**FH JOANNEUM - University of Applied Sciences**

# Continuous Delivery of Database Changes

**Master thesis**
**Submitted at the Degree Programme Information Management**
**for the degree of "Master of Science in Engineering"**

**Author:**
**Andreas Rait**

**Supervisor:**

**FH-Prof. Dipl.-Ing. Dr.techn. Peter Salhofer**

**Graz, 2015**

# Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgement has been made in the text.

Graz, December 2015 _____

Andreas Rait

# Table of Contents

**TABLE OF CONTENTS** ..... **I**
**LIST OF ILLUSTRATIONS** ..... **III**
**LIST OF TABLES** ..... **IV**
**LIST OF ABBREVIATIONS** ..... **V**
**ABSTRACT** ..... **VI**
**KURZFASSUNG** ..... **VII**
**1. INTRODUCTION** ..... **1**
1.1. CENTRAL QUESTIONS ..... 4
1.2. GOALS ..... 4
**2. DEFINITIONS** ..... **6**
2.1. DEFINITION: CONTINUOUS INTEGRATION (CI) ..... 6
2.2. DEFINITION: DATABASE CHANGE ..... 8
2.3. DEFINITION: DEVOPS ..... 10
**3. THE SOFTWARE FACTORY – CONTINUOUS DELIVERY** ..... **13**
3.1. A SHIFT IN THE SOFTWARE INDUSTRY ..... 13
3.2. PRINCIPLES OF CONTINUOUS DELIVERY ..... 13
3.3. BENEFITS OF CONTINUOUS DELIVERY ..... 17
**4. THE DEPLOYMENT PIPELINE** ..... **19**
4.1. WHAT IS A DEPLOYMENT PIPELINE? ..... 20
4.2. DEPLOYMENT PIPELINE BEST PRACTICES ..... 22
4.3. DEPLOYMENT PIPELINE: IMPLEMENTATION GUIDELINES ..... 26
**5. DATABASE CONTINUOUS DELIVERY** ..... **29**
5.1. ISSUES WITH DATABASES ..... 29
5.2. PRACTICES FOR INTEGRATING DATABASES IN CONTINUOUS DELIVERY ..... 31
5.3. STRATEGIES AND BEST PRACTICE FOR DATABASE DELIVERY ..... 35
**6. DATABASE AUTOMATION TOOLS** ..... **38**
6.1. DATABASE MIGRATION SCENARIOS ..... 38
6.2. RESEARCH ON DATABASE MIGRATION TOOLS ..... 39
6.3. TOOL CRITERIA ..... 44

6.4. VALUE ANALYSIS OF DATABASE MIGRATION TOOLS ..... 46

**7. CONCLUSION ..... 55**

**BIBLIOGRAPHY ..... LVII**

# List of Illustrations

Figure 1.1 - Costs of fixing a bug in classic software development approaches (AmbySoft, 2006) ..... 3

Figure 2.1 - Continuous Integration process (Hirt, 2015) ..... 7

Figure 2.2 - Traditional vs DevOps lifecycle phases (Craig, 2014) ..... 12

Figure 4.1 - Deployment pipeline trade-offs (Humble, et al., 2011) ..... 20

Figure 4.2 - Value Stream map of a common feature development process (Swartout, 2014) ..... 27

Figure 4.3 - Architecture of a deployment pipeline (Humble, et al., 2011) ..... 28

Figure 5.1 - Database versioning table before and after changes are applied (own illustration) ..... 36

Figure 6.1 – Scenario 1: Target environment in known state ..... 38

Figure 6.2 – Scenario 2: Target environment in unknown state ..... 39

Figure 6.3 – Test environment architecture ..... 46

Figure 6.4 - Liquibase changeset file structure ..... 49

# List of Tables

Table 5.1 - Database tasks that should be automated (Duvall, et al., 2007) ..... 35

Table 6.1 - Redgate tools - DLM feature overview (Red Gate tools, 2013)..... 40

Table 6.2 - Flyway feature overview (Flyway, 2015) ..... 41

Table 6.3 - Liquibase feature overview (Liquibase, 2015)..... 41

Table 6.4 - Datical additional features overview (Datical, 2013) ..... 42

Table 6.5 – Feature overview DbMaestro Teamwork (DbMaestro, 2015) ..... 43

Table 6.6 - Database migration tool criteria overview ..... 46

Table 6.7 - Use case: Setting up a database baseline..... 47

Table 6.8 - Use case: Executing database migration ..... 47

Table 6.9 - Use case: Performing several migrations ..... 47

Table 6.10 - Use Case: Generating migration scripts..... 48

Table 6.11 - Use case: Generate a script for referential integrity refactoring ..... 48

Table 6.12 - Use case: Testing integration capabilities with Jenkins..... 48

Table 6.13 – Value analysis Liquibase..... 50

Table 6.14 – Value analysis FlywayDb ..... 51

Table 6.15 – Value analysis Redgate tools: DLM ..... 52

Table 6.16 – Overview migration tool analysis including total score..... 53

# List of Abbreviations

| | |
|---|---|
| **CI** | Continuous Integration |
| **DBA** | Database Administrator |
| **API** | Application Programming Interface |

# Abstract

Continuous Delivery is a holistic software development approach that aims to utilize agile practices to improve efficiency and quality of a software product. Despite the fact that many IT professionals have heard about Continuous Delivery, the concept, requirements and benefits are often not clear. This thesis aims to present concepts and best practices of Continuous Delivery to provide IT professionals with knowledge and tools. Furthermore the thesis analyzes issues related to releasing database changes and provides tools and best practices to overcome them.

Based on literature research the thesis first describes the underpinning concepts and benefits of Continuous Delivery. Second best practices for Continuous Delivery are described. Furthermore challenges of delivering database changes in a continuous approach are presented. Based on the challenges a research on best practices for delivering database changes is shown. The challenges and best practices are used to research and evaluate tools that support database change delivery.

The research shows that Continuous Delivery builds on common software practices in order to achieve a reliable release process. Continuous Delivery emphasizes an automated release process from source control to the customer. Thus the concepts and best practices focus on a holistic view of the release process, high level of automation, continuous improvement and DevOps culture for delivery teams. The research on database delivery shows that database development practices do not fit to current application development practices. Hence best practices found for database change delivery describe techniques to better integrate databases in an automated release process. The research on tools to perform database delivery tasks shows that there are open source and proprietary solutions that provide features to better automate database delivery and integrate it into an automated release process.

Knowing the concept and goals of Continuous Delivery the benefits like fast feedback and focus on quality are obvious. The challenges come on organisational level as well as in small scale, when improving tasks of the process to be as efficient as possible. Creating a unified view on the release process and achieve a seamless process requires different organisational functions to work together closely and share assets and responsibilities. The tools found for databases can help to make database change delivery tasks more automated and thus less dependent on DBAs. Automating database related tasks allows DBAs to concentrate on valuable tasks and empowers other parts of the release process to perform their tasks more independently. Hence utilizing such tools implies a step towards Continuous Delivery.

# Kurzfassung

Continuous Delivery ist ein ganzheitlicher Ansatz der Software-Entwicklung, welcher agile Techniken nutzt, um Effizienz und Qualität eines Software-Produkts zu verbessern. Trotz der Tatsache, dass viele IT-Verantwortliche von Continuous Delivery bereits gehört haben, sind Anforderungen und Vorteile oftmals nicht klar. Diese Arbeit zielt darauf ab, Konzepte und Best Practices von Continuous Delivery zu präsentieren, um IT-Experten Wissen und Tools zu bieten. Darüber hinaus werden Datenbanken im Zusammenhang mit Continuous Delivery untersucht. Außerdem werden Tools und Best Practices vorgestellt, um damit verbunden Probleme zu überwinden.

Auf der Grundlage von Literaturrecherchen werden in dieser Arbeit zunächst die unterliegenden Konzepte und mögliche Vorteile von Continuous Delivery aufgezeigt. Danach werden Best Practices für Continuous Delivery vorgestellt. Im nächsten Schritt werden Herausforderungen von Datenbankänderungen beschrieben. Basierend auf den Herausforderungen wird eine Recherche über Best Practices für die Auslieferung von Datenbankänderungen vorgestellt. Diese werden als Grundlage verwendet, um eine Analyse von Datenbanktools durchzuführen, welche die Auslieferung von Datenbankänderungen unterstützen. Abschließend wird eine Bewertung dieser Tools präsentiert.

Die Recherche zeigt, dass Continuous Delivery auf eine kontinuierliche Verbesserung des Release-Prozesses abzielt, in dem bekannte agile Software-Praktiken angewandt werden. Continuous Delivery stellt dabei einen durchgängigen Release-Prozess dar. Die unterliegenden Konzepte und Verfahren konzentrieren sich auf eine ganzheitliche Betrachtung des Release-Prozess, einen hohen Automatisierungsgrad, kontinuierliche Verbesserung und eine Anwendung der DevOps-Prinzipien. Die Recherche über Datenbankauslieferung zeigt, dass aktuelle Datenbankpraktiken oft nicht mit Praktiken der Anwendungsentwicklung übereinstimmen. Daher wurden Best Practices für die Datenbankentwicklung recherchiert, welche Datenbanken besser in einen automatisierten Release-Prozess integrieren. Die Recherche über Datenbanktools zeigt, dass aktuelle Tools einige Funktionen bieten um Datenbankauslieferungen besser zu automatisieren.

Das Konzept und die Ziele von Continuous Delivery und die daraus resultierenden Vorteile wie schnelles Feedback und Fokus auf Qualität liegen auf der Hand. Die Herausforderungen liegen sowohl auf der organisatorischen Ebene, als auch in der Prozessoptimierung. Es braucht ein einheitliches Verständnis, um einen durchgängigen Release-Prozess zu etablieren. Zusätzlich braucht es aber auch unterschiedliche Abteilungen die eng zusammenarbeiten und sich Mittel und Ziele teilen. Die gefunden Datenbanktools ermöglichen eine stärkere Automatisierung von Datenbankänderungen. Damit kann sich das gesamten Team stärker auf wertschöpfende Aufgaben konzentrieren. Daher helfen diese Tools dabei einen weiteren Schritt in Richtung Continuous Delivery zu machen.

# 1. **Introduction**

Since the arrival of the “agile manfisto” in 2001 the word “agile” stands for modern software development approaches and processes. Especially when Extreme Programming – one of the first agile development processes - was presented to the public, software engineers were fascinated and different agile methods and processes sharing similar ideas gained popularity (Fowler, 2005). For many software engineers the key argument in favour of agile development is the basic idea: to keep the design phase short, go to development phase soon and to delivery functional software after short iterations. Some of the agile approaches are very specific and therefore hard to adapt. Others define just the basic framework and emphasize the agile principles. These frameworks provide a set of best practices for development teams to build their own agile processes which makes them easier to adapt and therefore more likely to succeed. As a result currently more than 80 percent of the software development companies are familiar with agile practices and are actively using one or more of them. (VersionOne, Inc., 2013)

Continuous Integration (CI) is probably one of the most well-known agile software development practices which was introduced along with the agile software development approach Extreme Programming (Beck, 1999). Among the agile practices continuous integration is the technique which also has been broadly adopted by most of the software developers and is considered best practice now. A survey on infoQ – a community site for software professionals - conducted in 2012 (Bharti, 2012) showed that more than 55 percent of the software developers asked were using CI. This is an 11 percent increase compared to the survey taken 2010. At the time of writing, data of the 2014 survey is not available yet. Based on the recent developments another increase in adoption of CI is predicted. Besides the findings on adoption of CI the survey also shows that companies are beginning to adapt new practices which should improve their software engineering capabilities (Bharti, 2012). One of these new practices is Continuous Delivery. Continuous Delivery is a software engineering discipline that focuses on automation and the delivery of high quality software products. It is especially interesting because it emphasizes a holistic approach to the whole software delivery process (Fowler, 2013).

There are many reasons why agile practices like CI have been accepted so broadly and became so successful. Some of them are directly influenced by the change of the software market itself which is fast-paced, highly competitive and adapts to new technologies very quickly as a survey of Gartner displays (Gartner, Inc, 2014). Furthermore the industry has changed to a customer market because of products that are directly developed for customers like smartphone OS or apps, websites and others, which definitely require an increased focus and involvement of users. Tech-Companies like Facebook (Feitelson, et al., 2013) or Google are already publishing new features on a daily basis to involve users faster and to improve based on the feedback (Claps, et al., 2015). Thus also established companies like Microsoft, which followed a strategy of big releases are adapting their strategies on product releases.

Microsoft recently announced that their operating system Windows no longer will come in versions with big changes but similar to a service bringing updates continuously (Myerson, 2015). Agile practices emphasize customer feedback in contrast to classic development projects where no iterations were specified which would allow to include feedback. In classic waterfall projects requirements were identified and specified in a very extensive early phase of a project and this specification would hardly change anymore. This phase was followed by an extensive development phase without any end user feedback and often failing to adhere to deadlines for the finished product. The result was a product that in many cases did not meet customer expectations (Leffingwell, 2007). One expertise of software companies nowadays is to manage changing requirements effectively in order to stay competitive. (Aurum, et al., 2005) Although with profound requirements analysis the changes should be manageable, customers will be more satisfied if they can change their minds about some specifications after getting hands on the product. This is called requirements evolution and is a factor that is anticipated by agile practices (Aurum, et al., 2005). Having the capability to quickly align to customer needs and demands, a software company can and only will be competitive. This is also emphasized in the first principle of the agile manifesto:

> *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software*. - *(Beck, et al., 2001)*

Customer expectations should be met by providing a "*continuous delivery of valuable software."* Ensuring that everything that is developed is of value for the customer is another important aspect of agile practices and shows that quality assurance is an important part of the software process. Furthermore it suggests a software process that is capable of continuously delivering in order to achieve an early involvement of the customer. This is basically what can be achieved implementing Continuous Delivery: Create a sophisticated software delivery process that empowers teams to release high quality software frequently to the customer. Continuous Delivery builds on best practices like CI and automated tests, automated deployments and aims at assuring quality in the product. (Humble, et al., 2011) Apart from satisfying the customer there is no doubt that such practices are also an advantage for software developers to cut costs and keep their work efficient. In classic software development processes finding and fixing a bug in an early phase of the software project has a huge diminishing return over fixing it in later phases were the product is already in production. Figure 1.1 shows the cost of fixing a bug in different project phases:

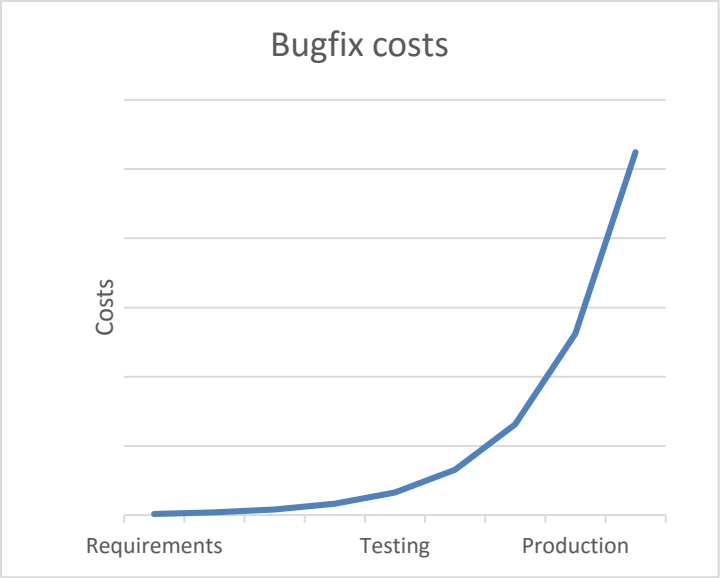


**Figure 1.1 - Costs of fixing a bug in classic software development approaches (AmbySoft, 2006)**

This is another problem of classic software development practices: fixing bugs in production comes with huge costs because it is not part of the standard process. Companies are trying to adapt to agile approaches for their development process because fixing bugs or adding changes is standard and practiced with every iteration. There are many examples of successful companies – for example Google, or Facebook (Claps, et al., 2015) - emphasizing an agile approach in order to have an efficient process. Referring to the survey of InfoQ which illustrated that companies are following the trends of software engineering and are continuously adopting to new agile practices (Bharti, 2012). When describing the Extreme Programming process, Kent Beck (Beck, 1999) writes that companies need and should accomplish a development environment where the curve above changes its shape from a parabola to a hyperbole. He argues that by implementing agile software practices in combination with best practices this should be possible. The idea is with the help of Continuous Integration, automated testing and an effective organization a process should be established which empowers developers to make no difference between fixing a bug in testing or production phase. Continuous Delivery is a discipline that very much aligns to this idea.

