release is already sent into the pipeline before the previous release has left the pipeline. Maybe there are exceptions for hotfixes—but one objective of Continuous Delivery is just to treat all releases equally.

1.5.1 The Example

Throughout the book an example application is used—a customer registration inspired by the example of Big Money Online Commerce Inc. (see [section P.2](#)). This example is intentionally kept very simple concerning the domain logic. Essentially the first name, name, and email address of a customer are registered. The registrations are validated. The email address has to be syntactically correct, and there is only one registration allowed per email address. In addition, a registration can be searched based on the email address, and can be deleted.

Since the application is not very complex, it is relatively easy to understand so that the reader can concentrate on the different aspects of Continuous Delivery that are illustrated by the example application.

Technically the application is implemented with Java and the framework Spring Boot. This makes it possible to start the application, including web interface, without installing a web or application server. Thus the testing becomes easier since no infrastructure has to be installed. However, the application can also be run in an application or web server like Apache Tomcat if that is necessary. The data are stored in HSQLDB. This is an in-memory database that runs inside the Java process. This measure also reduces the technical complexity of the application.

The source code of the example can be downloaded at <http://github.com/ewolff/user-registration-V2>. An important note: The example code contains services that run under root rights and can be accessed via the net. This is not acceptable for production environments

because of the resulting security problems. However, the example code is only meant for experimenting. For that the easy structure of the examples is very useful.

1.6 Conclusion

Putting software into production is slow and risky. Optimizing this process has the potential to make software development overall more effective and efficient. Continuous Delivery might therefore be one of the best options to improve software projects.

Continuous Delivery aims at regular, reproducible processes to deliver software—much like Continuous Integration does to integrate all changes. While Continuous Delivery seems like a great option to decrease time to market it actually has much more to offer: It is an approach to minimizing risk in a software development project because it ensures that software can actually be deployed and run in production. So any project can gain some advantage—even if it is not in a very competitive market where time to market is not that important after all.

Endnotes

1. Jez Humble, David Farley: *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley, 2010, ISBN 978-0-32160-191-9.
2. <https://martinfowler.com/bliki/ContinuousDelivery.html>
3. https://en.wikipedia.org/wiki/Continuous_delivery.
4. Eric Ries. *Kanban: The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*, Crown Business, 2010, ISBN 978-0-67092-160-7.
5. David J. Anderson: *Kanban: Successful Evolutionary Change for Your Technology Business*, Blue Hole Press, 2010, ISBN 978-0-98452-140-1.
6. <https://www.sec.gov/litigation/admin/2013/34-70694.pdf>
7. <http://www.kitchensoap.com/2013/10/29/counterfactuals-knight-capital/>

Chapter 2. Providing Infrastructure

2.1 Introduction

This chapter focuses on an essential foundation for Continuous Delivery: provisioning infrastructure. In the different phases of the Continuous Delivery pipeline, software has to be installed on computers to perform acceptance and capacity tests or explorative tests. This approach requires automation since a manual installation would be much too laborious. In addition, automation will decrease the risk associated with bringing a new release into production since it guarantees that the necessary environments are always installed identically.

[Section 2.2](#) shows how simple installation scripts can support automation. However, such scripts are often not sufficient. Therefore, [section 2.3](#) discusses a tool for infrastructure automation: Chef. [Section 2.3.1](#) first describes the technical basics. Chef can either be used as a simple command line tool (Chef Solo, [section 2.3.2](#)) or as a client/server solution ([section 2.3.4](#) and [section 2.3.5](#)).

Vagrant ([section 2.4](#)) is an excellent tool for installing virtual machines on developers' computers. [Section 2.4.1](#) provides a concrete example for using Vagrant and Chef. [Section 2.4.2](#) finishes this topic with a discussion of the usefulness of Vagrant.

Another alternative is described in [section 2.5](#): Docker is not only a lightweight alternative for virtualization, but also uses a very simple approach for the installation of software in Docker containers. The section discusses the use of Docker with Vagrant ([section 2.5.4](#)), as well as the installation of Docker on servers with Docker Machine ([section 2.5.5](#)), more complex Docker setups ([section 2.5.6](#)) and the coordination of multiple Docker containers with Docker Compose ([section 2.5.7](#)). It is easiest to reproduce the installation of software when the installation on a server is never changed—in line with the concept of the “immutable server” ([section 2.6](#)).

