The present work was submitted to
the Research Group
Software Construction

of the Faculty of Mathematics,
Computer Science, and
Natural Sciences

Master Thesis

An Architecture for Self-Organizing Continuous Delivery Pipelines

presented by

Jan Simon Döring

Aachen, January 4, 2018

Examiner

Prof. Dr. rer. nat. Horst Lichter

Prof. Dr. rer. nat. Bernhard Rumpe

Supervisor

Dipl.-Inform. Andreas Steffens

Eidesstattliche Versicherung

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Masterarbeit mit dem Titel

An Architecture for Self-Organizing Continuous Delivery Pipelines

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen, January 4, 2018 (Jan Simon Döring)

Belehrung

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicher ung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen.

Aachen, January 4, 2018 (Jan Simon Döring)

Acknowledgment

I would first like to thank Prof. Dr. rer. nat. Horst Lichter for the possibility of writing my master thesis at his chair. I also thank him and Prof. Dr. rer. nat. Bernhard Rumpe for reviewing this thesis.

Special thanks go to my supervisor, Dipl-Inform. Andreas Steffens for his input and continuous feedback.

Furthermore I thank everybody at Traveltainment for their support during the case study, especially Manual Tilp, Oliver Dickmeis and Arne Czerwinksi, who assisted me whenever possible.

Finally, I thank my parents for providing me with unfailing support throughout this thesis. This accomplishment would not have been possible without them. Thank you.

Jan Simon Döring

Abstract

Attracted by the competitive advantages of being able to release software quickly and reliable, many organizations have adopted Continuous Delivery (CD) practices in recent years. However, both academia and industry have reported on several adoption challenges. One major challenge are the infrastructure and tools supporting Continuous Delivery with existing delivery systems being inflexible and require lots of technical and process-related knowledge. This thesis introduces a microservice-based architecture for flexible and maintainable delivery systems which provides means to reduce the knowledge required through analyzing, complementing and optimizing the delivery model. While at the same time providing extensive validation support. A case study in an industrial context showed promising results and confirmed the practicability of our approach. The prototype's delivery process adoptions thereby provided up to 3,31 times faster feedback times as compared with the manually modeled process. Overall, the architecture provides a flexible foundation for further research in various areas, ranging from modeling tools with auto-completion and recommendation support over smell and anti-pattern detection mechanisms to the development of a holistic delivery ecosystem.

Contents

List of Tables

List of Figures

List of Source Codes

1. Introduction

Contents

An important activity in the Software Development Life Cycle is the delivery of the developed software [IEE02]. Depending on the software development approach used, it can be performed iteratively. Agile Software Development (ASD) is such an iterative approach to gain short customer feedback cycles and to quickly respond to changing requirements. As stated in the Agile Manifesto, working software should be delivered frequently, *"from a couple of weeks to a couple of months"* [Bec+01]. With the increasing adoption of ASD, practices like Continuous Integration became more and more mainstream [Rod+12]. A recent evolutionary step called "Continuous Software Engineering" [Bos14], extends ASD to approaches, which allow organizations "to develop, release and learn from software in rapid parallel cycles, such as hours, days or very few weeks" [Ber15]. Central to this holistic approach is Continuous Delivery (CD), *"a software development discipline where [...] software [is built] in such a way that the software can be released to production at any time"* [Mar13b]. A survey from 2015 finds an increasing adoption of Continuous Delivery by organizations, calling Continuous Delivery *"the new normal for software development"* [Per15].

However, adopting Continuous Delivery is challenging. One major challenge are the infrastructure and the tools supporting Continuous Delivery as reported by literature ([CBA15], [LIL17]) and indicated by the continuous development of new delivery system. Especially big, successful companies like Pivotal, Netflix, Facebook and Google build specific systems and toolchains tailored for their needs ([Pivd], [Neta], [Faca], [Goo]).

From a technological perspective, delivery systems need to integrate lots of different tools and technologies and need to cope with evolution, which is a major challenge for existing system. The ThoughtWork Technology Radar highlights the in-maturity of existing delivery systems and suggests to not use a single system across teams in order to prevent conflicts arising from shared tooling and infrastructure [Tho17].

From a modeling perspective, i.e. the definition of the delivery process activities to be executed, existing delivery systems in principle force users to copy and paste shell snippets into the GUI. This not only introduces serious problems in terms of maintainability and reuseability, but also leads to SnowflakeServer [Mar12]. The second generation delivery systems tackle those problems by following the *infrastructure as code* [Mar16] movement and keep all delivery process modeling related information in a single, version-controlled file. Thereby calling this approach *pipeline as code.*