Continuous Delivery has become already widely recognized discipline under software professionals and seems to be another step towards effectively using common agile practices to create software of value for the customer. (Bharti, 2012) It is a holistic approach used to automate and improve the process of software delivery. Among the benefits of Continuous Delivery is the creation of a reliable and repeatable release process. This process in turn emphasizes high quality and generates reduction in cycle times and gets features and bugfixes to users fast (Humble, et al., 2011). Continuous Delivery deals with every aspect of a software product which also includes databases. Databases in particular require sophisticated tooling and clever strategies and can become an obstacle on the way to Continuous Delivery. Martin Fowler is one of the

pioneers of agile development and describes the problems of databases as the following:

> *"[...] one of the biggest questions is how to make evolutionary design work for databases. Most people consider that database design is something that absolutely needs up-front planning. Changing the database schema late in the development tends to cause wide-spread breakages in application software. Furthermore changing a schema after deployment result in painful data migration problems." (Fowler, et al., 2003)*

Databases are at the heart of an application because they provide and persist data which the application depends on. Especially with regard to enterprise applications a data loss or malfunction can interrupt business services and therefore costs a lot of money. Integration of Database Development and deployment practices into Continuous Delivery faces different issues due to common practices and legacy issues which need to be addressed or avoided from the start.

## 1.1. Central questions

In the interest of providing a reference to IT professionals for implementing Continuous Delivery this thesis covers the following questions:

- What are the principles, concepts and goals of Continuous Delivery?
- What are best practices and strategies to approach Continuous Delivery?
- What are the problems of releasing database changes and how to solve them following a Continuous Delivery approach?
- Is there a set of tools that in combination with current best practices can be used to implement Continuous Delivery in an enterprise environment?

## 1.2. Goals

The thesis pursues the following goals:

- Create an understanding of Continuous Delivery and how a software development process can benefit from it
- Provide strategies to cope with difficulties related to database changes in a Continuous Delivery environment
- Provide a research on tools and current best practices that support Continuous Delivery

One of the main goals of this thesis is to present Continuous Delivery to IT professionals in order to provide an understanding of the principles and how a software delivery process can benefit from it. Furthermore a starting point for implementing a Continuous Delivery environment should be given. This should be

achieved by providing practices and strategies especially for database change delivery. Additionally a set of open source tools and best practices is presented that could be used in an implementation effort.

# 2. Definitions

In the following paragraphs a common understanding of important terms should be established. This includes a definition and introduction of Continuous Integration principles and how it can be implemented in practice, a definition of a database change and the introduction to the DevOps movement.

## 2.1. Definition: Continuous Integration (CI)

Continuous Integration (CI) is a software development practice which forms the foundation of any continuous delivery effort. It does so by ensuring a stable software which can be delivered at any time. A sophisticated CI system enables further infrastructure implementations which helps to create:

- Fast and high quality feedback by aggregating information from the build system
- Metric provider for software quality which can be analyzed and used by project management
- A system of reports and installers for testing team
- A system that can be improved to support push-button deployments, using the deployment pipeline which can be used to make testing and put in operations easier, more effective and more automated (Humble, et al., 2011)

Continuous Integration was first discussed by Kent Beck in his book *Extreme Programming Explained* (Beck, 1999) where he describes the benefits of integrating changes into the codebase more frequently. Part of the philosophy underlying Extreme Programming is to take practices that are helpful or improve the development process and perform it on an *extreme level*. In case of Continuous Integration that means to integrate with every change made in order to get feedback as fast as possible.

Based on the first idea of Kent Beck CI has become such an integral software engineering practice for any agile methodology that it has been characterized best practice for agile software development (Claps, et al., 2015). Kent Beck formulated the basic principal is that small changes are integrated and tested immediately when they are added to a larger codebase. This allows to identify code errors very fast and provides valuable feedback to the developers in order to fix the error with the next change. CI has proven its benefits and tools were developed that focus on build and test automation and report generation. Running a proper CI system provides a constant feedback on the software status and bugs or defects can be discovered earlier in the development process. Additionally, but very much depending on how disciplined the development team is, a positive side effect is that defects are smaller

and less complex because of the smaller changes they come with. (Anderasson, et al., 2013)


**Figure 2.1 - Continuous Integration process (Hirt, 2015)**

Continuous Integration is a best practice and hence there are many different implementations of it. Because it has spawned as part of the lean development movement there is a lot of variability on how the CI process is implemented. Basically as shown in Figure 2.1 the process has four phases (commit, initiate CI process, test and report) which are repeatedly executed. Nevertheless many different factors and design decisions influence the implementation. (Stahl, et al., 2014) One of the most important design decisions is defining the trigger of the integration process. At this point there is a need to define the term continuous. A simple interpretation of continuous is “without interruption” which could mean that the CI process runs the entire day without anything to integrate. The question is how often the process should run in order to align to the requirements and goals of a development process. Although resources might be cheaper today utilizing them without a reason is not very economic and efficient. Referring back to Kent Becks idea every change that is added to the codebase should be integrated. (Beck, 1999) It is very reasonable that with every committed change the integration process should be started. Which means any change will trigger the whole process and enter the commit stage. The commit stage is also the entry point into the deployment pipeline a basic concept in Continuous Delivery, which will be described in a later chapter. Having a single entry point allows to have specific rules and checks that can be applied to ensure code quality and process alignment. (Rümmler, et al., 2014)

Although it would be ideal in terms of information quality to start the integration process for every new change, there are other factors that need to be considered as they influence the process: build time, cycle time, integration frequency, team size,

result interpretation, quality metrics etc. (Stahl, et al., 2014) Considering these factors it is sometimes not efficient to start the integration process with every commit and trade-offs have to be accepted. One strategy is to start the process with the first commit, collect all changes which are added in between wait for the first build to finish and start the next process with the collection of changes. The trade-off in this case would be the lower quality and inaccuracy of feedback. If an error occurs it cannot be mapped to one change anymore. There are various ways to conquer these problems in order find the best solution for the development team in place. (Humble, et al., 2011)

Although CI is well established and there are different CI tools around it needs proper design to unfold its benefits for a development team. Furthermore when thinking about adopting Continuous Delivery there is no way around implementing a proper CI system at first.

## 2.2. Definition: Database Change

Generally a database change is any alteration that is executed on the database. This includes a wide range of different activities e.g. adding a data row, adding a table, changing user privileges, adding a new constraint to a table or changing the logfile size etc. In the following paragraph the term database change will be discussed with a focus on relational databases. Although NoSQL databases are getting more popular particularly for web applications (Klettke, et al., 2014) there are reasons (e.g. performance, integrity, durability) which make relational databases a mandatory asset for enterprise applications. Database changes do come with different consequences as they change performance, structure or the state of the database. In the context of this thesis different database change are categorized in order to have a common understanding. The following enumeration describes the different categories of database changes and their scope of change:

- **Configuration change –** A change in the configuration of the database management system. Those are changes which could come along because new modules are added or specific DBMS parameters are changed to achieve an improved performance of database activities. Hence this kind of changes are relatively rare thinking of a database that will be configured similar for numerous deliveries of the software product. Nevertheless ensuring that the database configuration fits to the software product is part of Continuous Delivery practices.
- **Schema change –** A Schema change or schema refactoring is a simple change in the design of the database. This change affects the structure (tables, constraints, indexes etc.) or the behavior (procedures, functions) of the database or both which definitely affects the interaction with the source code. A change on the database schema could be necessary because of new features or a changed requirements. In a traditional approach database

schemas were entirely designed upfront and hardly ever changed. Since requirements of the software product are evolving this is not the case anymore and database schema changes are more frequently required.

- **Data change –** The data represents the state of an application. Changes to the state of the application are frequently performed if user or other processes are performing actions. Common actions are store, update, compute and delete data.

This categorization shows that there are different scopes for database changes which will require different expertise. For data change the DBMS is responsible by ensuring the ACID (atomicity, consistency, integrity, durability) principle in the database. Configuration and schema changes need to be managed by the operations and the development team. Schema changes often come with a certain complexity which causes stress for operators especially when those changes eventually are released to production systems. The other way round developers design and implement new features with the assumption that the database matches a certain schema which complies with the application code. Continuous Delivery practices try to reduce risks and make database schema changes a day-to-day task which will be explained in a later chapter.

In order to create an understanding why schema changes are problematic the following paragraph list different types in combination with an example and the possible consequences on performing such a change on production data. Scott W. Ambler and Pramod J.Sadalage have conducted some research on database refactorings and described their findings in a book. According to them a schema refactoring basically is a simple change in the schema in order to improve the design of the database. (Ambler, et al., 2006)

- **Structural:** Changing the definition of a table or a view. For example moving a column into a different table or splitting up a multipurpose column into separate columns for each purpose. Those actions require changes in the data as existing data needs to be refactored in order to fit to the new table definition. Taking the value from the multipurpose column and put it into the new columns depending on containing value is a very complex task.
- **Data quality:** A change which should improve the quality of the information contained in the database. For example making a nullable column non-nullable. Again this category of schema change requires merging effort for existing data.
- **Referential Integrity:** Adding a change that ensures that a referenced row exists within another table. For example adding a foreign key constraint. This does not require any changes in the data as long as the constraint is already fulfilled before the change is applied otherwise a preceding work needs to be done.

- **Architectural:** A change that improves the way in which external programs interact with the database. Adding a new stored procedure as to replace a previous existing java operation and make it available to non-java applications. This does not need a change in the data itself thus this behavioral change can be applied with less problems.
- **Method:** Refactoring a database method (stored procedure, function, trigger) that improves its quality. Renaming variables to make it better readable. Method refactoring do not need any effort in changing the data but may require changes in the application that uses it.
- **Non-Refactoring Transformation:** Different to the above categories this is a schema refactoring that changes the semantics of the database. Adding a new column to an existing table. Again this requires a strategy for existing data rows.

The list of different refactorings shows that deploying database changes is an activity that sometimes requires significant effort and expertise and sometimes becomes a project on its own. Particularly if there is a production database involved and release processes are not mature. Continuous Delivery emphasizes strategies to master database change releases and deployments.

## 2.3. Definition: DevOps

DevOps is an idea that currently captured a lot of attention in the enterprise IT milieu. The starting point was a conference in 2009 which was named “DevOpsDays” where a new movement was founded and a range of follow up conferences was initiated. Well-known research companies like Forrester and Gartner have included DevOps in their research and provide critical evaluations. (Die DevOps-Bewegung, 2012) Recently Gartner published a research paper on application development including organizational and technical developments. The overall statement regarding DevOps is the following:

> *DevOps practices are emerging among mainstream IT organizations to manage faster and more reliable software delivery; but more so than technology, this initiatives depend on organization and process innovations and architectural principles (Driver, et al., 2014)*

Following that statement Gartner also posted a prediction on the impact of DevOps initiatives:

> *By 2019, DevOps initiatives will cause 50% of enterprises to implement automated configuration and release management of the application life cycle (Driver, et al., 2014)*

The DevOps movement definitely triggered a hype and many software producer were following it by offering and advertising DevOps capabilities in their software products.

This is particular interesting because for a certain time there was no clear and common sense what DevOps really is (Die DevOps-Bewegung, 2012). In these paragraphs the origin and principles of DevOps are described.

The incentives of the DevOps movement came from a traditional conflict between developers and operators. The main objective of developers is to develop features based on requirements which come as demand from customers or innovation. Each feature should promote additional value to customers. The more features developers complete and are released the more positive is their reputation because they are measured against completed features. At this point it does not matter if any of the feature is actually in production. Operators have the objective to take the developed features and deliver them to production environments. Afterwards they have to make sure that the delivered feature is available to the customer. Additionally they are responsible to ensure that the production environment satisfies some quality requirements. Operators are measured against these quality requirements and how well they are able to fulfill them. In order to keep their production environments and thus their quality measurements stable, operators tend to protect their environments from change. This shows that from a traditional point of view the development and operating departments have different goals. Developers are interested in fast and frequent releases whereas operators almost avoid new releases. Both departments are eager to maximize their value for the company which often leads to conflicts. (Die DevOps-Bewegung, 2012)

The basic idea behind DevOps is to emphasize interconnected processes and better communication and collaboration between software developers and operators. Thus the name is a combination of “Dev” which represents the developers (programmer, testers and quality assurance personnel) and “Ops” representing the operators(experts who put software into production, system administrators, database administrators and network technicians) – DevOps. By advocating this collaboration the feedback loop should be shortened while empowering personnel and maintaining the alignment to the goals of both the development and operations department. (Claps, et al., 2015)

According to Hüttermann (Hüttermann, 2012) DevOps describes practices to streamline feedback from production environments to development in order to optimize the cycle time (i.e. time from inception to delivery). Furthermore DevOps has the goal to empower developers and operators with faster and more efficient software delivery process and the development of high quality software. Furthermore Hüttermann outlines some basic principles of DevOps, such as the following (Hüttermann, 2012):

- Culture: People over processes and tools. Software is made by and for people.
- Automation: Automation is essential to gain quick feedback.
- Measurement: DevOps finds a specific path to measurement. Quality and shared incentives are critical.

- Sharing: Create a culture where people share, ideas processes, and tools.