The use of tools like Chef and Docker results in fundamental differences with regards to how infrastructure is handled. [Section 2.7](#) explains the effects of “infrastructure as code.”

[Section 2.8](#) shows how Platform as a Service Clouds (PaaS Clouds) can be utilized for Continuous Delivery.

Finally, [section 2.9](#) focuses on the specific challenges with regards to data and databases. Databases are especially difficult to install and to update since they contain a lot of data which might have to be migrated depending on the circumstances. Moreover, it is not easy to generate and provide suitable test data sets for the application. The chapter then finishes with a summary and conclusion in [section 2.10](#).

2.1.1 Infrastructure Automation: An Example

Section P.2 described the scenario of Big Money Online Commerce Inc., which did not use Continuous Delivery. However, Big Money Online Commerce Inc. did learn from its errors. One measure the company introduced was infrastructure automation. This makes it substantially faster to generate environments with the necessary software and renders the process more reproducible. In fact, the infrastructure automation solved the following problems for the team:

- Generating the testing environment had always been very laborious. Since this process is now automated, the effort to generate such an environment has become much smaller. This was a prerequisite for generating multiple testing environments which now allow for more comprehensive tests.
- In the scenario that was presented at the start of the book an error had happened in production because the production environment had not been built in the same way as the testing environment. Since the infrastructure is now automated, it can be reproduced exactly—this is true for the testing as well as for the production environment. Therefore, this error source is now abolished, and such errors indeed did not occur again.
- In addition, nightly releases into production are also easier: It is just an automated process that has to be initiated. Manual errors are impossible. And most important: The release process has already been used for the generation of the other environments. Consequently, it has been thoroughly tested.

To do so the team had to use technologies which are also able to generate complex environments—this is exactly the topic of this chapter.

2.2 Installation Scripts

System configuration and software installation have been automated for a long time already: Windows software uses, for instance, Windows installers to do so. However, the situation is different when the software is written in-house and has to be brought into production: In this case configuration and installation often have to be performed manually. In some cases the necessary steps are described in a handbook. These steps have to be manually executed. Of course, even if checklists are used for verification, it is rather difficult to follow complex guidelines absolutely correctly so that this process is quite error prone and hardly reproducible.

2.2.1 Problems of Classical Installation Scripts

Some projects have turned to automating software installation with the help of scripts. The scripts implement the necessary sequence of steps to create the required files and the correct configurations. For the example application in this book—the user registration—it is, for instance, first necessary to install a Tomcat server. If Linux is used, a possible installation script could look like [Listing 2.1](#): `apt-get` is a service program for the Ubuntu Linux distribution. Initially the script updates the index of available packages and installs all available updates with `apt-get`. Then it installs OpenJDK and Tomcat. Finally the application is copied into the directory in which Tomcat expects the executable applications. Prior to running the script the application is copied onto the server. It is also possible to download the application via HTTP onto the server. This allows installation of the application directly out of a repository.

Listing 2.1 *Installation script for user registration*

[Click here to view code image](#)

```
#!/bin/sh
apt-get
apt-get dist-upgrade -y
apt-get install -y openjdk-8-jre-headless
apt-get install -y tomcat7
cp /vagrant/user-registration.war /var/lib/tomcat7/webapps/
```

At first this approach looks convincing due to its simplicity. However, many of the typical challenges are not solved:

- Changes to the port that the Tomcat server uses or to the memory configuration for Tomcat are not possible with this approach. They would require that Tomcat configuration files be modified.
- In the case of a new deployment, such scripts might cause problems. In the given example the application will be copied again into the directory of Tomcat. This will cause Tomcat to restart the application, which will lead to a short downtime of the application. In the case of a complex application this could also result in users being logged out of the application. Of course, the script could be modified such that the file is only overwritten if it is not yet present or has been changed. However, this would make the script more complicated and would also have to be tested.
- When the server configuration is modified, this script is not able to return the server back to the correct state in all cases. When, for example, one of the Tomcat configurations is changed, this installation will not be able to undo this modification.