Taking a look at listing 1.1, an example pipeline as code model from Jenkins [Clo], a popular delivery system, indicates still existing problems with this approach:

```
 1  node {
 2      stage('checkout git') {
 3        git branch: 'master', credentialsId: 'GitCredentials',
              url: 'ssh://git@myScmServer.com/repos/myRepo.git'
 4      }
 5
 6      stage('build') {
 7        sh 'mvn clean package'
 8      }
 9
10      stage('deploy dev'){
11        sshagent(['RemoteCredentials']) {
12          sh "scp target/*.jar
                root@${devServer}:/opt/jenkins-demo.jar"
13          sh "ssh root@${devServer} nohup java
                -Dserver.port=${devServerPort} -jar
                /opt/jenkins-demo.jar &"
14        }
15      }
16  }
```

Source Code 1.1: Jenkins Pipeline as Code Example (taken from [HP17])

Beside some basic abstractions, users must define every detail. Thereby, they not only need to have deep technical tool knowledge as they are primarily forced to define raw shell commands (see line 13 for example), they are also required to have profound process-related knowledge to define the right execution order.

Considering that the delivery process already is defined "as code", fundamental software development practices like validation, code completion or refactoring support are completely missing, which ultimately makes the delivery process modeling error prone and hard to maintain. Therefore, this thesis tries to tackle the aforementioned issues by means of a flexible and maintainable delivery system architecture that allows both to reason about and to optimize a modeled delivery process, which, as a consequence, enables validation support and reduces the required amount of technical knowledge.

1.1. Thesis Structure

Clear terminology is important for discussing a topic. Therefore, chapter 2 introduces required foundations and defines central terms. In the course of this, the ambiguity of the Deployment Pipeline term is highlighted and alternative terms are defined. Based on the clear terminology, chapter 3 then summarizes the problems mentioned above, identifies related challenges and defines the thesis scope. Guided by these challenges and the scope, it derives requirements which need to be met in order to tackle the challenges. Chapter 4 then evaluates existing delivery systems against these requirements. Taking into account that they were derived from problems which existing delivery systems face, chapter 4 only evaluates bleeding edge delivery systems to see if the requirements still impose problems. Considering the complexity and ambiguity in the software delivery domain, chapter 5 then identifies and details important domain concepts incorporating on the terminology defined in chapter 2. Chapter 6 maps these concepts to a high-level architecture, whose central software components are then detailed in chapter 7. Based on the proposed design, chapter 8 provides a corresponding prototypical implementation and a description language for defining a delivery process. In chapter 9 both the prototype and the description language are used to conduct a case study and to evaluate the concepts against the requirements. Chapter 10 concludes this thesis.

2. Foundations

Contents

This chapter introduces required foundations for the remainder of this thesis.

2.1. Software Delivery Process

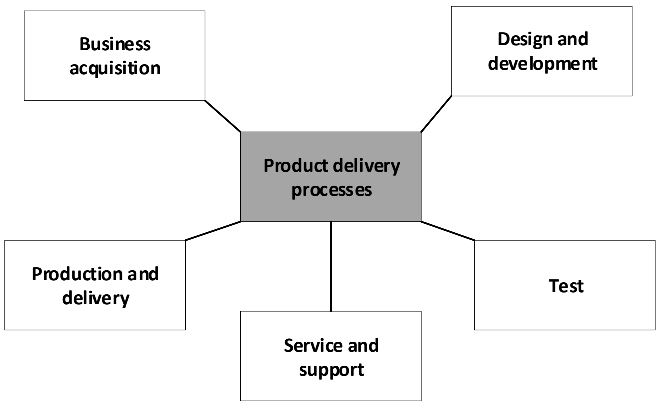


Figure 2.1.: ISO 9001 Core Delivery Processes (cf. [Som11])

Fundamental to this thesis is the Software Delivery Process, which this section therefore defines. The ISO 9001 standard specifies five core processes related to product delivery as depicted in figure 2.1. These processes match the software life cycle phases (requirements phase, design phase, implementation phase, test phase, installation and checkout phase)

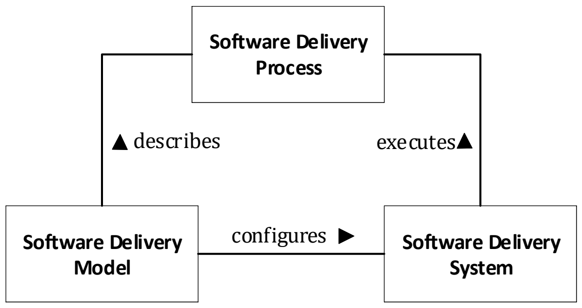


Figure 2.2.: Relations between Software Delivery Process, Model and System

(cf. [LL10]). Thus, the software delivery process can be defined as *the process of getting a software product to market.* Since this thesis only considers certain phases of the software life cycle, i.e. from implementation to installation in production, we - similar to [Bar+10] - slightly adapt the definition to our purpose and define:

> The **Software Delivery Process** is the process comprising the build, deploy, test and release process to get a software product from development to installation in production.

Furthermore, we define

> A **Software Delivery System (SDS)** is an integrated software system implementing one or multiple software delivery processes.