By creating an environment where operators and developers work together and communicate, another goal of DevOps is to ensure a mutual understanding. Developers should understand the issues associated with operations and vice versa. Thus together they are able to create and deliver high quality software faster and better aligned to the actual requirements. (Claps, et al., 2015)

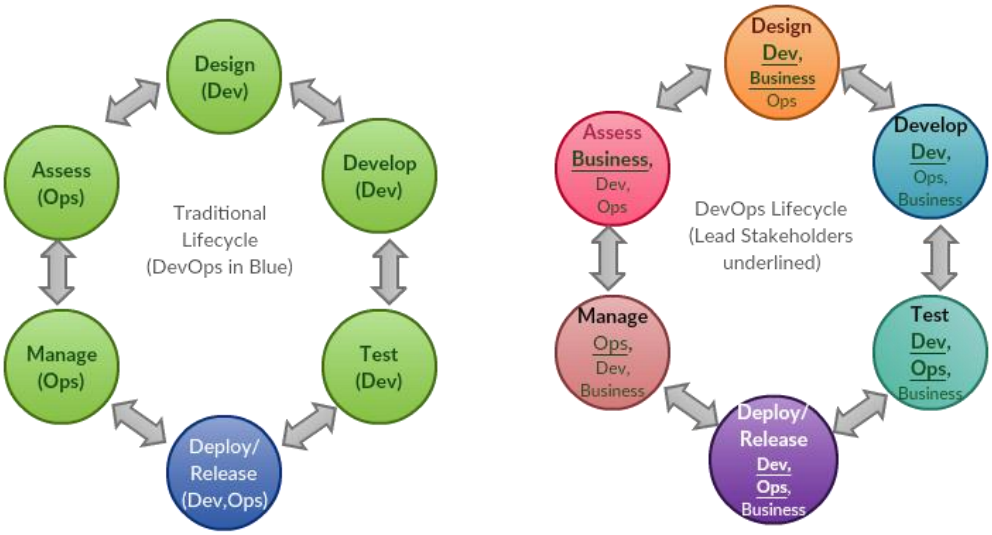


**Figure 2.2 - Traditional vs DevOps lifecycle phases (Craig, 2014)**

Figure 2.2 shows how a software lifecycle with DevOps teams differs from a traditional lifecycle. In traditional lifecycle only in the *Deploy and Release* phase developers and operators share responsibilities and work together. There are important phases where either developers or operators are solely responsible and therefore no communication and sharing of assets is performed. The only exchange is during *Deploy and Release* and at transition between *Asses and Design* which are therefore often prone to conflicts. Compared to that with inter-functional teams as emphasized by the DevOps movement different phases in the software lifecycle are handled co-operativley by the whole delivery team. Every function of the team is involved at every stage of the lifecycle therefore a mutual understanding can be established. DevOps emphasizes a culture of working together, empowering people and breaking down barriers between functions. Hence it provides a cultural and technical basis for a holistic software delivery process.

# 3. The Software Factory – Continuous Delivery

## 3.1. A shift in the software industry

In a recent research of Gartner (Driver, et al., 2014) analysts compare the current state of the software industry with the situation of manufactures in the early 1980s. Manufacturers in these days planned their assemblies on high volume mass-production model. These models required extensive planning efforts in order to produce months in advance. Only a few companies like Toyota were using different approaches and using minimal batch sizes to keep inventory levels at a minimum. Known as lean and just-in-time production this approach broke the conventional mass-production mindset. Under the pressure of the ongoing recession manufacturers quickly adopted the lean manufacturing concept which resulted in a worldwide change in manufacturing.

Compared to the situation of the manufacturers in the 1980's the software industry does not face a recession. The software industry is facing the digital business revolution and the Nexus of Forces (Driver, et al., 2014). Meaning a rapid change in business needs which makes large software release unworkable and similar to the manufactures there are companies in the IT which already have adopted lean software delivery methodologies and others still need to follow.

Another proper analogy is brought by Alan W. Brown (Brown, 2012) forecasting how software industry will develop by again looking at the manufacturing industry in the twentieth century. From handcrafted products to assembly lines to introduction of automation and cost minimizing. Production approaches of industrial manufacturing have been evolving through many changes. Today it is driven by the need for cost optimization, flexibility and reduction of waste. Consequently manufacturing processes have been reviewed developed and matured in order to be flexible, efficient and to deliver product-quality at a lowest cost. Thinking of the software industry he claims that many companies will follow a similar strategy as the manufacturers did. They will address the current challenges by providing modern-day software-factory approach to enterprise software delivery. Thus focusing on collaboration, maximize automation and monitoring of intuitive real-time metrics.

The software industry is adapting its practices in order to keep up with the demands of the market. As a consequence processes are analyzed and principles and practices are revised which results in new approaches like Continuous Delivery.

## 3.2. Principles of Continuous Delivery

Continuous Delivery was introduced to keep up with the situation in the industry mentioned by Brown. The discipline of Continuous Delivery emerged as the result of software engineers and companies heading towards the modern-day software-

factory approach. According to Martin Fowler, a contributor to the Agile Manifesto and one of the founders of the agile movement, Continuous Delivery is: a software development discipline where the software, throughout its lifecycle, is in a state which allows it to be deployed (delivered) to the customer at any time. Therefore to implement Continuous Delivery certain practices, which will be described in a later paragraph, need to be maintained and repeated throughout the whole software lifecycle (Fowler, 2013). From the stage of designing the software to releasing it to the customer Continuous Delivery needs to be considered. Because Continuous Delivery addresses matters that span over different departments and distinct expertise starting from product management, development, quality assurance, operations and so on. Those departments are part of the value chain of the software product and each of them needs to communicate and collaborate with each other to deliver value to the customer. (Humble, et al., 2011) Continuous Delivery emphasizes a holistic view on all the tasks and stakeholders of the value chain in order to optimize cycle times. In regards to Continuous Delivery Jez Humble and David Farley describe the following principles of software delivery:

**Reliable and Repeatable Software Release process**

This principle summarizes the bottom line or the big objective of Continuous Delivery. Releasing software should be as easy as pushing a button. Continuous Delivery aims to create an underlying process which is designed and tested hundreds of times in order to allow push-button releases. For creating a robust, repeatable and reliable release process Continuous Delivery promotes two fundamental principles: *automate almost everything* and *put everything required for build, deploy, test, release of the software application in version control*.

Deploying software is one of the most stressful parts as there is a risk if it goes wrong it might take a lot of working hours until a running state can be recovered. A deployment requires delivery teams to perform the following tasks:

- Provide the environment in which the application will run ( hardware configuration, software, external services etc.)
- Install the correct version of the software application
- Configuration of the application, including any data or state it requires

Following the Continuous Delivery approach all these steps should be easy by adhering to the fundamental principles of automation and version control. Which means version control provides any artifact that is required for example scripts which are fully automated. Those scripts deploy the whole application from version control. The same needs to be done for configuration of the software. Providing the hardware environment seems to be tricky to automate at a first thought. But with virtualization software being omnipresent, automating hardware provisioning and configuration becomes a relatively simple task. Automating the deployment is a huge contributor to ensuring a reliable and repeatable release process. (Humble, et al., 2011)

**Automate Almost Everything**

Similar to the deployment of the software other areas should be automated too. This includes the build process up to the point where human direction or decision making is necessary. Usually the build process includes different stages of testing which are generally automated. Database upgrades and downgrades should be part of the deployment and therefore also part of the automation effort. As already mentioned database change deployments tend to be more problematic this will be elaborated in a later chapter. In a release process there are probably parts that can hardly be automated. Something like Exploratory testing which relies on experience and excel of testers for example or a demonstration of the software cannot be performed by a computer. Despite a short list of things that cannot be automated everything else should be automated.

The reason why many delivery teams do not automate their release process is because they think that it is an overwhelming task. Compared to that keeping a manual release process seems to be less daunting. This is likely to be true the first few times a delivery team performs a task in the process. But certainly not after the tenth time. Automation is also a prerequisite for the deployment pipeline which is a key concept of Continuous Delivery and will be elaborated later. Only through automation there is a guarantee that people get what they need at the push of a button. However there is no need to automate everything at once. The best way to start is to automate bottlenecks in build, test, deploy and release process. (Humble, et al., 2011)

**Keep everything in Version Control**

The second fundamental principle of CD is to keep everything required for build, test, deploy, configure, release of the software application in version control. This includes all the source code, test scripts, automated test cases, configuration scripts, deployment scripts, database creation, upgrade and downgrade scripts, application initialization, libraries, documentation and so on. All this should be version-controlled and any build should be identified by a build number or revision number that references every piece.

It should be possible for new team members to check out the revision and build and deploy the application to their local environments using only a single command (or push of a button). Every application deployed to any environment should be identified by a build number and it should be possible to tell which of the versions in the version control system application came from. (Humble, et al., 2011)

These are the two fundamental principles of Continuous Delivery the following principles are guidelines for maturing the quality in the delivery process.

**If it Hurts, Do it More Frequently, and Bring the Pain Forward.**

This principle is a general one that has an important message. Anything that causes stress (or hurts) because it is error prone or takes a lot of time should be performed more frequently. This could be the deployment step in the release process or a transition between commit stage to testing environment. By performing it more frequently the task becomes more commonly known and can be improved / automated. Once a hurtful part is "cured" (automated) the follow up hurtful step can be approached. For example if the release process is painful following this principle the process should be executed as often as possible. With every improvement on the release process it will hurt less and gradually approach the ideal of a reliable, robust and repeatable process. (Humble, et al., 2011)

**Build Quality In**

This principle highlights one of the most emphasized attributes of CD. As already mentioned in the introduction in classic development and delivery approaches errors or bugs that are detected in production are more expensive to fix than a bug detected in an early development phase. *Build Quality In* refers to many techniques recommended in Continuous Delivery like CI, automated testing etc. that help to catch bugs as early as possible and also fix them as fast as possible. Teams have to be disciplined in fixing bugs. Besides that following *Build Quality In* testing should not be seen as a single phase after development. Testing is performed at any stage for any part of the software product especially as every member of the delivery team is responsible for maintaining tests and quality of the process. (Humble, et al., 2011)

**Done Means Released**

This principle should help to define what it means if a story or feature is "done". Often it is not clear when a feature is really done and what needs to happen to declare a feature done. For CD a feature is done when it is deployed. This is the only way the feature truly fulfills its purpose which is to bring value to the user. Once it fulfills its purpose it can be considered done. This is an ideal case but very often it takes an additional amount of time after implementation until a real user put hands on and gain value from the feature. This is why in Continuous Delivery the feature is done when it was successfully showcased and it was released in a production-like environment. Declaring a feature done only under these circumstances implies that not only one person in the delivery team can be responsible for the delivery of the feature. Different members in the team- developers, testers, operations personnel- need to work together to get something done. Which in return means that the whole team is responsible for delivering. (Humble, et al., 2011)

**Everybody Is Responsible for the Delivery Process**

Collaboration is the reason why organizations exist. Through collaboration organizations can accomplish goals, faster, more effective and more efficient. Collaboration is also a key in Continuous Delivery. The worst situation is to have

separated teams like developers, testers and operations personnel working in their silos only caring about their goals. By emphasizing teams where everybody is responsible for delivery and succeeding and failing always happens as a team, those barriers should be eliminated. The goal is to have continuous communication between different parts of the delivery team in order to help each other and to optimize the delivery process. Elevating the communication in the delivery team and empowering people to allow them to work together efficiently is part of the philosophy of the DevOps movement which was already described. DevOps is often referred to a prerequisite of Continuous Delivery because the mutual understanding and consistent communication between functions is required to implement an automated release process. (Humble, et al., 2011)

**Continuous Improvement**

Releasing software successfully for the first time is only the first stage of many in its lifecycle. The software will evolve over time as new features are added and more releases follow. The same way the software evolves the delivery process should evolve with it. Thus it is necessary that the whole delivery team gathers regularly and discusses aspects of the delivery process. It should be a discussion about tasks that hurt and which need improvement and about finding ideas for innovation. This should follow the well-known Deming cycle: plan, do, check, act (Moen, et al., 2011). Furthermore at this point the previous principle of collaboration is of the essence. If there is no exchange between different departments it will lead to local optimization and incompatibilities. (Humble, et al., 2011)

## 3.3. Benefits of Continuous Delivery

As stated above there is a shift in the software industry and Continuous Delivery is an opportunity for companies to compete. For certain companies it is very much required because it enables organizations to increase throughput, innovation and stability at the same time. Continuous Delivery highlights opportunities for both improvements and excellence. The following list shows the major benefits of Continuous Delivery:

- **Reduced Risk**: Frequent and early releases allow to monitor the progress of a project at any time. This allows organizations to assess if things needed required are done right. Furthermore problems are detected faster while they are still cheap to fix (Minduel, et al., 2014). Big releases are connected to big costs and big consequences. Keeping products in a release-ready state reduces cost of delivery. (CloudBees, Inc, 2015)
- **Reduced Waste**: Manual steps in a delivery process are prone to errors and time inefficiency. Automating process steps reduces the probability of errors caused by manual setup, deployment or testing (Minduel, et al., 2014)
- **Increased Quality:** Rehearsing the releasing process continuously forces to raise the quality bar and increase automated testing. Better quality implies

happier customers, lower costs and less unplanned work. (CloudBees, Inc, 2015) Fast feedback gathered from frequent releases of the product can be used for improvements immediately. This increases the quality of the process, reduces defects in the product and exposes what really matters to the user. (Minduel, et al., 2014)

- **Increased resilience:** For Continuous Delivery having a fallback or recovery strategy is an integral part. This should ensure that if a deployment to production fails there is always a way to restore a previous state. This fallback strategies are automated and tested the same way as the deployment in order to avoid surprises when they are needed. Additionally small changes make fallbacks less complex. (Minduel, et al., 2014)
- **Increased responsiveness:** Long lead times for changes result in delayed reactions to events. With Continuous Delivery lead times of changes are short, thus faster reactions to unpredicted events and changed circumstances is possible. Again smaller changes allow to faster identify reasons for a problem. (Minduel, et al., 2014)
- **Increased innovation:** Having the ability to deliver and react fast allows to explore the market, to measure the value, to analyze user reactions to the product, to assess the fitness of a product for its purpose and discover new business opportunities. (CloudBees, Inc, 2015) (Minduel, et al., 2014)

Taking the opportunities of Continuous Delivery to improve and approach excellence has a reported increase in organizations revenue. A recent survey on Continuous Delivery and DevOps in enterprises reports that 87 percent of organizations with development and operation functions that are rated “excellent” saw a revenue growth of more than 10 percent. Only 13 perent of organizations with development and operation functions rated average or worse saw similar growth. (Minduel, et al., 2014)

These benefits show the opportunities of implementing a software factory. The principles form the framework and guidelines for building a software factory that follows Continuous Delivery. For an implementation in practice delivery teams need to keep this principles in mind and design a *deployment pipeline*.

# 4. The Deployment Pipeline

A software factory requires a sophisticated approach to provide software efficiently and of value for the customer. Software delivery is an interdisciplinary activity that needs different functions working together. Similar to an automotive factory for example producing a car requires different departments and stages like design, assembly and quality testing stages which will be passed until a new car is ready to be delivered to a customer. The same is true for a software product that has to go through different stages before it is delivered. Generally the stages include design, development, automated testing, manual testing and production. Thus if those process steps are performed by different departments that do not collaborate effectively the process as a whole is prone to waste caused by miscommunication and inefficiency. The term “waste” comes from lean manufacturing and is simply the opposite of value. Eliminating waste is one of the principles of agile development which wants to make sure that there is only value adding activities (Bandaru, 2013). Such waste will lead to software that takes too long to get into production-like environment and is buggy because the feedback cycle between developers, testers and operations teams is not of good quality or takes too long as well.