Apart from these problems, which are associated even with such a simple script, there are additional fundamental challenges:

- It is laborious to install a real application including all components. Of course, it is even more laborious to automate this process. Therefore, many attempts to automate never reach completion since the costs for a full automation are too high. Consequently, a handbook and manual interventions are still required. It can be very difficult to perform the manual steps and interventions in the right place—this often requires comprehensive knowledge about the script. The tools presented in this chapter facilitate the implementation of such automation and scripts and therefore make it easier to really achieve full automation.
- When an installation crashes, it has to be restarted. In such a scenario the system is in a state where some parts have already been installed. Just to start the installation again can create problems. The script usually expects to find the system in a state without any installed software. If the script then tries to create folders and files, these

operations can fail and might thereby interrupt the entire installation. To fix such problems the installation script would have to be able to deal with these special cases and would have to be tested in different environments.

This problem is also the reason why updates to a new software version are problematic. In such a case there is already an old version of the software installed on the system that has to be updated. This means that files might already be present and have to be overwritten. The script has to implement the necessary logic for this. In addition, superfluous elements that are not required anymore for the new version have to be deleted. If this does not happen, problems can arise with the new installation because old values might still be read out and used. However, it is very laborious to cover and automate all update paths that occur in practice. On the other hand, one can circumvent this problem by just building the server from scratch for each new release.

In the end such simple scripts are often sufficient. But when changes to the configuration are necessary or when, in addition to an installation, an update of an old version also has to be supported, the scripts quickly get very complicated. Thus it can be sufficient to implement installation scripts if software just has to be completely newly installed. Docker ([section 2.5](#)) is well suited for building servers completely anew for each new software release and therefore often uses installation scripts.

downtime of the application. Therefore, the overwriting has to be implemented in a proper manner. Problems can also arise if the rights for accessing the file have not been set correctly, so that the file is present but cannot be read. Of course, the required directories have to exist, too. Besides, overwriting a file in a running system can lead to additional problems. For example, applications that are accessing the file can run into trouble during write access to the file.

An alternative would be to describe which content the file is supposed to have and which access rights should be set. In that case the desired state could be compared with the current state during the installation and the file could be adjusted accordingly. Such an approach has a number of advantages:

- The installation can be repeated as often as required—the result should always be the same. Such an installation procedure is called idempotent.
- Updates to a new version are relatively easy since it is not assumed that the system has to be built entirely anew. While a system where an older software version is installed is a special case, it can nevertheless be dealt with by the scripts. In that case the file might already be present, but is then overwritten with a new version if necessary.
- The changes to the system can be documented. Each change to a file or installation of specific software packages can be recorded in a log file. When, for instance, a file with new content is written, the old content can be automatically backed up so that the change is traceable. A regular installation script again would have to implement special cases to do so.
- Finally, if the file is already present and the access rights are set correctly, there are no changes necessary at all. This can profoundly speed up the process.
- Often a process has to be restarted to make sure that the new file is really read. The restart can now likewise be ensured—this, however, is only necessary if the file content really changed.

Ideally, the system has to be described completely—that is, including the information about which version and which updates of the operating system

have to be installed. This is the only way to ensure that the installation is really completely repeatable.

Chef, which will be introduced in this section, belongs to the software systems following this approach.

2.3.1 Chef versus Puppet

Another system with a similar approach like Chef is Puppet.⁴ The Continuous Integration infrastructure from the next chapter is generated with a Puppet script. However, the implementation differs in important points: Puppet pursues a declarative approach with a Ruby DSL (domain-specific language) and JSON data structures. The user defines dependencies, and Puppet installs the software based on a plan that it constructs based on the dependencies and components. While Chef also uses a Ruby DSL, the user writes a kind of installation script instead of the declarations used by Puppet. Since a domain-specific language is used in both cases, it is not strictly required that one be proficient in Ruby in order to use Chef or Puppet. However, Chef and Puppet can easily be supplemented with all the power of Ruby. This approach is also easier to understand for developers.

Another advantage of both approaches is that the technologies are relatively widespread. Therefore, installation scripts exist for many typical software packages, which can be used for your own system installation. However, these scripts have to be approached with some caution: Most of the time they have to be adapted to your own needs for your own infrastructure, because, for instance, additional modifications in the configuration files are necessary in the context of the project. In some cases the project can require installation of plug-ins for software, and the installation is not contained in the generic script. It can also be necessary to adjust specific settings—for example, the network port the process uses—but these options are not part of the regular scripts. In addition, not all variants of operating systems are necessarily supported. Linux is not just Linux—different distributions store configuration files in different places and software packages have different names. Even if the scripts seem to take different Linux variants into consideration, there might be problems in real life because the scripts have not been tested on each version of the operating system.