Software Delivery Systems usually do not statically implement a single software delivery process. Instead, they provide variability to allow for different delivery processes, i.e. they can be configured. This configuration is done by means of a Software Delivery Model. We define

> A **Software Delivery Model** describes a Software Delivery Process.

Following DevOps practices like *infrastructure-as-code* [BWZ15], modern Software Delivery Systems use a domain specific language to define the Software Delivery Process *as-code.* This language can be both declarative or imperative. We call such languages **Software Delivery Process Description Languages (SDPDL)**. Figure 2.2 provides summarizes these definitions by providing their static relations.

2.2. Continuous Delivery

Humble and Farley coined the term *Continuous Delivery* with their book Continuous Delivery - Reliable Software Releases Through Build, Test and Deployment automation (see [HF10]). The book title already provides a glimpse what Continuous Delivery is:

> Continuous Delivery is a set of practices (see section 2.2.1) that aims to deliver value to customers rapidly, reliably and repeatedly with minimal manual overhead (cf. [HF10]).

Continuous Delivery thereby bases on Continuous Integration (CI), an agile practice that provide means to ensure to have a shippable product available that passed at least unit and integration tests [FS14].

2.2.1. Principles

At it's heart, Continuous Delivery relies on five principles [HF10]:

Automation As much as possible of the Software Delivery Process should be automated. If it is not automated, it is not repeatable. In addition, manual steps are error-prone. Therefore automation helps to delivery value reliably and repeatably.

Work in small batches Small batches help to keep both the amount of changes small and the duration between releases short, i.e. the overall delta between the old and the new version is small, which reduces the risk associated with releasing since it is easier to overlook changes and to roll back in case of a problem.

Build quality in Adopted from the lean movement, "build quality in" advocates to use Continuous Integration and comprehensive automated testing in order to catch defects as early as possible in the process to reduce follow-up costs.

Everyone is responsible for the Delivery Process This principle is similar to the Dev-Ops driven encouraging of collaboration. It states, that enterprises make money by delivering their products to customers, thus everybody should care for the Delivery Process. Blaming each other, when something goes wrong does not help to deliver the product.

Continuous Improvement The delivery process evolves with the application. Similar to the Deming cycle (plan, do, study, act), the whole team should retrospect the delivery process.

2.2.2. Benefits

Following Continuous Delivery principles provides several benefits. Rodriguez et al. identified the following in their systematic mapping study [Rod+17]:

Shorter time-to-market Continuous Delivery practices allow to deliver features faster, which helps in responding to changing conditions.

Improved release reliability Continuous Delivery practices detect errors early in the development process, which allows to quickly fix them. In combination with repeatability it helps to release more reliability.

Improved developer productivity Developers have fewer struggles in releasing software since Continuous Delivery enables the to release by means of pushing a button. Thus, developers can concentrate on other tasks.

Continuous feedback Releasing features early and often allows to gather feedback from customers frequently.

2.2.3. Deployment Pipeline

The Deployment Pipeline is a central part of Continuous Delivery (cf. [Mar13c]). Humble and Farley coined the term as *"an automated manifestation of your process for getting software from version control into the hands of your users."* [HF10], i.e. the deployment pipeline is a software project that automates the process. At other places, Humble and Farley describe the deployment pipeline as a model (*"the process modeled by the deployment pipeline"*[HF10]). Bass et al. identify similar dimensions: *First, the DevOps pipeline itself is a piece of software [...]. Second, the DevOps pipeline has characteristics of a process.* [BWZ15].

We think that these different dimensions lead to ambiguity and thus complicate understanding when using the deployment pipeline term as is. In addition, the pipeline metaphor is to linear in our opinion and does not embrace the potential parallelization of software delivery process activities. Therefore, we try to stick to our dedicated terms defined in section 2.1 in the remainder of this thesis. Thereby, we use Software Delivery and Delivery interchangeable. Since literature uses the term pipeline, we sometimes need to fall-back to this term too. Summing up, a Deployment Pipeline has three dimensions:

- A deployment pipeline models a delivery process. Here, we use **Software Delivery Model** (see section 2.1), abbreviated as Delivery Model, instead of deployment pipeline.

- A deployment pipeline is a software product. Here, we use **Software Delivery System** (see section 2.1), abbreviated Delivery System, instead of deployment pipeline.

- A deployment pipeline itself has characteristic of a process. Here, we use **Software Delivery Process**, abbreviated Delivery Process, instead of deployment pipeline.

In Section 2.1 we introduced the term Delivery Process Description Language (cf. section 2.1). To ease understanding outside this thesis context, we use *Pipeline Description Language (PDL)* as a synonym.