For example software developers commonly have a CI process at hand. They can ensure that the software developed is consistent. As mentioned before CI is best practice and helps developers to ensure code quality and a working code base (Beck, 1999). Furthermore CI establishes a fast feedback cycle for developers as they will be notified what part of the code does not compile and breaks the build, or what new changes do not pass the defined unit and acceptance tests. CI mainly focuses on development teams and the output of the CI process will be used as input for manual testing and the rest of the release process. Having a silo-style delivery approach nurtures an opportunity of waste. The build needs to go through the functions in order to be released. Thus having an end-to-end release approach allows potential bottlenecks - which can be found at transitions between functions - to be easier identified and eliminated. Such bottlenecks are for instance: Testers that are not notified on time when there are new builds available and therefore wasting time testing old versions. Operators that are not informed which build has passed the manual testing process. Those bottlenecks may stall the release process and the feedback cycle becomes slow. Therefore a good delivery process is designed based on an end-to-end perspective which involves every part of the value chain and thus paves the way for optimization. (Humble, et al., 2011)

Today developer, testers and operators need to work together as one team working on accomplishing a goal not a task. Having cross functional teams at hand speeds up the development of production-ready code and test the code in production-like test environments. Following the DevOps principles will certainly improve the release process as the information between the peers of the delivery team will gain on quality and the mutual understanding of issues will allow faster feedback and more efficient collaboration. (Hüttermann, 2012) Even though communication and collaboration is

important to utilize the full potential the release process also needs to be viewed from an end-to-end perspective of delivering software. The goal is to automate the transitions between the teams and use a smart information system that makes the state of the release process transparent to the team. The resulting process should empower the delivery team to share artifacts and information and thus reducing potential waste between departments. Such a process is often called a *deployment pipeline*. A deployment pipeline should form an end-to-end automated build, deploy, test and release process that provides developers, testers and operators with the assets and information (feedback) they need at the time they really need or demand it. Such a deployment pipeline allows to perform the same steps more often unveiling bottlenecks and problems – waste that needs to be eliminated. Ultimately by consistently utilizing and improving the deployment pipeline and thus the process steps, the whole release process becomes faster and safer. (Humble, et al., 2011)

## 4.1. What is a Deployment Pipeline?

Martin Fowler (Fowler, 2013) describes the concept of the deployment pipeline on his website as a way to perform a breakup of the release process into stages. The early stages are fast and automated, and with each stage more processing time will be required and each stage will add confidence to releasing the software. Early stages will provide faster feedback and the feedback will get less and slower in the later stages because they require manual interaction and probing.

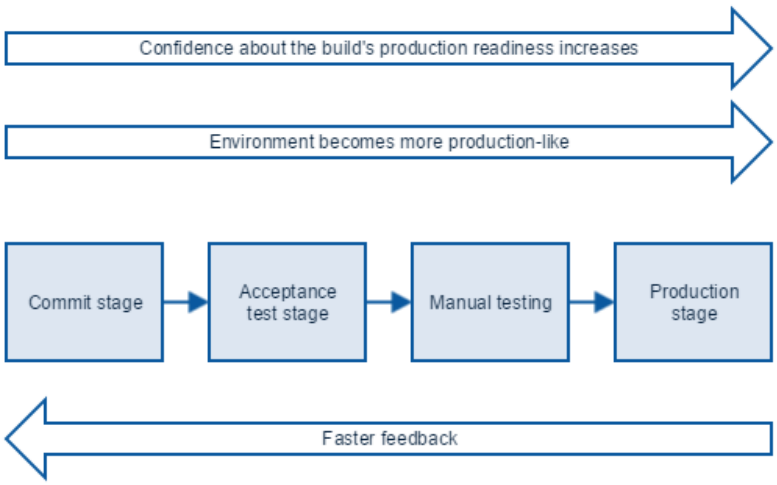


**Figure 4.1 - Deployment pipeline trade-offs (Humble, et al., 2011)**

The deployment pipeline is an abstraction of how the software will get from the version control system to the customer. It describes different stages a delivery team needs to pass until they feel confident enough to release the software. The input for the process is a build that is triggered by a committed change and each change will trigger a new build. Each build will go through the stages in the deployment pipeline passing different tests in order to verify that it is a valid release candidate. With each

stage the tests will become more production-like which requires more resources and more time to go through. Approaching a production-like environment increases the confidence to release with each stage. The objective is to find errors as early in the process as possible in order to eliminate builds which are not releasable and to provide feedback to the team as rapid as possibly. Consequently if the build fails a stage it will not be promoted to the next stage. For each stage a delivery team has to define when a build is marked as failed. Reasons could be failing a test, or not aligning to code or performance metrics. Applying this pattern for every build has some important benefits (Humble, et al., 2011):

1. Only builds that have proven to be fit for their intended purpose are released to production
2. If the release process is automated it is rapid, repeatable and reliable. This makes release into production a “normal” event which allows to release more frequently and that increases valuable feedback from customer (Humble, et al., 2011)

In order to achieve this desirable state there needs to be a suite of automated tests that ensures that the release candidate is ready for production. Furthermore the deployment between development, testing and production need to be automated. Transitions between environments or functions are often sources of waste and therefore prone to errors if they are executed manually.

Since there is no such thing as the one true implementation of a deployment pipeline, different implementations can have different stages but the following stages describe a common subset which can be seen as a starting point for conception:

- The *commit stage* is the initial stage which ensures that the system works at technical level. It compiles, passes a suite of automated tests and performs code analysis
- *Automated acceptance tests*: At this stage the system is asserted on a functional and non-functional level. It needs to make sure that it provides behavior that meets the requirements of the user and fits to the specification of the customer
- *Manual testing*: At this stage tests need to be passed that assure that the system is usable and fulfills its requirements. Additionally any defect that passed automated tests need to be detected here. Furthermore the value for the user needs to be verified. This stage usually includes user acceptance tests and capacity tests.
- *Release stage*: Prepare the build for delivery to the customer by packaging it or by deploying it to production or staging environment. A staging environment usually is identically to the production environment. Nevertheless it depends on how much control a delivery team has over the production environment.

An automated software delivery does not exclude human interaction with the system. It only makes sure that the error prone and complex steps that can be automated are automated, reliable and repeatable in execution. (Humble, et al., 2011)

Jez Humble and Davit Farley describe those deployment pipeline stages in their book Continuous Delivery. They used a Continuous Delivery approach and the concept of a deployment pipeline in different projects which allowed them to aggregate those common pipeline stages. Furthermore based on their experienced they described some interesting best practices that should be followed when designing the stages.

## 4.2. Deployment Pipeline Best Practices

As already mentioned this is only a common subset and others may have a different view and specify additional stages in order to create a better model of their real-world e.g. coding as initial stage because the feedback eventually flows to the coding stage (Rümmler, et al., 2014). Nevertheless the set of stages provided by Jez Humble and Devit Farley is reasonable and a entry point for delivery teams as they used this set in various projects. In the following paragraphs each stage and its purpose is described in more detail.

**Commit stage**

The commit stage is the first stage every build has to pass in order to go through the deployment pipeline. Ideally each committed change creates a new instance of the build process but for performance and capacity reasons changes could be collected before a new instance is started. The goal is to provide the developer with feedback as fast as possible. This is achieved by applying practices of continuous integrations at this stage. There are a few activities that should be executed in a reasonable length of time.

- Compile the code
- Run a set of commit tests
- Create binaries for later stages
- Perform code metric analysis
- Prepare artifacts for use in later stages (Humble, et al., 2011)

The commit stage should have the shortest feedback time which means that it is wise to design this stage pursuing very short cycle times. In the commit stage a fast check should be performed that verifies basic qualification of the build by performing a set of commit tests. These tests will include a bunch of unit tests at the beginning but with the evolution of the pipeline some acceptance test which are known to fail very often could be added to provide faster feedback. Additionally a code analysis can be performed identifying duplicated code, measuring test coverage, cyclomatic complexity basically any useful metric that can be computed at this stage. Similar to unit tests the build should fail if a certain value is exceeded. Binaries and artefacts

will be created and provided for later stages if the build passes the commit stage. It is important to note that those binaries and artefacts will only be created once during the whole pipeline cycle time. (Humble, et al., 2011)

In an ideal setting developers should wait until the build goes through all the stages before they continue their work. In most cases this is not practical as tests at subsequent stages will take more time. At the commit stage the cycle time through these steps should take about 5 to 10 minutes which can be achieved by optimizing the test suite and using build grids. At this stage it still makes sense to have developers stop their work until the commit stage is passed before they continue with new tasks. If the build breaks at any stage developers need to fix the build as fast as possible. This requires disciplined teams but is essential in order to ensure releasable candidates. Again it requires a combination of both: short cycle times at any stage in order to provide fast and valuable feedback and disciplined delivery teams that report and fix broken builds as fast as possible. (Humble, et al., 2011)

**Acceptance Test Stage**

The acceptance test stage performs test in order to ensure that the software which is delivered actually provides functions that are required. Furthermore the tests are run in a more production-like environment to increase the confidence of a production-ready build. If the commit stage contains unit tests which are testing the code on a low level, the acceptance test stage includes functional tests with acceptance criteria. Furthermore the acceptance test stage also contains regression tests that make sure that new changes do not introduce bugs in existing behavior. Tests at this stage make sure that certain specifications from customers or other stakeholders across the delivery team are met. Therefore acceptance tests are not created by separate teams instead they are specified and maintained across the delivery team. The acceptance test stage is the second milestone of a release candidate. (Humble, et al., 2011)

At this stage it is important to have a good idea of how the target environment (where the application will run) looks like. As already mentioned the more control over the production environment the easier it is to provide software that works. If there is full control over the production environment it is easy to simulate it and run acceptance tests in this simulation. If the production environment is complex or expensive, a scaled down version needs to be designed for example using just a couple of servers instead of hundreds. If the application requires external services often an implementation of test doubles is a good strategy. (Humble, et al., 2011)

Providing production-like test environments can be tricky because in order to be effective it needs to be provided to every part of the delivery team so everyone can run acceptance tests. This is important because once for example developers cannot perform acceptance tests and therefore cannot fix errors that occurred during builds that failed acceptance test stage, they will stop taking care if a build that fails at this stage at all. As a result broken release candidates will remain broken for a much

longer time. Again cross-functional DevOps teams are required. Similar as the whole team is responsible for the deployment pipeline and the stages included, the whole team is responsible for the acceptance tests. Following this principle there is a need for developers to run acceptance tests on the same production-like environment in order to fix a broken build. Common blockers are licensing of testing software and software architecture that cannot be deployed to test environments which should be avoided. (Humble, et al., 2011)

Additionally Acceptance tests can become tightly coupled to a specific solution in the application. As a result minor changes in the behavior of the application invalidates the tests. This is why acceptance tests should be expressed in a business language and not the language of the technology of the application. That means the abstraction should work on the level of the business behavior for example: “create order” instead of “press create order button”. (Humble, et al., 2011)

**Subsequent testing / user acceptance tests**

Once the release candidate has passed the acceptance test stage it has reached a significant milestone. The release candidate has gone from the stage that is of concern for the development team only to a phase were it is interesting for delivery. For very simple release processes after passing the automated acceptance test stage the release candidate is ready to be delivered to the customer. The stages passed so far contained fully automated tests suits. If the release candidate passes all that tests it automatically gets promoted to the next stage. After the automated testing stages usually there are some manual tests. The release candidate is deployed to different environments for capacity testing, exploratory testing, and staging and production testing. At this stage sometimes automated testing is not efficient and effective anymore and it often requires to have some human interaction for doing some exploratory tests or determine if usability requirements are met. Furthermore testing of nonfunctional requirements like security or capacity could be part of the release process. The deployment pipeline needs to make sure that only candidates are deployed to the logical next stage that passed the previous stages. It is good practices that deployments to different environments again follow the release process flow. That means if usability testing and capacity testing could be done in parallel it should be done but the pipeline needs to have a mechanism to make sure that only those builds can be deployed to production or staging environments that passed all the tests in the user acceptance test stage. (Humble, et al., 2011)

The deployment pipeline needs to provide testers with the possibility to deploy any build to their testing environments. Instead of just getting the “nightly build” with an arbitrary revision, testers should be able to choose any successful build that passed the previous stages. Further more information about the set of changes included in the build needs to be presented so they can choose what they want to test. In case the build is not suitable for their tests because it contains a bug or an important change is not included it should be possible to switch to another build with the push of a button. (Humble, et al., 2011)

**Release to Production**

The release to a production system always includes risks that something goes wrong and the rollout of new capabilities fails or even worse the production system breaks and important information is lost or business units cannot work anymore. In order to be prepared a delivery team can follow these guidelines (Humble, et al., 2011):

- Create and maintain a release plan that includes everybody involved in delivering the software (developers, tester, operations, infrastructure and support personnel).
- Follow the principle of *Automating almost everything* in order to minimize errors from people making mistakes
- Perform the whole procedure on production-like environments frequently, in order to debug the process
- Implement the ability to back out a release if something does not work as planned
- Define a strategy for migrating configuration and production data as part of upgrade and rollback.

The first two guidelines simple describe the DevOps principles of being responsible for the whole process as a team and using automation for empowering people to do things that are of value.

The third guideline emphasizes to repeat the release process as often as possible by rehearsing it in production-like environments. This is a good practice that should reveal problems during release to production but it is hard to establish. As already mentioned the more control over the production environment exists the more automated the release process can be performed. In a best case the production environment is completely locked down and changes are only deployed using an automated process which includes application, configuration, state, network topology, software stack infrastructure. The environment management process should be part of the deployment pipeline and used for managing testing environments. It is a good practice to automate the provisioning and management of environments in order to ensure tests are executed on identical environments. (Humble, et al., 2011)

Having a back-out strategy as the fourth and fifth guideline suggests is a challenging task but with regards to the risks it is something that is very reasonable to have. Rehearsing the release pipeline many times using the automated deployment pipeline should already reduce the risk of a malfunctioning or broken application. Nevertheless a simple strategy is to keep the old version of the application available while deploying the new version. From case to case this could be very simple but also very complex. The most complex problems for back-out strategies are often related to production data and database changes. Barriers and Strategies for continuous delivery of database changes will be discussed in a later chapter. Besides having a copy of the old

version another strategy is to fall back to the latest working build and redeploy this version of the application. (Humble, et al., 2011)

Sometimes it will be hard to provide a good back-out scenario or it will require a lot effort and working hours. At no point should the back-out process be different from the deployment process. The reason for that is that such a process will hardly be tested as it is not part of the normal release process. It is recommended to use the deployment process to either keep an old version of the application deployed or simply redeploy a previous known-good version. (Humble, et al., 2011)

## 4.3. Deployment Pipeline: Implementation Guidelines

The deployment pipeline is the basic tool to execute and improve the release process and align to the principles of Continuous Delivery. As mentioned above the deployment pipeline is an abstraction of the release process which describes how the software comes from the source control system to the customer. The first part of implementing a deployment pipeline is to reflect upon the release process.