In addition to different Linux derivatives, Puppet and Chef also support Windows and Mac OS X. Therefore, the approaches described here are also adaptable to these platforms—however, there are of course differences when it comes to details. For example, neither Windows nor Mac OS X comprises a package manager as part of the basic installation.

Chef and Puppet are open source projects under an Apache 2.0 license. This is a very liberal license so that from this perspective nothing argues against their use. In addition there are commercial versions available from the companies Chef and Puppet Labs. Therefore, it is either possible to use the products for free or to buy greater security by paying for commercial support.

2.3.2 Other Alternatives

In addition to Puppet and Chef there are many other tools. Some examples are:

- Ansible⁵ focuses on having its own syntax for installation scripts, which it calls playbooks. YAML files are used for defining the servers. Servers are installed remotely via SSH. This approach is simple and secure via SSH—and, most of all, it hardly requires any additional software on the systems which are to be installed. However, it can be necessary to install Python libraries on the machines because Ansible is implemented in Python. It supports, in addition to Linux, Windows, and Mac OS X. There is a document, “how to get started with Ansible”⁶ that is very useful. An example of the installation of Tomcat⁷ is provided.
- Salt—or more specifically the SaltStack⁸—are likewise implemented in Python. Salt is based on a Master Server and Minions which run as Daemons on the administered systems. These components communicate via ZeroMQ—a messaging system. This allows Salt to communicate quickly and effectively and therefore to scale on a large number of systems. Linux, Mac OS X, and Windows are supported. There is also a tutorial.⁹

The implementation of Ansible started in 2012, and Salt has been on the market since 2011. So these technologies are substantially newer than Chef

(2009) and Puppet (2005). Consequently, they do not have such a large community yet, and there is not as much experience using them in practice. On the other hand, they solve a number of problems which are encountered in the use of the older technologies.

2.3.3 Technical Foundations

In general Chef can be used in three different ways:

- *Chef Solo* is the simplest variant. Here, Chef is used as normal command line tool—a kind of universal installation script. However, the entire configuration has to be present on the computer. This can be sensible for generating systems for developers who get by without a large infrastructure. For example, Vagrant ([section 2.4](#)) uses this approach. With the same approach servers can be installed as well. This requires, for example, that the configuration be read from a version control like Git and then the installation process is triggered. This allows the installation of large environments without the need for a central server. However, these configurations have to be especially secured since they can, for instance, contain the passwords of the production databases. Likewise the rollout of changes to the different servers should be automated. This requires Chef Solo to be executed again on the servers.
- *Chef Server* configures and manages a number of Chef clients. The client itself possesses only a very small core. This approach is especially useful when several computers have to be installed. Besides, this approach makes it possible for the server to keep track of all computers. In this way, a suitable entry for each computer can be generated on a monitoring system. In addition, it is possible to execute the requests via the computers and the installed software. This allows, for instance, automatic supplementing of the configuration of a load balancer when a new web server has been installed.
- *Chef Zero* is a variant of the Chef server which runs in-memory. This makes it small and very fast. Chef Zero is especially useful for tests where a Chef server would be too slow or the setup too laborious.

A commercial variant is *Enterprise Chef*. This version has a number of additional features—for instance, support for tenants or authentication with

LDAP or Active Directory. This version is also available as a hosted system. In that case it is the company Chef Inc. that runs the Chef server. This is a useful approach when an infrastructure is supposed to be operated in a Cloud environment like Amazon Web Services. In that case it is not necessary to care anymore about how the Chef server is installed, how backups are made, or how sufficient availability can be guaranteed. These tasks are taken care of by Chef Inc.

Basic Chef Terms

Three terms are central for the configuration of systems with the help of Chef ([Figure 2.1](#)):