2.2.4. Software Delivery System Requirements

Humble and Farley define the following requirements for a Software Delivery System [HF10]:

Provide visibility The Software Delivery System implements the Software Delivery process. One important requirement of the Software Delivery System therefore is to make this process visible both statically and dynamically to everyone involved (Bass defines developers and operators as delivery system stakeholder [BWZ15]). This supports collaboration and eases process improvement. [HF10]

Provide feedback The Software Delivery System executes the Software Delivery Process on every commit while automating as much as possible. Thus, problems can be identified early on in the process. Important goal of the Delivery System therefore to provide feedback on the execution to enable teams to resolve problems as fast as possible.

Enable deployment at will A third requirement of the Software Delivery System is to enable Continuous Delivery, i.e. to provide the capability of deploying in a self-service manner.

Generalizing these requirements, a Software Delivery System provide means to both improve the Software Delivery Process (feedback and visibility) and to improve the product quality (deployment at will requires to build quality in).

2.2.5. Principles

This section briefly discusses important Software Delivery practices & principles:

Build artifact once [HF10] In the previous section a central Delivery System quality was discussed, namely to gain confidence that a modification is ready for release. Therefore a series of tests is performed. If they do not test exactly the same artifact, but instead built the artifact repeatedly, there is no confidence in the artifact's release fitness, as each built might introduce some differences.

Deploy the same way to every environment [HF10] Beside gaining confidence about the artifact release fitness (cf. build once principle), it is also important to gain confidence in the deploy process. Therefore the Delivery System should deploy the same way to every environment to minimize the error potential.

Keep everything in version control [HF10] Everything required to re-create or rollback the entire system (delivery system configuration & related environments) to a certain snapshot should be stored in version control. This way the delivery process is controllable and potentially reproducible and *snowflake-servers* [Mar12] are prevented.

Smoke-Test the deployment [HF10] When the Delivery System deploy an application, Humble advocates to smoke-test the application and services the application depends on, to make sure that the application is in a stable state.

Propagate changes instantly [HF10] Continuity and speed are central continuous delivery themes (cf. section 2.2. Therefore it is necessary that every change instantly propagates through the Delivery Process activities instead on relying on a fixed schedule.

Fail Fast [Mar13a] In system design fail fast is the property of the system to abort normal operation if some unexpected conditions occurs instead of trying to continue the execution in a possibly flawed stated [Sho04]. This property is important for Delivery Systems. They shouldn't waste resources. Instead they should run tests as early as possible to detect defects.

Parallel Workflow, keep critical path short [Ken15] [Her15] A requirement for the previous principle (propagate changes instantly) is to parallelize as much as possible to keep the delivery process duration short and to provide fast feedback.

Immutable infrastructure [Kim17] The idea of immutable infrastructure is to produce images, that do not change during runtime. If a change should be made to this image, a new version is deployed. This way, the delivery process is as deterministic as possible, greatly improving reproducibility and traceability.

Use deployment strategies [Kim17] The Delivery System is not only about orchestration of tools. One important aspect is the deployment itself. A delivery system should offer different deployment strategies taking advantage of the cloud. The following briefly describes the two common ones:

Blue / Green deployment : This strategy assumes two independent stacks with a load balancer in front. One stack serves the currently deployed version of our system and the other one is idle. With a release, the new version gets deployed to the idle stack and the traffic is switched to this stack, so that the previously active stack becomes idle. To rollback, one just has to adapt the load balancer and switch traffic to the old stack. (cf. [Mar10])

Rolling Upgrade : The rolling upgrade strategy incrementally replaces instances of the old version with instances of the new versions and shifts the traffic accordingly. For example, first 5% of the instances are replaced, then 20%, 50% and finally 100%. Between each cut-over a validation gate can decide if the new instances are working as expected. (cf. [Kim17], [BWZ15])

Operational integration [Kim17] During the delivery process execution, lots of data (e.g. relevant commits, configuration changes, which tests were run) is generated, which might help operations in resolving issues that occur during runtime. This practice therefore requires the delivery system to make these data easily accessible.

2.3. Terminology

Figure 2.3 distills central Software Delivery concepts and their static relations based on our definition of a Software Delivery System and the Software Delivery Model. Because

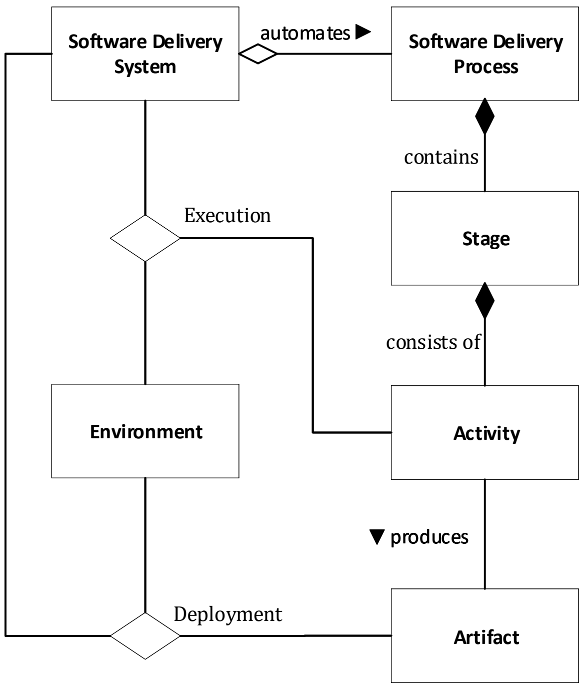


Figure 2.3.: Static relation between Software Delivery terms

few research exists in this area, the concepts were extracted from field reports and existing tools [Tho][EBM16][Pivc][Spi][FFB13].