**Find and gather the right people**

Modelling the release process cannot be done by a single person but requires the whole delivery team to contribute. Again a principle of Continuous Delivery is to make everybody responsible for releasing software which will only work if the whole delivery team is included in designing the tool they are going to work with. Furthermore different team members will contribute different perspectives and different expertise which need to be considered in order to have a holistic view of the process. A first step is to gather people with clear and hands-on understanding of the tasks in the release process. This group of people should be able to generate a value stream map that highlights value versus waste. (Swartout, 2014)

**Value Stream Mapping**

Value stream mapping is a methodology that comes from the manufacturing industry. It is a technique to illustrate how a process works and break down a process into a series of steps and handover points and offers the lead time as the key metric. Furthermore value stream mapping offers a technique to discover waste in a process. Value stream mapping intents to highlight a value versus waste model which can be analyzed to find where potential bottlenecks and delays occur within the process. In other words the value stream map shows which activities are adding value and which do not. Figure 4.2 exhibits the time spend during certain tasks in a feature development process. The numbers below the tasks indicate the time spend for these tasks whereas the numbers between the tasks indicate waiting time.

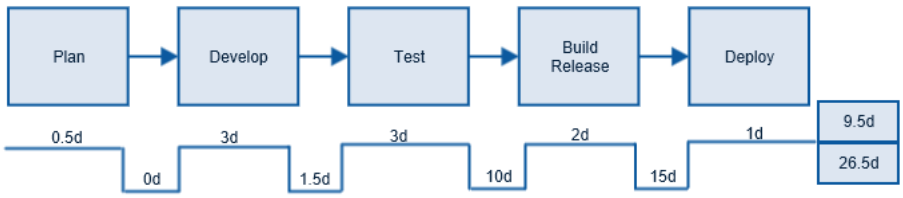


**Figure 4.2 - Value Stream map of a common feature development process (Swartout, 2014)**

In case of the deployment pipeline the value stream map is a valuable input for planning as it represents a common view over the whole process and it shows possible bottlenecks. Having accomplished a common understanding the delivery team can focus on areas where inefficiency was identified and think about solutions. Furthermore it is an instrument to measure and improve by reevaluating the value stream map after changes are implemented. (Swartout, 2014)

### Defining Goals

With the value stream map at hand the delivery team has a tool for communicating the current situation and a starting point for improving the current process. Thus the next reasonable step is to define a goal which should be achieved by following the deployment pipeline concept of Continuous Delivery. At a certain point the IT managers will have to justify to their superior why time and money is spend in implementing Continuous Delivery. Implementing Continuous Delivery and a deployment pipeline is a project on its own and it is important to set goals from a management perspective as well as to generate a common plan that aims to reach certain goals. This plan might include a change in used technologies and changes in process flows and steps. As already mentioned DevOps is often seen as an enabler for Continuous Delivery. A first step would be to establish intra-disciplined teams of developers and operators who have a mutual understanding and follow common goals. Changing the culture might already have a positive effect on the value stream map as communication and empowerment reduces waste in the release process. (Swartout, 2014)

### Find the right tools

Continuous Delivery aims to automate parts of the release process. Delivery teams will need to agree on tools they want to use in order to create the technical building blocks for a sufficient process automation. Figure 4.3 shows a generic architecture of a deployment pipeline including common stages and components as discussed in the previous chapter. Furthermore there are interaction and transitions illustrated. The delivery team has to decide how these stages, components, interactions and transitions should be implemented and have to find solutions for:

- Version Control System
- Automated Build and CI tools
- Artefact Repositories

- Automated Testing Tool
- Deployment Tool
- Provisioning Tool
- Monitoring Tool
- etc.

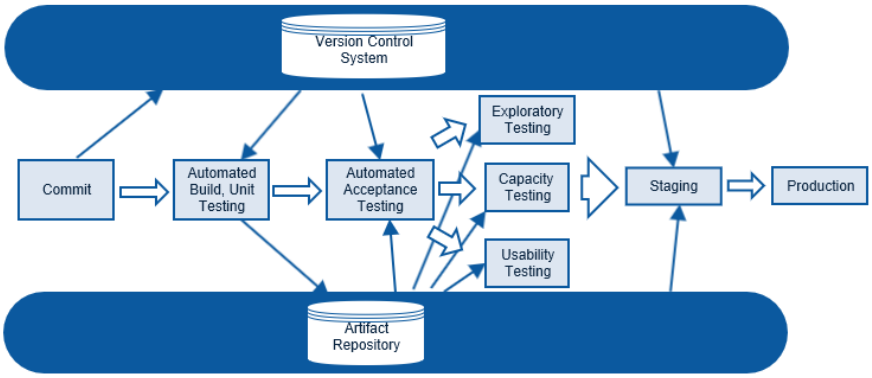


**Figure 4.3 - Architecture of a deployment pipeline (Humble, et al., 2011)**

As Continuous Delivery has become more established the number of tools developed to support Continuous Delivery increased. Sometimes it will not be sufficient to buy a tool and teams will need to create their own scripts, tools, plugins that support their requirements. The whole delivery team is responsible to find the best solution in order to achieve a robust, reliable and repeatable release process. (Swartout, 2014)

### Plan, do, check, act

In certain intervals (after a release, after a customer project) the delivery team will do a review on the deployment pipeline in order to find inefficient areas for improvement. Continuous Delivery follows the principle of continuous improvement which is based on Deming cycle Plan, Do, Check, Act (Moen, et al., 2011). In order to evaluate the process the delivery team needs to define metrics to measure performance in certain parts of the release process. The cycle time of a feature as exhibited in Figure 4.2 is such a metric which can be used to measure performance. Besides performance metrics delivery teams might define metrics for quality, automation etc. depending on where a need for improvement and measurement was identified. (Swartout, 2014) Although metrics are an essential part in DevOps and Continuous Delivery presenting, elaborating and researching on important metrics is not part of this thesis.

# 5. Database Continuous Delivery

At this point the concepts of Continuous Delivery have been thoroughly described and the benefits of it have been presented. One major goal of this thesis is to analyze databases in the context of Continuous Delivery. Many enterprise applications are processing information over a long time and therefore require to persist the information which makes them rely on databases. As described in the chapter 1 Introduction, databases are different than application code and often introduce additional complexity to an automated release process. Database changes are often avoided (Grolinger, et al., 2011) because delivery teams in particular often do not have the right processes and tools at hand to minimize database change impacts (Curino, et al., 2013). Additionally database related tasks in an automated release process become more complex the closer the process gets to the production environment (Gmeiner, et al., 2015). Therefore databases often require resource intensive procedures and sophisticated strategies (Feitelson, et al., 2013). The following chapter deals with release process related database development and deployment problems by first identifying underlying issues, describing practices to tackle the problems and explaining best practices and strategies for databases.

## 5.1. Issues with databases

Databases are important in modern information systems as they enable information sharing between different applications, business processes and users. Additionally having a proper database allows enterprises to better understand their business by utilizing practices like Business Intelligence (Bogza, et al., 2008). Databases are important for businesses and therefore treated carefully. Nevertheless they are part of software that needs to fulfill specifications and is designed, developed and released. Thus having the aspiration to implement an automated release process, databases need to be addressed as well. There are some difference when it comes to databases and the application which is accessing the data. Databases in particular present some challenges for release automation which will be described in the following paragraphs.

**Database schema evolution challenge**

A fundamental part of the database is the schema that defines the structure of the database. Most data oriented techniques are serial in nature hence requiring a detailed definition upfront. Further the schema is often put in change management control to minimize changes. Thus database development techniques do not reflect the realities of modern software development processes (Ambler, et al., 2006). Requirements do evolve as improvements are demanded or new technologies are adopted. When the original specification changes, databases often need modification or refactoring. Due to the adherent possibility of losing valuable operational data operation teams know the risk of changing databases. As a result they try to avoid

certain database changes either by not releasing upgrades or by avoiding database refactoring which causes a degradation of the underlying schema. In contrast to application code where refactoring is common practice and intended to happen, database tables, queries, procedures and views etc. are often problematic to change due to the underlying data (Grolinger, et al., 2011). Additionally a big prebuilt database schema may be useful to generate domain knowledge but building a application around it may come with some tradeoffs (Harriman, et al., 2004). Furthermore as databases may be accessed by different applications a database change may require additional refactoring in all these applications. Due to that a schema where refactoring is avoided or not managed correctly becomes increasingly hard to maintain because of the resulting unstructured data. This also makes the development of applications that rely on the database more challenging. (Grolinger, et al., 2011)

**Database scripting is prone to error**

Databases come with a huge amount of scripting which is required to set up a database. Carrying out changes to these scripts requires a disciplined team. Most of the changes are developed using database objects in a running database which has no enforced connection between the objects and the scripts. This means that there is nothing that ensures good development practices (check-in, check-out etc.). As a consequence not all changes find their way back into the version control system, code overrides are common and out of process updates are unnoticed. Additionally a single change often requires to change several scripts: a script for the actual object, the change and the rollback need to be written. Besides that for most scripts it is mandatory to run in a specific order as they are unaware of changes in the target environment. This may cause already applied production hot fixes to be overridden. Furthermore when several people are working on the same database objects eventually they need to merge their changes. If the version control system does not provide sophisticated functionality it will require manual effort to merge the changes. The management of these scripts and manual scripting is always prone to human error, syntax errors etc. (DBMaestro, 2013)

**Databases are bottlenecks**

Databases are bottlenecks in a development process but there is a good reason for that. There is a friction between development and operations staff as mentioned in chapter 2.3 Definition: DevOps. Database administrators are part of the operations team and hence pursue a strategy of protecting production environment against changes. (Rümmler, et al., 2014) Despite having various tools for database management, monitoring integrity as well as ensuring high availability none of the tools emphasizes speed in database development rather the opposite is true. Which comes from the already mentioned nature of queries, functions, procedures, views etc. where refactoring is less elaborated. (Grolinger, et al., 2011). This is why database administrators (DBAs) like to slow down or avoid changes which grants them a system as stable as possible. The reason why they strive for slow or no

changes is clear: they are aware of the complexity of database changes and the business value of data and application availability. Developers follow the opposite. They want to get their changes to production as fast as possible to gain feedback and do so by working with methods that focus on speed and quality like iterative development processes, CI processes, automated tests etc. Thus database development is a bottleneck in the software lifecycle especially if teams are not integrated and there is no good communication. (Rümmler, et al., 2014)

Besides that databases are bottlenecks for another reason and that is not process or organization related, it is simply because of the data itself. During initial release databases are similar to application code and it would be simple to deploy database changes just like code by dropping the existing database and replacing it with a new one. But once the initial deployment is passed it is mandatory to ensure that there is no loss of data. At least the deployment strategy has to guarantee that there is no loss of valuable business data. This on the other hand slows down the deployment process as more data will require longer to refactor and back up. (Fritchey, 2014)

The issues above illustrate the challenges coming along with databases. Most of these challenges relate to database practices which are not conform to agile practices. In order to cope with these challenges, reduce risks and emphasize database changes a more automated and agile approach needs to be applied. Having the principles of Continuous Delivery and CI in mind, database practices need to align to the following goals:

- Integrate database in CI process
- Code and Database synchronization
- Automated and repeatable database setup and refactoring
- Back out strategies for database change deployments

The following sections describes results of the literature research on database development and deployment practices and strategies.

## 5.2. Practices for integrating databases in Continuous Delivery

Continuous Delivery describes the deployment pipeline as a necessary tool in order to achieve a reliable, robust and repeatable release process. Thus for databases and database changes a developing and deployment framework needs to be introduced that empowers a delivery team to integrate it into the deployment pipeline. However in the previous section database related issues were presented which make this integration problematic. These issues require database related tasks to become more agile and more automated. Therefore a research on database practices was conducted in order to find practices to overcome these issues. The following practices were found:

**DevOps: DBAs collaborate closely with developers**

A fundamental premise for Continuous Delivery are teams that are working together as described in 3.2 Principles of Continuous Delivery. For databases in particular it requires database administrators (DBAs) to be approachable and available. Every task a developer works on may require a significant change of the database schema. If this is the case it should be easy for the developer to consult a DBA. The developer knows the specification of the new function and the database administrator has the global view on the data in the application thus they need to work together to find a solution. (Fowler, et al., 2003)

**Use Version Control Systems to share Database Assets**

All software assets need to be version controlled and this also includes database assets.

Such assets include:

- DDL to drop and create all database objects.
- Stored procedures and functions
- ERD diagrams
- Test data DML scripts
- Specific Database configuration

In order to align to Continuous Delivery principles having the database fully available in the version control system for automated deployments is mandatory. This helps in various development, test and deployment scenarios and relieves DBAs from manually setting up databases. Developers for example are able to fetch the latest version, do their changes and run them on their own development environment before committing. Having database under version control also allows to integrate it in a CI build and in automated acceptance tests and eventually for any stage in the deployment pipeline. Database set up no longer requires a DBA which reduces work load. Additionally for projects a distinct branch in the version control system could be reasonable where only project specific changes are committed. DBAs need to be consulted for bigger changes and for merging back developments from a branch to the master. (Duvall, et al., 2007)

Nevertheless maintaining database scripts can be error prone. Therefore version control of database scripts requires to think about a meaningful directory structure in the repository. This should clearly describe where a script belongs and thus reduces errors. (Duvall, et al., 2007)

**Database Continuous Integration**

Databases need to be included in the CI process. Even though it is best practice to back application code with automated build and automated tests in the CI process,

databases and their internal behavior are often overlooked. Just building the database using the same scripts can help to reduce errors in database deployment. Having a suite of tests for internal behavior would even increase the quality. Integration database development in CI as described in chapter 2.1 Definition: Continuous Integration (CI) should also ensure smaller database changes and thus better feedback and a more stable system. (Red Gate tools, 2013)

**Encapsulate database access**

The more the database access is encapsulated in a particular section of the source code or using procedures and views in the database itself the easier it is to perform database schema changes. It is good practices to use DataAccessObjects (DAO) that obtain shared methods like save(), retrieve(), find(), delete() and implement the data access separately from business classes. Even if it is necessary to have hardcoded SQL (e.g. query optimization) all the database access related files should be kept together in one place in order to find them easier. Encapsulating databases should help to abstract the database layer and make it exchange able thus making database changes less difficult. (Ambler, et al., 2006)

**A database consists of schema and test data**

A database consists not only of its schema definition but what is the most interesting for application development is the data it stores. This data includes domain keys and sample test data for example dummy master data.

Providing test data in the database has several reasons. First it enables testing. An automated test suite helps to stabilize the development of an application. It makes sense to work on a database seeded with some sample test data, which all tests can assume in place before they run. Having sample data in the database also helps to test the migration of data after a change to the schema of the database. Sample data often consists of fictional values, which is quite understandable if a project is in a very early state. If real data is available this is the preferable choice. (Sadalage, 2007)

**Consistent change management**

When DBAs and developers closely work together it should be easy to synchronize application changes that are dependent of some database changes and package them together in one database refactoring.

Such a database refactoring then addresses these three different aspects:

- Changing the database schema
- Migrating the data in the database
- Changing the database access code

Whenever a database refactoring is described it has to include all three aspects and make sure that all of them are applied before the next refactoring can be executed.