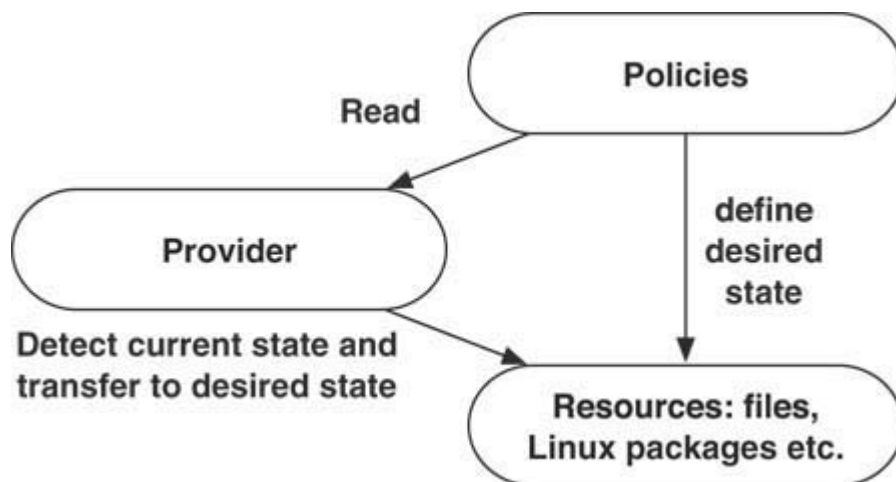


Figure 2.1 *Resources, policies, and providers*

- *Resources* comprise everything that can be configured or installed: for example, files or Linux packages. Chef supports a multitude of resources, such as code repositories, network adaptors, and file systems.
- *Policies* define what resources should look like. For example, a policy can state that the user “wolff” should be there or that the package “tomcat” should be installed. Developing automation with Chef consists mainly of generating such policies.
- *Providers* are responsible for determining the current state of a resource and to modify it so that it is in line with a policy. The built-in providers of Chef support typical operating system resources, such as files.

whether the owner of the file and the access rights are correct. Only when a change of the files is necessary is the Tomcat service started anew.

Obviously one recipe on its own makes little sense. Different additional elements are required, such as the referenced templates. Therefore, a recipe is usually stored in a directory structure: a so-called cookbook. It consists of a directory with different sub-directories ([Figure 2.2](#)). Typically, a cookbook contains the following components:

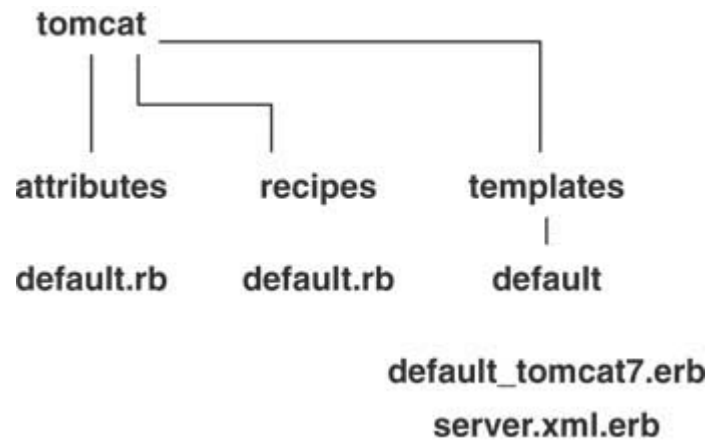


Figure 2.2 *Directories of a Chef recipe*

- In the directory `recipes` the discussed recipes are found; they define the policies for resources. Usually, there is a recipe in the file `default.rb`. For complex installations additional recipes can be defined, which can be stored in other files.
- Within the `recipes` values can be adjusted that are specific for a certain server. This requires that attributes be defined in the recipes. The possible attributes and their default values are defined in the directory `attributes`. As will be discussed later, these attributes can be overwritten outside of the recipes so that the recipes can be used in different contexts. Consistently, “Convention over Configuration” is implemented. This approach means that sensible defaults are used if no specific configuration is provided.
- Additional files can be stored in the directory `files`. These are files that can be copied onto the servers without additional changes. However, it is often more sensible to load files; for instance, from a repository server.

recipes and the values for the attributes to Chef upon each installation. However, this does not make a lot of sense. In the end it is always the same kind of servers that have to be installed in each phase of the Continuous Delivery pipeline. Besides, there can be several identical servers; for instance, when multiple web servers have to be provided behind a load balancer.

Therefore, Chef uses the concept of “roles”: A role defines which recipes are executed and which values the attributes are supposed to have. Each type of server requires the definition of a role. Subsequently, it is very easy to install an arbitrary number of such servers. In addition, the Chef server can be used to execute queries, such as how many servers are there with a certain role, and which servers they are. In this manner, a load balancer whose configuration files contain, for example, all web servers can be installed.

The information for a Chef role is stored in a JSON file; alternatively, a Ruby DSL can be used. An example is shown in [Listing 2.5](#). For a role definition the `json_class` and the `chef_type` have to be set to the values in the listing. Afterwards information follows that specifies which cookbooks are executed and which values the attributes should have. Initially the role is denoted as “tomcatserver,” and this role is described. Then the individual attributes of the cookbook `tomcat` and `webapp` are adjusted. Finally, the `run_list` defines which recipes are to be executed.