Activity Activities are the units of work in the Software Delivery Process. They constitute relevant functions in the delivery process like compiling or testing. Activities might be performed in parallel or sequentially depending on their dependencies on each other and the capabilities of the Software Delivery System.

Stages Stages are a collection of activities that semantically belong together. Each stage is uniquely identifiable by its name. Borrowed from stage-gate processes [Cam], a stage in the Software Delivery Process should end with a decision point to ensure that certain requirements (e.g. quality related) are met. This (manual or automatic) decision point is called quality gate [And14]. A typical stage is the build stage that comprises compile and unit tests activities. Section 2.3.1 discusses further stages.

Artifacts As the Software Delivery System automates the Software Delivery Process, its activities inherently deal with artifacts, e.g the source code or the compiled binary. Depending on the activity the Software Delivery System infrastructure either stores (if the activity execution produces an new artifact) or provides the artifact.

Environment In Software Delivery an environment describes the context to which an artifact is deployed. An environment is defined through the servers hardware configuration and configuration of the operating system and supporting infrastructure (e.g. messaging systems, logging) [HF10]. Bass differentiates four different environments [BWZ15]:

- **Pre-commit** : The pre-commit environment is the developer's computer. Often only a certain module of the system is used and external systems are mocked.
- **Build and integration testing** : The build and integration testing environment is used to build and integrate the system and to perform integration testing.
- **UAT/staging/performance testing** : The staging environment is as similar to the production environment as possible. It is used to perform user acceptance or stress testing.
- **Production** : The production environment reflects the production / live system, which serves the end-users, respectively clients.

Following Continuous Delivery practices, these environment should get increasingly (from pre-commit to staging) more similar to production. In addition, they should be easily reproduceable (cf. infrastructure as code).

2.3.1. Stages & Activities

Stages and activities are central concepts in the Software Delivery domain. As described above, a stage represents a certain phase in the software delivery process (cf. section 2.1) and activities represent the corresponding process activities for this phase. While the stages and activities vary between projects and organization from an implementation perspective, they are similar on a conceptual level [HF10]. Bass argues, that Software Delivery systems standardizes the application life cycle [BWZ15]. Therefore a common stage sequence can be defined. Table 2.1 depicts these stages and their related activities extracted from literature (cf. [HF10] [AM16] [Leh+15] [Rod+17] [BWZ15]).

Build Stage The build stage ensures that the system to be delivered works at the technical level. For this reason, the build stage compiles the code, runs unit tests and statically analyses the code (cf. [HF10]). Finally a deployable assembly is *baked*. Baking integrates the compiled binary and configuration and bootstraps them onto a self-contained package with all required dependencies. Bass differentiates four types of packaging possibilities [BWZ15]:

- **Runtime-specific packages** require a runtime to be executed. Examples are Java Archives (they need the Java Virtual Machine) or Web Application Archives (they require an Application Server).
- **Operating System Packages** are Operating System-specific as the name implies. They cannot be installed or executed on an operating system different from the target one. Examples are Debian (.deb) packages or a Microsoft Installer (.msi).

Number	Stage	Activity
1	Build	Compile Bake Unit Tests Static Analysis
2	System Testing	Integration & System Testing Environment Provision Deployment of binaries Acceptance testing
3	Capacity	Performance Testing Security Testing Environment Provision Deployment of binaries
4	User Acceptance	User Acceptance Testing Exploratory Testing Environment Provision Deployment of binaries
5	Release	Environment provision Configuration of supporting infrastructure (e.g. Monitoring) Data Migration DNS Modification
6	Teardown	Undeployment

Table 2.1.: Software Delivery Process Activities

VM images are Operating System independent. They virtualize the hardware, contain everything required to run the application and isolate the process and files from the host system. They can also be used as a template for other virtual machines. However, they require an compatible hypervisor to run.

Lightweight containers are like VM images, i.e. they are self-contained and isolate the application from the host system. In contrast to VM images, they do not require a hypervisor. Instead, they make use of certain kernel features, which reduces overhead, load and size.

System Testing Stage The system testing stage assures compliance with (mainly) functional requirements by performing automated integration and system tests. Since these kind of tests require a running system, the system testing stage also contains activities to configure, respectively to provision an environment for executing these tests.