Many changes to the database do not interfere with applications accessing the database. Like adding a column to a table. If the application queries the table without knowing about the column it will not break. There are many changes on the other hand which do have effects. As described in 2.2 Definition: Database Change there are different types of database refactoring and they need to be handled carefully. (Ambler, et al., 2006)

**Automation: Refactoring**

Refactoring can and should be automated using tools. Every database refactoring is written in its own script. These scripts are never executed manually instead there is a small script or a tool which executes all refactorings and applies them to the database. Once done, these script files can be used to produce a change log of all alterations applied to the database as a result of database refactoring. This allows to update any database to the latest version by running all scripts required to update from a certain version to the latest version. The ability to sequence automated changes is essential for integrating databases in the deployment pipeline. A reliable and robust release process also provides back out strategies. Similar to creating scripts to update a database to a certain version there should be scripts to revert a database to a previous version. (Sadalage, 2007)

**Automation: Database Integration**

Additionalyl to automate refactorings it is recommended to create scripts for setting up a database which is required for integrating the database in CI process and eventually in the deployment pipeline. Making the database deployable by any team member empowers teams to perform valuable tasks and reduces cycle time as described in 4.2 Deployment Pipeline Best Practices. The following table shows some tasks which should be scripted in order to automate database integration. (Duvall, et al., 2007)

| | |
|---|---|
| **Drop database** | Drop the database and remove the associated data so that the database can be recreated with the same name. |
| **Create database** | Create a new database using Data Definition Language (DDL). |
| **Insert system data** | Insert any initial data (e.g., lookup tables) that the system is expected to contain when delivered |
| **Insert test data** | Insert test data into multiple testing instances |
| **Migrate database and data** | Provide database migration scripts that migrate database schema and data from one version to another one |

| | |
|---|---|
| **Set up database instances in multiple environments** | Create different database setups for automated test, user acceptance and capacity tests and for production |
| **Modify column attributes and constraints** | Modify table column attributes and constraints based on requirements and refactoring. |
| **Modify test data** | Alter test data as required for multiple environments |
| **Modify stored procedures (along with functions and triggers)** | Modify and test stored procedures, functions and triggers in order to ensure database behavior |
| **Obtain access to different environments** | Test required database users by logging in with different IDs and password. Test access privileges to database objects |
| **Back up/restore large data sets** | Create specialized functions to backup data for especially large data sets or entire databases |

**Table 5.1 - Database tasks that should be automated (Duvall, et al., 2007)**

**Test Every Change to the Database Design**

Having an automated tool that builds every database in the same way enables to set up a database at any version which allows to do transition testing. Transition tests is a method for protecting data and knowledge already existing in the database. Transition tests create a database of a specific version filled with representative data. Then an update is performed to the next version. During the assertion phase the data in the database is asserted to ensure it is in an expected state. This is a good practice to ensure that database changes to not interfere with a target environment.

Additionally to that a complete automated test suite for internal behavior is also recommended. This automated tests help to verify that changes on the database did not harm the expected internal behavior. Furthermore with automated testing of the database design the expected behavior of the database from an applications point of view can be verified. (Guernsey, 2013)

## 5.3. Strategies and Best Practice for Database Delivery

Providing a more agile and automated approach for database development and deployment is a crucial step towards an automated release process. The more automated the certain parts of the database release process are the more reliable, robust and repeatable it becomes. Once a certain level automation could be reached databases can be included in the stages of the deployment pipeline. By continuously performing database deployments at different stages deploying the database will

become less problematic in general. Of course the most problematic part is the deployment to real production systems. Thinking about the business value of data, migrating data from an existing system can become very complex and needs accuracy and caution. In order to cope with difficulties during those deployments there are best practices and strategies for strengthening the release process and securing database assets and business data. The following strategies and best practices were found for database change delivery.

**Maintain a database version table**

A very effective practice to support and strengthen automated database migration is to version the database. As Figure 5.1 illustrates a versioning table is a table in the database that records every change to the database and stores a version number and additional information (e.g. a comment, checksum, executioner information). As mentioned in the previous section any change to the database requires at least two scripts one that introduces the change (from version x to x + 1) and one that reverts the change (from version x+1 to x). In the versioning table these changes are recorded and each refactoring has to make sure to update the table after it was executed.

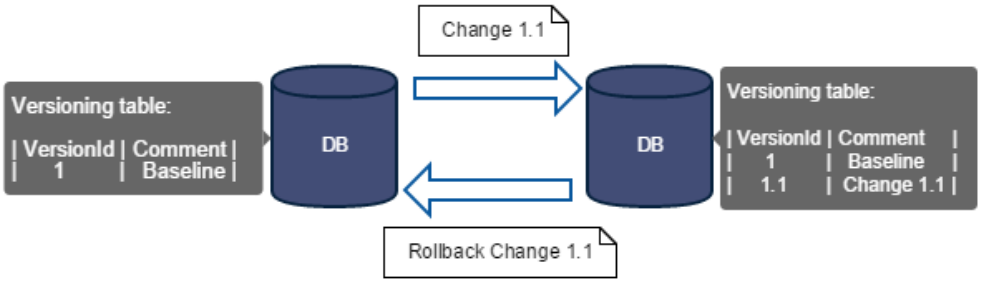


**Figure 5.1 - Database versioning table before and after changes are applied (own illustration)**

From the application point of view there needs to be a configuration setting that identifies which database version the application requires. At deployment time a tool or script does a check on the required database version and applies any changes required in order to migrate the database from the current version to the target version. Managing the database version in this way allows to continuously deploy the application without worrying about the current state of the database in the target environment. Of course this requires that no manual changes were made and deployments are done using the same automated process. Furthermore this kind of versioning decouples the database development from application development to some extent because database changes are not deployed until the application code requires them. (Ambler, et al., 2006)

**Transaction caching**

A versioning of the database allows to create a very robust and reliable database deployment process. One strategy during deployment to production is to cache any transaction that happens during the upgrade of the database. This should allow to keep the system running and enables transaction data restoring after the deployment

or the rollback. This can either be achieved by storing messages that are send between components or simply by copying each database transactions from the transaction log. (Humble, et al., 2011)

**Blue-Green deployments**

With blue-green deployments the current and the new version of the application run side-by-side. The current version in the blue environment and the new version in the green environment. Releasing simply means to switch requests from the blue to the green environment and rolling back means to switch back. This is a costly but safe strategy as data migration can be rehearsed and performed before and after the switch additionally no data gets lost. (Humble, et al., 2011) A company that uses this deployment strategy successfully is Facebook (Feitelson, et al., 2013). During a database change deployment two systems run in parallel copying data from old to new system while also writing new data in both systems. This strategy is used in combination with the following strategy.

**Decoupling Application and Database**

This strategy makes use of the previous mentioned decoupling effect of database migration and application deployment using database versioning and database layer abstraction. If the application can be deployed without necessarily deploying a new database change it allows to try out some application changes before migrating the database. In case rolling back of changes needs to be avoided at any cost this can help to try out the new application features before migrating the database. This requires the application to work with the current and the new version. Once the application change was deployed and does not need to be rolled back the database change can be deployed. (Humble, et al., 2011) (Feitelson, et al., 2013)

# 6. Database automation tools

In the previous chapter database practices and strategies were described that enable an agile and automated approach to eventually integrate the database into a Continuous Delivery environment. This requires to a large extend disciplined teams and an intelligent utilization of tools. As described in the chapter 4.3 Deployment Pipeline: Implementation Guidelines an important activity for delivery teams is to find tools that support them in doing valuable tasks. In this chapter a research on tools for automating database integration and migration is described. First the results of an online research for database migration tools is described. Afterwards a set of criteria is defined which reflects on the functionality expected from such a tool. These criteria will be used to assess the tools.

## 6.1. Database migration scenarios

Before starting the research on database deployments tools two basic deployment scenarios were defined based on the Continuous Delivery literature and database problems described above. These scenarios should help to choose criteria and evaluate tools.

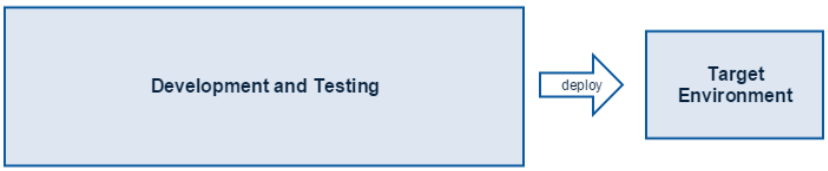


**Figure 6.1 – Scenario 1: Target environment in known state (own illustration)**

The first scenario as illustrated in Figure 6.1 describes a database release where the target environment is in a known state. In this scenario the delivery team controls the target environment. Thus the deployment of a database change can be entirely tested. This makes it easier as the delivery team does not have to create custom deployment scripts for every different target.

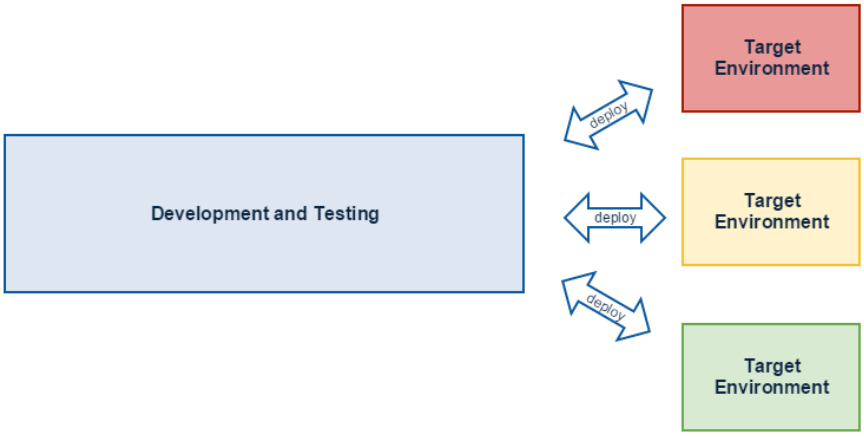


**Figure 6.2 – Scenario 2: Target environment in unknown state (own illustration)**

Figure 6.2 describes the second scenario: A database release where the target environment is in an unknown. In this scenario the delivery team shares control over the environment. Thus each deployment to a different target might require adjustments before the actual change can be deployed. In such a scenario the delivery team would need to compare the target database schema with their own test schema and figure out differences. Afterwards those differences need to be solved and then the changes can be deployed.

## 6.2. Research on database migration tools

In a first step an online research was conducted to find tools which provide functionalities for database migration and additionally are designed to be used in an automated environment.

**RED GATE TOOLS – Database Lifecycle Management (DLM) for Oracle**

Red Gate Tools DLM is a suite of tools which work together to enable CI of database changes. DLM is a proprietary software which needs to be licensed per installation. Red Gate Tools offers its products for SQL Server and Oracle. Table 5.1 shows a feature overview of DLM:

| *Category* | *Features* |
|---|---|
| ***Database Versioning*** | • Commit schema and data changes to any version control system<br>• Inspect database version history and access specific revisions<br>• Store custom migration scripts in source control |
| ***Database Deployment*** | • Create a database from source files in version control |

| | |
|---|---|
| | • Generate schema and data deployment scripts<br>• Validate that two databases are identical<br>• Generate pre-/post-deployment reports for troubleshooting |
| ***Automation*** | • Command line tool<br>• Support of TeamCity, Octopus Deploy, Bamboo Jenkins |
| ***Supported Databases*** | • SQL Server<br>• Oracle Database |
| **Costs** | • 1595 $:<br>  ○ DLM Automation Suite<br>  ○ 1 license = 1 installation<br>  ○ 1 year support & upgrades |

**Table 6.1 - Redgate tools - DLM feature overview (Red Gate tools, 2013)**

**FLYWAYDB**

Flyway is an open-source database migration tool. It strongly favors simplicity and convention over configuration. Flyway provides some helpful features for database automation:

| Category | Features |
|---|---|
| ***Database Versioning*** | • Flyway tracks changes on a database using its own metadata stored in a special metadata table which is created when flyway runs the first time.<br>• Review which changes have been applied and by whom |
| ***Database Deployment*** | • Deploy a database version from scratch<br>• Support migration from one database version to a newer one |
| ***Automation*** | • Executable via command line<br>• ANT, Maven, Gradle, API |
| ***Supported Databases*** | • Oracle Database<br>• SQL Server<br>• DB2<br>• MySQL<br>• Postgres |

| | |
|---|---|
| | • SQLite |
| ***Costs*** | • Open Source |

**Table 6.2 - Flyway feature overview (Flyway, 2015)**

### LIQUIBASE

Liquibase is an open source database-independent library for tracking, managing and applying database schema changes. It was started in 2006 to allow easier tracking of database changes, especially in an agile software development environment. Liquibase offers the following feature:

| Category | Features |
|---|---|
| ***Database Versioning*** | • All database changes are stored in XML and identified by id, author and filename.<br>• A list of all applied database changes is stored in each database to determine what new changes need to be applied |
| ***Database Deployment*** | • Create database from an existing version<br>• Apply updates to current version<br>• Rollback database changes |
| ***Database Documentation*** | • Database change documentation generation |
| ***Automation*** | • Executable via command line<br>• ANT, Maven |
| ***Supported Databases*** | • SQL Server<br>• Oracle Database<br>• MySQL<br>• PostgreSQL<br>• IBM DB2 |
| ***Costs*** | • Opensource |

**Table 6.3 - Liquibase feature overview (Liquibase, 2015)**

### Datical DB

Datical is both the largest contributor to the Liquibase project and the developer of Datical DB a commercial product which provides the core Liquibase functionality plus additional features to remove complexity, simplify deployment and bridge the gap between development and operations. Datical DB was created to satisfy the database schema management requirements of large enterprises as they move from CI to

Continuous Delivery. Additional to the features of Liquibase, Datical has added these enterprise relevant features:

| Category | Feature |
|---|---|
| **Database Deployment** | • Change Forecasting: Forecast upcoming changes to be executed before they are run to determine how those changes will impact your data.<br>• Rules Engine to enforce Corporate Standards and Policies.<br>• Supports database Stored Logic: functions, stored procedures, packages, table spaces, triggers, sequences, user defined types, synonyms, etc.<br>• Compare Databases enables to compare two database schemas to identify change and easily move it to change log.<br>• Change Set Wizard to easily define and capture database changes in a database neutral manner.<br>• Deployment Plan Wizard for modeling and managing logical deployment workflow |
| **Automation** | • Plug-ins to Jenkins, Bamboo, UrbanCode, CA Release Automation (Nolio), Serena Release Automation, BMC Bladelogic, Puppet, Chef,as well all popular source control systems like SVN, Git, TFS, CVS, etc.<br>• Command line |
| **Costs** | • Not available |

**Table 6.4 - Datical additional features overview (Datical, 2013)**

### DBMAESTRO TEAMWORK

DBmaestro TeamWork is a DevOps for Database solution that enables Agile Database Development, CI and Continuous Delivery for the database. DBMaestro promotes the following features of its solution:

| Category | Features |
|---|---|
| ***Development Process Management*** | • Enable change management of database structure, code and content<br>• Check-in/Check-out mechanism for preventing unsynchronized changes and collisions<br>• Prevent accidental and undocumented changes to database objects and content<br>• Link database changes to tasks or business requirements imported from SCM system |
| ***Database Deployment*** | • Enable generation of database baseline from the version repository<br>• Support branch & merge processes for database schemas<br>• Undo updates to structure, code and content of your database<br>• Analyze differences between different versions of your database objects<br>• Analyze the impact of database version changes<br>• Security and audit Trail |
| ***Automation*** | • Command line and WS APIs for integration with your automation process – continuous integration, automatic builds, automatic back-ups, etc. |
| ***Supported Databases*** | • SQL Server<br>• Oracle Database |
| ***Costs*** | • Not available |