Listing 2.5 *JSON configuration of a Chef role*

[Click here to view code image](#)

```
{
  "json_class": "Chef::Role",
  "chef_type": "role",
  "name": "tomcatserver",
  "description": "Install Tomcat and a web application inside
Tomcat",
  "default_attributes": {
    "webapp": {
      "webapp": "demo.war"
    },
    "tomcat": {
      "port" : 8080
    }
  },
  "run_list": [
```

```
"recipe[apt]",  
"recipe[webapp]"  
]  
  
}
```

Each server—denoted as a node in the terminology of Chef—can have one or multiple roles. Nodes can also overwrite attributes to further adjust each server individually.

2.3.4 Chef Solo

The easiest approach for using Chef is Chef Solo. Here, the provisioning of software on the system is triggered by the command line. This means that the configurations and installation scripts have to be transferred onto the system. Therefore, the use of Chef Solo makes sense for test scenarios and local developer machines where this limitation is not critical. In the case of a distributed installation on multiple computers, version control tools and other tools for the transfer of files can also be used to support these scenarios. The infrastructure for such an approach is often easier than the installation of a central Chef server. In addition, this strategy helps to avoid a central bottleneck.

Try and Experiment

This task is meant to help the reader to gain experience with Chef Solo. The objective is to install a simple Java application completely with a Tomcat web server and JDK.

1. This task is best solved on a virtual machine. Therefore, install software like VirtualBox.¹³
2. Now install a VM with Ubuntu 15.04. The CD image can be downloaded at <http://releases.ubuntu.com/15.04/>. The server version is sufficient. Please make sure to install version 15.04—with other Ubuntu versions there can be problems with the scripts.
3. Install only a simple server. You do not need any of the special packages that are offered during the installation. However, it might be easier to install an SSH server and to perform the subsequent steps per ssh—in that case you can use, for instance, copy/paste from other windows.
4. Start the VM with the installed operating system.
5. Update the software. The following commands are used for updating:
`sudo apt-get update` and `sudo apt-get upgrade`.

Now you have a VM with a simple Ubuntu installation without special software. The next step will be to install Chef.

1. Install Chef. During the installation a server is asked for—however, it is not necessary to answer this question. The command for installing is:
`sudo apt-get install chef`.
2. In addition, install the version control Git with: `sudo apt-get install git-core`.
3. Fetch the GitHub repository using the command: `git clone https://github.com/ewolff/user-registration-V2.git`.
4. Adjust `solo.rb` in the directory `chef`: In the first line the variable `root` is defined. It has to contain the directory in which the Git repository is contained—for example: `root = '/home/ubuntu/user-registration-V2/chef/'`.

thereby roll out changes; however, with Chef Solo nobody keeps track of the installed servers. But such an overview would certainly be very helpful.

2.3.6 Knife and Chef Server

With Chef Server, information pertaining to roles or cookbooks is kept centrally on a server. Such a Chef server can of course be installed.¹⁴ This is necessary if you want to keep your cookbooks and roles exclusively in your own computing center. In this case backups and disaster recovery are mandatory since roles and recipes should not get lost. However, a high availability does not necessarily have to be ensured since a server failure only means that no new computers can be installed and that roles and recipes cannot be modified.

The alternative is to use Chef Enterprise, where the Chef server is supplied by Chef Inc. and can be used. Here, there is no additional expenditure for operation.

Knife represents a kind of “remote control” for a Chef server. It can administer roles, cookbooks, and so on on the Chef server. However, a lot more is possible: Knife can interact with a virtualization solution. Knife can also install a Chef client on a computer. When this setup is properly configured, a new computer can be started and the software installed on the computer without any manual intervention.

Try and Experiment

Let us have a look at Chef Server and Knife. By using the Amazon EC2 Cloud infrastructure and the Hosted Enterprise Chef Server we can get an impression of the possibilities offered by Chef Server and Knife without large investments in infrastructure.

This example can be run on any computer. In contrast to the previous example it does not require the use of a virtual machine. However, you can of course reuse your virtual machine to install the necessary software separately.