Capacity Stage The capacity stage asserts that the system fulfills non-functional requirements [HF10]. In this stages, eventual long running (but still automated) capacity (e.g. performance) tests are executed.

User Acceptance Stage While the previous two stages dealt with verification, i.e. the product conformance with its specification, the User Acceptance Stage primarily is about validation, i.e. the product conformance with the customer's expectation (cf. [Boe79]). This stage therefore comprises User Acceptance tests as well as manual system tests, e.g. exploratory tests.

Release Stage The release stage is about delivering the system to users. How the delivery is realized depends on the domain, e.g. the system might be shipped as packaged software, or it is deployed into an environment.

Teardown Stage After the new system has been released in the previous stage, the old system can be torn down. Activities in the teardown stage are responsible for these clean-up tasks. Depending on the context, these activities might be realized in different ways. In the cloud context, for example, the old stack gets undeployed, while for packaged software the release might be deleted from content-delivery networks.

2.4. Activity Classification

As the general stage and activity abstraction (cf. section 2.3) makes it difficult to reason about Software Delivery Processes, Hermanns introduced an activity classification [Her15]. Our concept (cf. section 5.1.1) is inspired by this classification, therefore we briefly introduce Hermanns classification here. Figure 2.4 depicts the classification based on a sample delivery process.

Hermanns introduced two types of (data-flow related) activities:

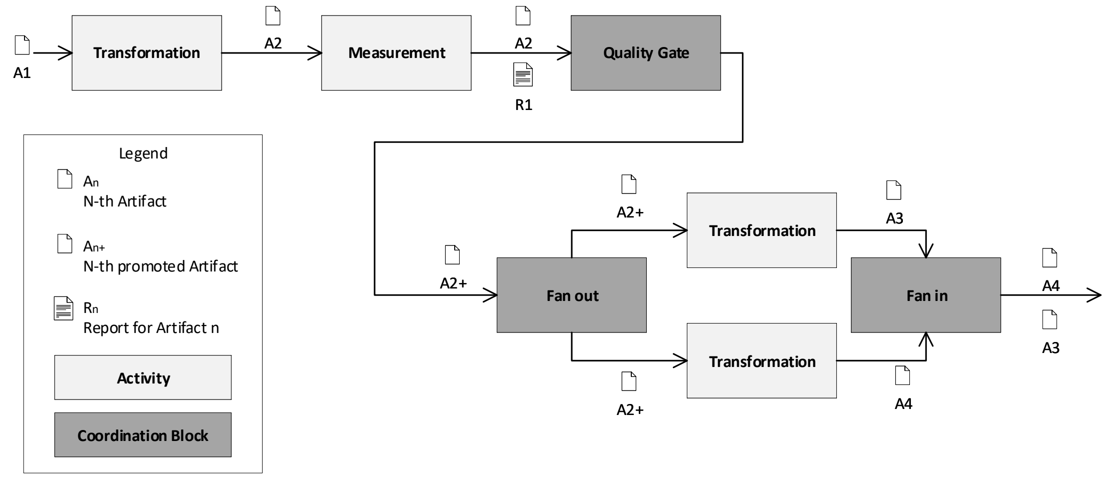


Figure 2.4.: Delivery Process Building Blocks according to [Her15]

Transformations accept one or multiple artifacts as input and produce a new artifact by applying operations on the input. A typical transformation is the compile activity, i.e. the translation from source code into executable machine code.

Measurements determine quality-related characteristics of their input artifact in a report. They output this report and the original input artifact.

Furthermore, he differentiated two types of coordination (control-flow related) activities:

Quality Gates are a means to abort the delivery process execution if certain quality criteria are not met. They accept an input artifact, related measurement reports and a policy describing the expected quality attributes. If the reports meet the policy, the quality gate promotes and outputs the input artifact. If the policy is not fulfilled, the quality gate fails and the delivery process execution gets aborted.

Fan in/out activities enable the parallel execution of activities. Fan out activities fork the execution flow and Fan in activities synchronize these forked activities, i.e. Fan in activities wait until their previously forked activities are finished and output all results.

2.5. Summary

This chapter introduced required foundations for the remainder of the thesis. We introduced the Software Delivery Process, and defined related terms. We motivated Continuous Delivery and detailed related principles. These principles will serve as requirements in the course of the thesis. We also provided an overview of the terminology. Thereby we explored the ambiguous dimensions of Deployment Pipelines: A deployment pipeline simultaneously is a software product, models a process and itself is a process.

To avoid misunderstandings, we clearly address each dimension individually by means of our definitions. A Software Delivery Model reflects the process model characteristics of deployment pipelines, a Software Delivery System the software product dimension and Software Delivery Process reflects the process dimension. We use these definition in the remainder of the thesis. The next chapter details the problems we identified and want to tackle with this thesis.