**Table 6.5 – Feature overview DbMaestro Teamwork (DbMaestro, 2015)**

## 6.3. Tool criteria

The criteria described in this section reflect on 5.2 Practices for integrating databases in Continuous Delivery and the database deployment scenarios described above. Additionally common criteria used to assess and compare tools like usability, monitoring, extensibility are added. Table 6.6 shows the elaborated criteria, a short description and how points are awarded.

| Criterion | Description |
|---|---|
| **Automate Database Integration** | The most important capability of a tool for testing and deployment scenarios is to set up a database , execute schema changes to a target database, insert test data.<br><br>*2 Points*: Executing scripts to set up a database schema<br><br>*1 Point*: Control mechanism before or after executing scripts<br><br>*2 Points*: Use simple SQL files |
| **Automate Database Migration** | A time consuming and error prone part of database migration is writing migration scripts. The tool should support the delivery team by generating migration scripts for roll forward and roll back based on database schema differences.<br><br>*2 Points*: Generate migration scripts based on reference databases<br><br>*2 Points*: Generate rollback scripts<br><br>*1 Point*: Generate migration scripts by comparing scripts in a repository with a target database |
| **Integration and Extensibility** | As part of the release process automation effort the tool needs to provide features to integrate in an automated build of a CI process. Additionally APIs for customization should be provided.<br><br>*2 Points*: command line execution |

| | |
|---|---|
| | *2 Points*: Plugins for build tools (e.g. ANT, Maven etc.)<br><br>*1 Point*: API to customize |
| **Versioning and Change Prediction** | The tool should support the delivery team by providing functionality to determine which database version is in use in a target environment. Additionally the tool should be able to identify differences between a target database and a reference database and predict outcomes if pending changes are applied.<br><br>*2 Points*: Compare database and show differences<br><br>*2 Points*: Maintain a common database version number<br><br>*1 Point*: Change forecast |
| **Monitoring** | The tool should provide information about changes that have been applied to the database and give feedback about ongoing integration and migration results.<br><br>*2 Points*: Generate reports about applied changes (success, error)<br><br>*2 Points*: Audit (which change was applied)<br><br>*1 Point*: Audit (who applied the change) |
| **Usability** | Does the tool provide an easy to use and learn user interface. Is it difficult to get support? Are there vendor specific constraints which make the tool unusable in certain environments?<br><br>*1 Point*: Documentation and community support<br><br>*1 Point*: No database vendor constraints<br><br>*2 Points*: Usable with simple SQL files |

| | |
|---|---|
| | *1 Point*: Usable for test use cases |

**Table 6.6 - Database migration tool criteria overview**

In context of this thesis the different criteria are equally important. In the next paragraphs these criteria are used to compare the tools.

## 6.4. Value analysis of database migration tools

In this section a subjective assessment of the tools found during the research is described. As no trial version of Dbmaestro Teamwork and Datical DB could be acquired those two tools will not be part of the assessment. At first a description of the test environment and use cases will be presented. Second a short summary of the tools will be given following by the value analysis. In order to assess the tools each tool was evaluated with a score between zero and five for each criterion defined. Where five is the best and means from the author's point of view the tools is state of the art and fulfills the criterion in a professional way.

**Test environment and Use cases**

For assessing the tools a simple test environment was set up in order to provide a first-hand impression of how the tools can be used for certain use cases and what skills are required. Figure 6.3 shows the test environment used which is composed of an Oracle Database, a local SVN repository for versioning of the database scripts and a Jenkins installation in order to test the integration capabilities of the different tools.

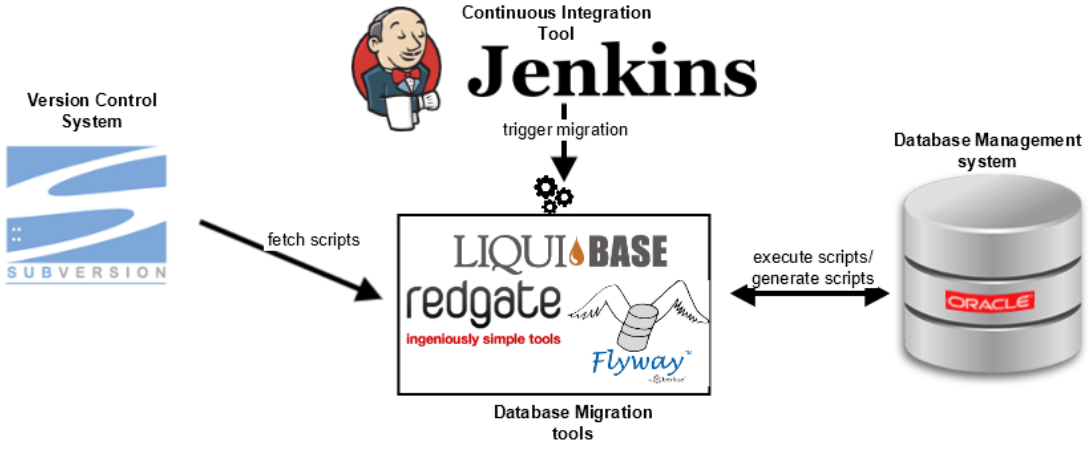


**Figure 6.3 – Test environment architecture**

After setting up the test environment a set of use cases was chosen to assess usability and capabilities of the tools. This set of use cases was elaborated in order to cover basic database migration tasks. The following tables describe the different use cases:

| Use case | Setting up a database baseline |
|---|---|
| **Description** | The migration tool is used to set up a database schema in a target environment from database scripts in the version control repository. |
| **PreCondition** | Database scripts for schema definition are available in the version control repository. The database has no structure and data definitions. |
| **PostCondition** | The schema in the database conforms to the expected schema according to definitions in the database scripts. |

**Table 6.7 - Use case: Setting up a database baseline**

| Use case | Executing database migration |
|---|---|
| **Description** | The migration tool is used to perform a non-refactoring transformation as described in 2.2 Definition: Database Change by executing a script that adds a nullable column to a table |
| **PreCondition** | The database script for adding the column is available in the repository. The column is not part of the table before executing the script. |
| **PostCondition** | The column was added to the table |

**Table 6.8 - Use case: Executing database migration**

| Use case | Performing several migrations |
|---|---|
| **Description** | Performing a referential integrity refactoring by adding a lookup table and a new foreign key constraint to an existing table to the database schema |
| **PreCondition** | The database scripts are available in the repository. |
| **PostCondition** | The table and the foreign key were added in the correct sequence. |

**Table 6.9 - Use case: Performing several migrations**

| Use case | Generating migration scripts |
|---|---|
| **Description** | The target database schema is missing a table compared to the newest schema version. The tool should create migration script to add the schema migration |
| **PreCondition** | The target database is available and a table is missing compared to the schema in the version control repository. |

| | |
|---|---|
| **PostCondition** | A script for adding the missing table was generated |

**Table 6.10 - Use Case: Generating migration scripts**

| Use case | Generate a script for referential integrity refactoring |
|---|---|
| **Description** | The target database schema is missing an integrity constraint compared to the newest schema version. The tool should create migration script to add and rollback the schema migration |
| **PreCondition** | The target database is available and a constraint is missing compared to the schema in the version control repository. |
| **PostCondition** | A script for adding the missing constraint and rolling it back was generated |

**Table 6.11 - Use case: Generate a script for referential integrity refactoring**

| Use case | Testing integration capabilities with Jenkins |
|---|---|
| **Description** | Testing automation capabilities by triggering the tool in a Jenkins job and setting up a database schema using scripts in the repository. |
| **PreCondition** | The target database is available and no table definitions are present |
| **PostCondition** | The database schema was set up in the target database. |

**Table 6.12 - Use case: Testing integration capabilities with Jenkins**

These use cases were used to test the tools and compare their features according to the criteria defined before. The following paragraphs describe the evaluation of the tools.

### Liquibase value analysis

Liquibase is a database migration tool that comes with its very own migration concept. The tool uses a XML file based approach to apply migration scripts. This introduces an abstraction between the migration tool and the database management system making it vendor independent. The tradeoff is that certain vendor specific instructions need to be scripted manually. Liquibase is using a two tier file structure consisting of one master “databasechangelog”-file which refers to many “changeset”-files. A “changeset” as can be seen in Figure 6.4 contains one or more database change instruction written in the Liquibase change language:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog/1.9"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog/1.9
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-1.9.xsd">
  <changeSet author="test" id="1">
    <createTable tableName="dept
      <column name="deptno" type="number(4,0)">
        <constraints nullable="false" primaryKey="true"
                primaryKeyName="DPT_PK"/>
      </column>