1. Install Git (see <http://git-scm.com/book/en/Getting-Started-Installing-Git>).
2. As before, we will employ the example of the Tomcat server.
Therefore, you have to check out the project with `git clone https://github.com/ewolff/user-registration-V2.git`
3. Then you have to install Knife, which is part of the Chef Development Kit. Instructions for the installation can be found at <https://downloads.chef.io/chef-dk/>.
4. Finally, the Knife EC2 plug-in has to be installed—see <https://github.com/chef/knife-ec2#installation>.

Now we have assembled all software required for this example. The next step is to open an account at Amazon and to transfer the information into the configuration:

1. You need an account at Amazon which can be set up at Amazon at <http://aws.amazon.com/>, which provides new users with a considerable amount of free usage of virtual machines. However, this comprises only micro instances, the least powerful virtual computers in the Amazon Cloud. Nevertheless, it makes sense for tests to use micro instances, if possible, since they are very cheap and still normally sufficiently powerful for tests.
2. At https://console.aws.amazon.com/iam/home?#security_credential an access key can be generated. It consists of an access key ID and a secret key. The environment variables `AWS_ACCESS_KEY_ID` and

Now Knife can start a virtual machine with Ubuntu. A hosted Chef server is responsible for the cookbooks. This server has to be set up next:

1. We need a Hosted Enterprise Chef account, which can be obtained at <https://manage.chef.io/>. Hosted Enterprise Chef is free of charge up to a certain number of computers.
2. At <https://manage.chef.io/organizations/> you can create an organization.
3. Download the validation key for the organization and save it in the `.chef` directory.
4. In `knife.rb` the file name has to be set in `validation_key`.
5. Now the organization name has to be entered in `knife.rb`—at the definition of `validation_client_name` and `chef_server_url`.
6. Click on the link “Users.” There you can have a key created for your account—a file with the extension `.pem`. Download the key, save it in the directory `.chef`, and change the line with `client_key` in `knife.rb` to the name of this file.
7. The name of this file—without extension—has to be set as the value for the `node_name`.

Now everything is configured. When we now simply execute `knife node list` in the directory with the sub-directory `.chef`, we should obtain a sensible response—namely that there are no nodes yet, since we have not configured any servers yet.

As the next step we will feed the necessary information into the Chef server:

1. Enter the command `knife cookbook upload -a` into the directory `user-registration/chef`. This will cause all cookbooks to be uploaded onto the server. Uploading the web applications can take some time.
2. Do the same with the role `tomcatserver` via `knife role from file roles/tomcatserver.json`.
3. Now you can use `knife ec2 server create -r 'role[tomcatserver]' -i .chef/<Amazon PEM>.pem -r 'role[tomcatserver]'` to start a new server onto which Chef and Tomcat will be installed. The file name,

which is passed in as an argument, is the file belonging to your key pair at Amazon.

Using a single command we have now started a server that is also accessible from the internet. When the server started, the public DNS name was stated under which the server is now accessible. A curl can be used to contact the application, for instance via `curl http://:8080/demo/`.

Let us have a look at the newly started node by using Knife—this can be done with `knife node list`. This command returns all running nodes. `knife node show <instance-id>` shows more detailed information about the nodes. These instances should now be deleted with `knife ec2 server delete <instance-id> --purge`—otherwise costs are incurred. If there were difficulties during the setup of the machines, it might happen that they do not appear in the list of nodes—in such a case please use the EC2 console mentioned in the boxed text to also delete these servers.

One more comment: The virtual machines that are used in this example are very slow. Real machines for production are considerably faster. For readers who feel like experimenting: It is possible to generate AMI images of running instances in the EC2 console, and also of the instances that we started with Knife. In that case the entire content of the hard disk is saved. Via this new AMI machines can be started. They contain the entire software that was previously installed, but without an installation process—that is, practically immediately.

Additional ideas for experimenting:

- It is also possible to view and monitor the running nodes in the AWS console at <https://console.aws.amazon.com/>, in the EC2 dashboard of the EU-West-1 region.
- The software could be started with another version of Ubuntu. The names of the AMIs that have to be entered in `knife.rb` can be found at <http://uec-images.ubuntu.com/>.
- The machines can also be started in another region. To do so another image has to be used—a list can be found at <http://uec-images.ubuntu.com/vivid/current/>. This image and the region have to be entered in `knife.rb`. In addition, a new SSH key pair has to be generated at Amazon for each region.