3. Problem Statement

After chapter 1 introduced the thesis and chapter 2 discussed the foundations, this chapter details the problem this thesis tries to tackle. For this reason, the following introduces the main challenges related to delivery systems, afterwards we define the thesis's scope and deduce top-level requirements from the identified challenges.

3.1. Challenges & Scope

Continuous Delivery (CD) offers huge benefits as discussed in the previous chapters. But adopting these practices is difficult [CBA15]. Laukkanen et al. identified six recurring problem themes in their systematic literature review [LIL17]. The themes are build design, system design, integration, testing, release, and human and organizational resource related problems. Following the build design theme, Chen postulates the need for a new "CD platform", since existing tools are inhibitory in achieving CD [Che17]. A systematic mapping study by Rodriguez et al. found, that only 7% of their analyzed publications contribute to this Delivery System area [Rod+17]. We therefore want to concentrate on the Delivery System problems in this thesis. The identified Delivery System problems are:

P1 - Modifiability *"The build system cannot be modified flexibly"* [LIL17]

P2 - System Complexity *"The build system [..] [is] complicated or complex"* [LIL17]

P3 - Model Complexity *"The build [..] scripts are complicated or complex"* [LIL17]

Overall, Laukkanen et al. find both technical, architecture-related issues (the delivery system is complex and inflexible) and functional, delivery model related issues (the delivery models are complicated). Our findings in chapter 1 confirm the delivery model related issues.

These problems indicate two major challenges for building delivery systems:

C1 - Project Evolution A delivery system has to cope with evolution. Following the Law of Continuing Change [Leh80], a software project will change and evolve [1]. As the delivery system realizes the project's delivery process (cf. section 2.1) it needs to reflect these changes. Similar to our differentiation between delivery system, delivery process and delivery model (cf. section 2.1), the evolution has multiple dimensions:

[1] Assuming the program is an P- or E-Program [Leh80]

C1.1 - Technical Evolution The software project architecture might evolve, which imposes new functional requirements on the software delivery system. For example new technologies might be integrated into the software project, which require the delivery system to cope with heterogeneous technologies.

C1.2 - Process Evolution The software project's delivery process might evolve, e.g. a testing activity is integrated. The delivery system needs to reflect such changes.

C1.3 - Organizational Evolution From an organizational perspective, new non-functional requirements or policies might emerge that affect the delivery process but also the delivery system itself.

Overall, the evolution challenge requires delivery systems to be flexible and maintainable.

C2 - Modeling Usability ISO 25010 defines Usability as the *"degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use"* [ISO15]. The identified usability challenge relates to the modeling context. Generalizing Laukkanen's finding, the delivery model is complex and difficult to understand. This problem has several related challenges:

C2.1 - Minimize required technical knowledge Currently, users need to provide lots of technical details which complicates the modeling. The Delivery System should minimize the amount of technical knowledge required to define a delivery model.

C2.2 - Minimize required process knowledge The Delivery System should minimize the amount of process knowledge required to define a delivery model. Users should not be concerned with parallelization respectively defining an explicit order.

C2.3 - Support modeling activities Delivery Systems should assist users in defining the delivery model as much as possible. They need to offer extensive validation support and support-mechanisms like activity auto-completion or recommendation of matching activities.

Research suggests that build maintenance account for 27% of code development and 44% of test development effort (cf. [McI+11]). Therefore it is of great importance to tackle the modeling usability challenge.

3.1.1. Scope

In this thesis we try to tackle the aforementioned challenges from an architectural view. To tackle section 3.1 flexibility is important, i.e. the capability to adapt when external changes (in our case technology changes, delivery process changes, organizational changes) occur. We focus on flexibility of the delivery system architecture and extensibility of new (heterogeneous) technologies. For section 3.1 we focus on accessibility and user error protection. Both are sub-qualities of usability ([ISO15]). Accessibility is defined as the *"degree to which a product or system can be used by people with the widest range of characteristics and capabilities"* [ISO15], which means we want to reduce the required knowledge for defining a delivery model. User error protection is the *"degree to which a system protects users against making error"* [ISO15], i.e. we also want to provide validation support for the delivery model.

3.2. Architectural Requirements

The previous section identified challenges relevant for this thesis scope. Based on these challenges, this section derives requirements necessary to tackle the challenges.

Following our scope (section 3.1.1), we do not consider specific functional requirements for a concrete delivery process (e.g. the delivery system needs to compile java code). Instead, we focus on providing a framework architecture, that has dedicated hot-spots for tailoring to a concrete project. The following provides (top-level) requirements for this framework architecture. Because of C2 - Modeling Usability, some requirements tailor the delivery model. Since these requirements also influence the architecture by requiring certain components, we list them here. With each requirement we list the addressed challenge from above.