  </changeSet>
```

**Figure 6.4 - Liquibase changeset file structure**

Liquibase is using its own XML-structure which can be validated before execution in order to avoid breakdowns. Different XML-tags allow Liquibase to add control mechanism in the data migration procedure (e.g. precondition). Liquibase comes with a set of predefined refactorings and predefined rollbacks and provides extension API which allows to write customized refactorings and rollbacks. This allows to automatically generate migration scripts. When executing migrations the tool maintains a table that stores all changes applied to the database accompanied with the user executed the change, allowing to do some auditing and track a shared version number for database instances making them comparable. The log table and the information in the migration changeset files ensure that migrations that have been applied will not be applied twice.

| Criterion | Score | Comment |
|---|---|---|
| **Automate Database Integration** | 3 | ✓ Liquibase can set up a database using scripts in the repository.<br>✓ It provides control mechanism for database migration procedures.<br>✗ The XML files used need to be generated and maintained which requires additional effort. |
| **Automate Database Migration** | 4 | ✓ For general database migration the tool is able to automatically generate migration scripts.<br>✓ It also includes rollback instructions<br>✗ For generating scripts the tool requires a running reference and target database. |
| **Integration and Extensibility** | 5 | ✓ Liquibase provides a command line and plugins for various build tools.<br>✓ Furthermore it provides an API to generate customized refactoring in order to generate vendor specific refactoring instructions |

| | | |
|---|---|---|
| **Versioning and Change Prediction** | 4 | ✓ It provides a versioning of databases by maintaining a changelog table.<br>✓ It provides a function to compare two databases that shows differences between databases.<br>✗ The tool will not provide any warnings if there is a destructive change like referential integrity refactoring or dropping a column. |
| **Monitoring** | 2 | ✓ The changelog table gives information about applied changes. |
| **Usability** | 3 | ✓ The general approach is simple and there is a good documentation available and support from the community.<br>✓ The tools features are sufficient for the presented use cases<br>✗ It is recommended to use the XML changesets instead of SQL scripts. The XML-file approach requires training and manual effort to maintain database versioning.<br>✓ The tool has no database dependencies |

**Table 6.13 – Value analysis Liquibase**

### Flywaydb

Flywaydb is using a very simple approach for applying migrations to the database - in contrast to Liquibase. The migration concept of Flywaydb uses simple SQL scripts that only contain SQL instructions for database migration. In order to make this concept work the files need to match a certain naming pattern - indicating a version number – that allows Flywaydb to execute them in the correct sequence. Furthermore Flywaydb provides “SQL-Callbacks”(e.g. afterEachMigrate) in order to provide control mechanism during the migration procedure. FlywayDb - similar to Liquibase - maintains a database table to record applied database changes and to provide a shared database version.

| Criterion | Score | Comment |
|---|---|---|
| **Automate Database Integration** | 4 | ✓ With the simple approach the tool provides capabilities to set up a database from simple SQL scripts in the repository<br>✓ It provides control mechanism.<br>✗ The tool depends on SQL scripts following a filename pattern in order to operate correctly. |
| **Automate Database Migration** | 0 | ✗ Migration scripts need to be generated by hand or using a different tool |

| | | |
|---|---|---|
| **Integration and Extensibility** | 4 | ✓ FlywayDb is a command line tool that can be executed using Jenkins.<br>✓ It provides plugins for various build tools. |
| **Versioning and Change Prediction** | 3 | ✓ The tool maintains a changelog table that tracks applied changes and prevent executing changes twice.<br>✓ There is a function to check if changes applied to a database correspond to the scripts in the repository |
| **Monitoring** | 3 | ✓ The changelog table gives information about applied changes, when and by whom they have been applied. |
| **Usability** | 4 | ✓ The simple SQL script approach makes it easy to integrate the tool.<br>✓ FlywayDb provides a simple set of database migration functions.<br>✓ Documentation and community support is available.<br>✗ No automated migration script generation<br>✓ Database dependencies only exist in form of database connection libraries which need to be available. |

**Table 6.14 – Value analysis FlywayDb**

### Redgate Tools – Database Lifecycle Management (DLM) for Oracle

Redgate Tools offers its product for SQL Server and Oracle and the tools tested are specifically developed to work with Oracle databases. The DLM suite is composed of Schema Compare, Data Compare and Source Control tool. In contrast to the previous products these tools come with a clear structured GUI but can also be executed from a command line. The database migration concept of DLM uses simple SQL files that are generated using the Schema Compare and Data Compare tools. Furthermore using the Source Control tool a database can be put under source control and for any change to the database schema the corresponding scripts in the source control repository are adjusted. This repository can then be used to compare it with a target database in order to generate migration scripts and deploy them. During the test no custom scripts could be executed which implies that the Schema Compare and Data Compare tools are restricted to use scripts that have been generated by the DLM tool suite. Assuming that this concept emphasizes a process of utilizing a reference database where changes are developed and any migration script is generated using the same tools. Which would prevent errors from manually created scripts.

| Criterion | Score | Comment |
|---|---|---|
| **Automate Database Integration** | 4 | ✓ DLM allows to automate database setup using scripts that are generated and maintained by the tool suite.<br>✓ No control mechanism |
| **Automate Database Migration** | 4 | ✓ By simply comparing a target database with a reference database or the repository it can generate migration scripts.<br>✗ It does not automatically create rollback scripts. In this case target and reference database need to be exchanged. |
| **Integration and Extensibility** | 4 | ✓ DLM is executable via command line which allows to utilize the tool using Jenkins.<br>✓ Furthermore it provides plugins for various build tools. |
| **Versioning and Change Prediction** | 4 | ✓ Versioning is handled using the Source Control tool which manages changes in corresponding scripts automatically.<br>✓ The tool recognizes if there are destructive changes that require additional information in the script. Notifies user and aborts migration if necessary.<br>✗ No revision number for databases |
| **Monitoring** | 3 | ✓ The tool generates reports on executed database changes<br>✓ Reports describe what changes were applied |
| **Usability** | 4 | ✓ Profound documentation allows an easy start with the tool.<br>✓ The migration approach enforces using the same tools for generating and deploying simple SQL scripts.<br>✗ The tested tool has dependency to the database since it only works with Oracle database. |

**Table 6.15 – Value analysis Redgate tools: DLM**

**Value analysis overview**

| | Liquibase | FlywayDb | Redgate tools: DLM |
|---|---|---|---|
| **Automate Database Integration** | 3 | 4 | 4 |
| **Automate Database Migration** | 4 | 0 | 4 |
| **Integration and Extensibility** | 5 | 4 | 4 |
| **Versioning and Change Prediction** | 4 | 3 | 4 |
| **Monitoring** | 3 | 3 | 3 |
| **Usability** | 3 | 4 | 4 |
| **Total Score** | **22** | **18** | **23** |

**Table 6.16 – Overview migration tool analysis including total score**

In context of this analysis the overall result of the value analysis – illustrated in Table 6.16 shows that the Redgate Tools DLM suite is the solution with the best combination of features for database automation. Although the features and usability of DLM are superior in the context of this analysis, the costs of the tool suite might be a knockout criterion in other scenarios. Similar to that the migration concept of Liquibase might be an opportunity in some database automation scenarios but rewriting SQL scripts to the required XML format might be a knockout criterion in other scenarios. As already mentioned in context of this analysis all criteria are equally important. Impressive are the migration automation features of Liquibase and DLM. In contrast to FlywayDb which does not provide this feature at all the other tools provide functions to automatically create migration scripts based on the differences between a reference and a target database. This feature can significantly reduce scripting effort and makes it less error-prone. Nevertheless this feature is limited to a set of database schema changes. For schema refactorings like referential integrity, data quality and structural changes the tools need additional information which needs to be provided by a user. An analysis of web information systems shows that such schema changes occur relativley often for these systems. The underlying database of Wikipedia for example experienced 240 different schema version over six years. Another example is the scientific database of Ensembl Genome that experienced 410 database versions over nine years (Curino, et al., 2013). Relating to database automation efforts the more frequent such schema refactorings are the more human interaction is required in the release process.

The analysis shows that all tools provide features that fit the criteria for database automation defined in this thesis. It is notable that different databases automation scenarios will focus on some criteria more heavily. Referring to 4.3 Deployment Pipeline: Implementation Guidelines the delivery team has to find the best solution for their requirements which implies finding the right feature set. Database automation tools and best practices will help to strengthen the database deployment procedures since manual human interaction can be avoided. However it is similar important to integrate databases in the holistic release process that includes testing on different stages pursuing quality in database related tasks at a similar rate as with code. Database automation tools should help to reduce bottlenecks and complexity from database development and deployment tasks in order to relieve delivery teams and support them by releasing software in a Continuous Delivery approach.

# 7. Conclusion

The research on Continuous Delivery indicates that the demands on software has changed thus software companies need to adapt their practices in the interest of staying competitive. Research companies like Gartner predict that software companies will increasingly invest in adapting to DevOps and Continuous Delivery approaches. Therefore this thesis describes the fundamental principles of Continuous Delivery. Although many papers (Gmeiner, et al., 2015) (Feitelson, et al., 2013) (Akerele, et al., 2013) discuss the implementation of different Continuous Delivery practices in case studies and analysis only a few literature sources describe the fundamental theory. The main contributors to Continuous Delivery describe it as the evolution of the well-known software practice CI in combination with extended automation effort and continuous improvement. Additionally Continuous Delivery emphasizes to expand this techniques from the development team to the entire software release process. The research describes that in a Continuous Delivery environment the entire delivery team is equally responsible for the release process which implies common goals and a unified view on the release process. The fundamental concept to achieve this is an end-to-end release process or deployment pipeline. This abstract process view describes all tasks required to deliver software from the source control to the customer. The benefits of Continuous Delivery found in the course of this thesis reflect in process improvements that reduce costs, minimize risks of deployments and accelerate releases. Thus Continuous Delivery enables software companies to react faster on changes in the market and to focus on quality and innovation. This allows to pursue new business opportunities to strengthen the power to compete. However the remarks in this thesis also show that Continuous Delivery is not a product that can be bought but is a set of principles and practices that requires companies to design their own solution that fits to their needs. Nevertheless the best practices and implementation guidelines described in this thesis give a basic idea where to start.

Furthermore the thesis describes a research on database in a Continuous Delivery environment. The results presented reinforce that databases are a source of complexity for Continuous Delivery approaches as well as any automation effort. The result of the research on databases shows that an increased amount of automation of database related tasks is required. Hence delivery teams have to adapt to more agile database development practices. The thesis describes practices that aim to make database development more agile, which also includes automating database set up and update tasks to remove error-prone manual interactions. Achieving a consistent level of automation for databases empowers delivery teams to tackle higher level problems of deploying databases into production. In the interest of presenting a tool based solutions for database automation this thesis describes a research on database migration tools. The research shows that there is a proper number of products and open source projects that are engaged in database automation. In order to make the tools comparable the thesis describes a set of

criteria which in context of this thesis were found relevant and reflect on practices to automate databases. Although the migration concepts of the different tools vary, the tools shared certain functionalities. In order to gain an impression of the features, the tools were tested against a collection of uses cases. These use cases describe common database migration tasks. Based on the tool tests and the elaborated criteria a value analysis of the tools is described. Notably the overall result indicates that the tools are performing very similar in the context of this tool evaluation. Red Gate Tools: DLM has scored the highest because of its mature realization of database automation features. But both open source tools provided proper features as well. However it has to be said that in other projects certain criteria would be valued higher or additional criteria would be defined that describe the project requirements and hence the overall result has to be put in perspective of this thesis. Additionally a result of the tool analysis is that these tools have a limitation in concern of automated database migration. Human interaction is required as soon as the tool needs to generate data in the course of a database migration for example when adding integrity constraints. This requires context based information and thus DBAs or developers to provide this information. However most of the tools tested try to cope with this limitations by providing features to discover required human interaction before starting the migration.

Delivering database changes continuously requires delivery teams to excel on testing the changes against related applications as well as securing operational data while releasing to production environments. The right practices and strategies in combination with a disciplined team using a proper set of tools are required to achieve this. Continuous Delivery is a software discipline that emphasizes concepts and practices to implement an automated release process. Hence implementing Continuous Delivery should encourage teams to deliver changes of any part of the software in a continuous way. This should be achieved by continuous improvement of the entire release process. Metrics are required to measure changes in order to continuously improve the process. The thesis describes the value stream mapping methodology which gives an overview of valuable and nonvaluable tasks and the cycle time of a process. Although value stream mapping provides a useful metric to present improvements on the process level, in pursuance of continuous improvement, measurements have to be taken at a more detailed level of the process. Finding a good set of metrics to measure changes in the release process would be part of another research.

# Bibliography

**Akerele, , Olumide, Ramachandran, Muthu and Dixon, Mark. 2013.** System dynamics modeling of agile continuous delivery process. *Agile Conference (AGILE), 2013.* s.l. : IEEE, 2013.

**Ambler, Scott W. and Sadalage, Pramod J. 2006.** *Refactoring Databases: Evolutionary Database Design.* s.l. : Addision Wesley Professional, 2006.

**AmbySoft. 2006.** Why Agile Software Development Techniques Work: Improved Feeback. *AmbySoft.* [Online] Januar 01, 2006. [Cited: Oktober 01, 2015.] http://www.ambysoft.com/essays/whyAgileWorksFeedback.html.

**Anderasson, O.O and Tarasenko, A. 2013.** *Continuous Integration using LABView, SVN and HUDSON.* Geneva : s.n., 2013.

**Aurum, Aybüke and Wohlin, Claes. 2005.** *Engineering and Managing Software Requirements.* Heidelberg : Springer, 2005.

**Bandaru, Vijaya Kumar. 2013.** How to Manage the "7 Wastes of Agile Software Development". *Scrum Alliance.* [Online] September 27, 2013. https://www.scrumalliance.org/community/articles/2013/september/how-to-manage-the-7-wastes%E2%80%9D-of-agile-software-deve.

**Beck, Kent. 1999.** *Extreme Programming Explained.* s.l. : Addison-Wesley, 1999.

**Beck, Kent, et al. 2001.** Principles. *Manifesto for Agile Software Development.* [Online] 2001. [Cited: 10 1, 2015.] http://agilemanifesto.org/principles.html.

**Bharti, Nitin. 2012.** Results from InfoQ 2012 User Survey. *InfoQ.* [Online] 04 11, 2012. http://www.infoq.com/articles/infoq-user-survey-results-2012.

**Bogza, R.M. and Zaharie, D. 2008.** Business intelligence as a competitive differentiator. *Automation, Quatility and Testing, Robotics 2008. AQTR 2008. IEEE International Conference.* 2008, Vol. 1.

**Brown, Alan W. 2012.** *Enterprise Software Delivery.* s.l. : Addison-Wesley, 2012.

**Claps, Gerry Gerard, Svensson, Richard Berntsson and Aybüke, Aurum. 2015.** On the journey to continuous deployment: Technical and social challenges along the way. *Information and Software Technology.* Januar 1, 2015, pp. 21-31.

**CloudBees, Inc. 2015.** The Business Value of Continuous Delivery. *The Business Value of Continuous Delivery.* s.l. : CloudBees, 2015.

**Craig, Julie. 2014.** *DevOps and Continuous Delivery -Ten Factors Shaping the Future of Application Delivery.* s.l. : Enterprise Management Associates, 2014.

**Curino, Carlo, et al. 2013.** Automating the database schema evolution process. *The VLDB Journal.* 2013, 22.

**Datical. 2013.** Datical DB Technical Brief. s.l. : Datical, 2013.

**DbMaestro. 2015.** DBMaestro Teamwork - Database Development Life Cycle Solution, Agile Database Development for Oracle and SQL Server. *DBMaestro.* [Online] 01 01, 2015. [Cited: Oktober 20, 2015.] http://www.dbmaestro.com/product/product/.

**DBMaestro. 2013.** The Challenges and Pitfalls of Database Deployment Automation. 2013.

*Die DevOps-Bewegung.* **Peschlow, Patrich. 2012.** 2012, JavaMagazin, pp. 2-10.

**Driver, Mark, et al. 2014.** *Predicts 2015: Application Development.* s.l. : Gartner, Inc., 2014.

**Duvall, Paul M., Matyas, Steve and Glover, Andrew. 2007.** Continuous Integration - Improving Software Quality and Reducing Risk. *Continuous Integration.* s.l. : Addison-Wesley, 2007.

**Feitelson, Dror G., Frachtenberg, Eitan and Beck, Kent L. 2013.** Development and Deployment at Facebook. *Internet Computing, IEEE.* s.l. : IEEE, 2013. 17. 1089-7801.

**Flyway. 2015.** Documentation. *Flyway.* [Online] 01 01, 2015. [Cited: September 10, 2015.] http://flywaydb.org/documentation/.

**Fowler, Martin and Sadalage, Parmod. 2003.** Evolutionary Database Design. *Martin Fowler.* [Online] Januar 01, 2003. [Cited: 10 01, 2015.] http://martinfowler.com/articles/evodb.html.

**Fowler, Martin. 2013.** Continuous Delivery. *Martin Fowler.* [Online] May 30, 2013. http://martinfowler.com/bliki/ContinuousDelivery.html.

**—. 2013.** Deployment Pipeline. *Marting Fowler.* [Online] Mai 30, 2013. http://martinfowler.com/bliki/DeploymentPipeline.html.

**—. 2005.** The New Methodology. *Martin Fowler.* [Online] Dezember 13, 2005. http://www.martinfowler.com/articles/newMethodology.html#xp.

**Fritchey, Grant. 2014.** Building An Automated Database Deployment Pipeline. *SQL Server Pro.* [Online] November 11, 2014. [Cited: Oktober 13, 2015.] http://sqlmag.com/database-administration/building-automated-database-deployment-pipeline.

**Gartner, Inc. 2014.** Gartner Says Worldwide Software Market Grew 4.8 Percent in 2013. *Gartner.* [Online] 03 31, 2014. http://www.gartner.com/newsroom/id/2696317.

**Gmeiner, Johannes, Rammler, Rudolf and Haslinger, Judith. 2015.** Automated Testing in the Continuous Delivery Pipeline: A Case Study of an Online Company. *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on.* Graz : IEEE, 2015.

**Grolinger, Katarina and Capretz, Miriam. 2011.** A unit test approach for database schema evolution. *Information and Software Technology.* 2011, 53.

**Guernsey, Max. 2013.** Ten tips for Building an Agile Database Development Environment that Works. *informIT.* [Online] February 26, 2013. [Cited: Oktober 20, 2015.] http://www.informit.com/articles/article.aspx?p=2020066&seqNum=1.

**Harriman, Alan, Hodgetts, Paul and Leo, Mike. 2004.** Emergent Database Design: Liberating Database Development with Agile. *Agile Development Conference.* 2004.

**Hirt, Mitch. 2015.** Continuous Integration testing: What is it and how can you use it? *FileCatalyst.* [Online] Januar 08, 2015. [Cited: Oktober 15, 2015.] http://filecatalyst.com/de/continuous-integration-testing-what-is-it-and-how-can-you-use-it/.

**Humble, Jez and Farley, Davit. 2011.** *Continuous Delivery.* Upper Saddle River, NJ : Addison-Wesley, 2011.

**Hüttermann, Michael. 2012.** *DevOps for Developers.* 2012.

**Klettke, Meike, Scherzinger, Stefanie and Störl, Uta. 2014.** Datenbanken ohne Schema? *Datenbank Spektrum.* Februar 03, 2014, pp. 119-129.

**Leffingwell, Dean. 2007.** *Scaling Software Agility: Best Practices for Large Enterprises.* s.l. : Addison-Wesley Professional, 2007.

**Liquibase. 2015.** Liquibase | Database Refactoring | Home. *Liquibase.* [Online] Janure 01, 2015. [Cited: Oktober 13, 2015.] http://www.liquibase.org/documentation/index.html.

**Minduel, Luca and Morris, Kief. 2014.** Continuous Delivery Overview. *InfoQ.* [Online] März 31, 2014. [Cited: Oktober 15, 2015.] http://www.infoq.com/minibooks/continuous-delivery-overview#idp_register.

**Moen, Ronald and Norman, Clifford. 2011.** Evolution of the PDCA Cycle. *The University of West Georgia.* [Online] 10 01, 2011. [Cited: 11 29, 2015.] http://www.westga.edu/~dturner/PDCA.pdf.

**Myerson, Terry. 2015.** Announcing Windows Update for Business. *Windows Blog.* [Online] 05 04, 2015. http://blogs.windows.com/bloggingwindows/2015/05/04/announcing-windows-update-for-business/.

**Red Gate tools. 2013.** Continuous Integration for databases using Red Gate tools. *redgate.* [Online] Mai 07, 2013. [Cited: August 20, 2015.] https://www.red-gate.com/assets/hubspot/continuous-integration-using-red-gate-tools.pdf.

**Rümmler, Thomas and Schlag, Christian. 2014.** DevOps und Continuous Delivery: sich gemeinsam kontinuierlich verbessern. *ObjektSpektrum.* April 01, 2014.

**Sadalage, Pramod J. 2007.** *Recipes for Continuous Database Integration.* s.l. : Addison-Wesley, 2007.

**Stahl, Daniel and Jan, Bosch. 2014.** Modeling continuous integration practice differences in industry software development. *The Journal of Systems and Software.* January 1, 2014, pp. 48-59.

**Swartout, Paul. 2014.** *Continuous Delivery and DevOps: A Quickstart Guide.* s.l. : Packt Publishing, 2014. 978-1784399313.

**VersionOne, Inc. 2013.** State of Agile Survey Highlights Importance of Executive Support in Scaling Agile. *VersionOne.* [Online] 02 26, 2013. http://www.versionone.com/pdf/2013-state-of-agile-survey.pdf.