- Configure a virtual machine in such a way that it can use files of the host. This allows the virtual machine, for instance, to access Chef recipes that are stored on the host.
- Now Vagrant can log in on the virtual machine via SSH and install the machine with Chef Solo (see [section 2.3.2](#)). In addition to Chef Solo, Puppet or simple shell scripts can be used.

Vagrant can also be used to generate and provision multiple virtual machines.

For which purposes is it worthwhile to use Vagrant? Vagrant is very easy to install on a laptop or a developer machine. In addition, virtualization software is needed. Once both are in place, software can automatically be installed on the local computer on one or multiple VMs. Since the same tools can be used that are also employed for installations in production, the developers can try out the software in an environment that is similar to production.

2.4.1 An Example with Chef and Vagrant

To better illustrate Vagrant let us return to the example that was introduced in [section 2.3.2](#). We will again install Java, a Tomcat server, and a Java web application on a computer.

Central for Vagrant is the Vagrantfile. This is actually in Ruby-Code—even if it maybe does not look like it at first glance. In fact, a Ruby DSL is used here—like previously in the case of Chef.

[Listing 2.6](#) shows the Vagrantfile for the example. Here, it is defined which base image is supposed to be used for the system. This image is stored locally. This is followed by port forwards: Certain ports of the computer are redirected to ports of the virtual machine. If for instance the URL `http://localhost:18080/` is opened, it is redirected to port 8080 of the virtual machine—where Tomcat should be running. Finally, the provisioning of the software is configured with Chef. It contains the paths to the cookbooks and the roles including the role the server is supposed to assume. Since the virtual machine can access the directories of the host, it is very easy in this manner to provide the VM with cookbooks and roles.

Listing 2.6 *Vagrantfile*

[Click here to view code image](#)

```
Vagrant.configure(2) do |config|
  config.vm.box = "ubuntu/vivid64"

  config.vm.network "forwarded_port", guest: 8080, host: 18080
  config.vm.network "forwarded_port", guest: 8081, host: 18081

  config.vm.provision :chef_solo do |chef|
    chef.cookbooks_path = ["cookbooks"]
    chef.roles_path=["roles"]
    chef.add_role("tomcatserver")
  end

end
```

With the command `vagrant up` the system can be started. This requires only that the Vagrantfile be in the current directory. The base image is downloaded, and the software is installed. `vagrant help` can be used to list additional commands. `vagrant provision`, for instance, can be employed to trigger a new Chef run. Thus, it is not necessary to build the VM completely anew—with `vagrant provision` only the necessary updating is performed, which can save a lot of time. `vagrant halt` can be used to stop the environment—only the VM is terminated. `vagrant destroy`, on the other hand, completely destroys a VM—including all files. Finally, `vagrant ssh` provides a shell on the virtual machine.

- Instead of a complete virtualization, Docker uses the so-called Linux Containers (LXC—Linux Container) which in turn employ the cgroups of the Linux kernel. This allows implementation of a lightweight alternative to virtual machines: All containers use the same underlying operating system. Therefore, only one instance of the kernel is in memory. However, processes, networks, file systems, and users are kept separate. In comparison to a virtual machine (VM), a container has a profoundly smaller overhead—it is easily possible to run hundreds of containers on a simple laptop. In addition, it is much faster to start a container than a virtual machine since no operating system has to be booted—just a new process is started. The container is easy to manage since it only requires an individual configuration of the operating system resources. In addition to LXC as a basis technology, Docker is supposed to also support technologies like BSD Jails or Solaris Zones in the future.
- The file system is optimized as well: Basis images can be used from which only reading is possible. At the same time additional file systems can be blended into the container onto which it is also possible to write. One file system can overlay another file system. This allows, for instance, generation of a basis image that contains an operating system. If software is installed in a running container or if files are changed, only the modified data have to be saved. This substantially reduces the requirement for memory for the container on the hard drive.
- There are interesting additional options: For instance, a basis image can be started with an operating system, and afterwards software can be installed. As mentioned, only those changes to the file system are saved that were done during the installation of the software. Based on this delta an image can be generated. Then a container can be started that blends in this delta in addition to the basis image with the operating system—of course, subsequently additional software can be installed. In this way every “layer” in the file system can contain changes. The real file system at run time can be composed of such layers. This allows efficient reuse of software installations.
- [Figure 2.4](#) shows an example for the file system of the running container. On the lowest level there is the Ubuntu installation. On top is the installation of Java. This is followed by the example application.