Req-1: Integration of heterogeneous technology The architecture needs to support easy integration of different tools and services ranging from version control systems, build and test tools, infrastructure provisioning tools and deployment tools. (C1 - Project Evolution)

Req-2: Modularity Modularity is an important strategy to improve the flexibility of a design which also aids maintainability [BCK12]. Given our focus on flexibility to cope with evolution the architecture therefore should be modular. (C1 - Project Evolution)

Req-3: Activity Abstraction The architecture should introduce abstractions for the delivery process activities. From a technological perspective this eases to provide new activities (extensibility). From a modeling perspective it results in encapsulation of technical details, reducing the amount of knowledge required to define a delivery model. (C1 - Project Evolution, C2 - Modeling Usability)

Req-4: Self-Organizing The architecture should support mechanisms to gather information relevant for the delivery process and use these additional information to analyze, complement and optimize the delivery model. From a modeling perspective, this reduces the amount of information required and eases to meet Req-7: Best practices. From a technical perspective, the system can more easily cope with evolution (C1 - Project Evolution, C2 - Modeling Usability)

Req-5: Custom PDLs The architecture should support different, custom pipeline descriptions languages. This allows to users to introduce PDLs tailored for their context. For example, if their projects predominantly use Amazon EC2 for deployment, it might makes sense for an enterprise to introduce a Amazon-specific PDL to improve further improve usability. (C2 - Modeling Usability)

Req-6: Model Validation A sub-quality of usability in ISO 25010 is user error protection. Important requirement for user error protection is validation. Thus, the architecture should enforce mechanisms to validate a delivery model. The validation should not only be technical (syntax) but also functional (e.g. compatibility of modeled activities) (C2 - Modeling Usability)

Req-7: Best practices The architecture should support and conform to Continuous Delivery principles & best practices (cf. 2.2.5), e.g. build once. Best practices ease flexibility and maintainability, thus allowing the architecture to more easily cope with evolution. In addition, they improve usability as they are founded on experiences others made. (C1 - Project Evolution, C2 - Modeling Usability)

Req-8: Traceability Visibility, i.e. to provide insight into every step in the delivery process, is an important delivery system requirement (cf. section 2.2.4). From an architecture perspective, traceability, i.e. the capability the backtrack what action produces which result, is important for visibility. In the delivery system context, traceability means to determine how a system got to into production (cf. [BWZ15]), i.e. the delivery system needs to track the executed activities and needs to make their results (including produces artifacts) available. These results can then also assist debugging or to evolve the delivery system. (C1 - Project Evolution)

3.3. Summary

This chapter set the thesis's scene. It introduced several problems existing Delivery Systems face. Given our missing large-scale experience of existing Delivery System, these problems were primarily taken from literature reviews. We identified two related challenges: First challenge is the project evolution. As stated by the Law of Continuing change, a software project will change and evolve. As the delivery system realizes the project's delivery process it needs to reflect these changes. The second challenge we identified is usability, more preciously usability in defining a delivery model. Existing delivery models require a lot of technical information and do not provide validation support. Based on these challenges we then defined this thesis scope, which is geared towards a framework architecture that is independent of specific functional requirements for a concrete delivery process. For the usability challenge we restricted ourselves on the usability sub-qualities of accessibility and user error protection. Having defined our scope, we introduced requirements the architecture needs to meet. In the next chapter we then evaluate existing delivery systems against these requirements. Since literature already reports problems of existing systems in this chapter, we thereby only evaluate bleeding-edge delivery systems.

4. Related Work

The previous chapter identified challenges Software Delivery Systems face and deduced architectural requirements. This chapter evaluates existing Software Delivery Systems against these requirements. Since the requirements were derived from problems existing Delivery Systems have (cf. chapter 3), we only evaluate bleeding-edge Software Delivery Systems in this chapter to verify if the reported problems still exist.

4.1. Spinnaker

Spinnaker [Netd] is a cloud-deployment-focused delivery system, open-sourced by Netflix in November 2015 [Net15]. Google actively supported the development. Since then, many other companies like Microsoft or Oracle have joined the Spinnaker community. Key focus of Spinnaker is to decouple deployment activities from a certain cloud provider. The deployment to provider A should work the same way as to provider B. As of today AWS EC2, Kubernetes, Google Compute Engine, Google Kubernetes Engine, Google App Engine, Microsoft Azure, and Openstack are supported, with Oracle Bare Metal and DC/OS support coming soon [Netd]. Spinnaker thereby follows Continuous Delivery best practices like immutable infrastructure and usage of deployment strategies (cf. section 2.2.5). Spinnaker originated from Netflix's previous cloud-deployment focused delivery system, Asgard ([Net12]). The Netflix team noticed that Asgard was not flexible and extensible enough to support their growing needs [Net12], thus they chose to design Spinnaker using the microservice architectural style. Overall, spinnaker consists of the following microservices [Neta]:

Deck is Spinnaker's browser-based frontend

Gate is the API gateway of Spinnaker. It is the entrypoint to Spinnaker and mediates requests of Deck and all other api consumers to the corresponding microservice.

Orca is Spinnaker's orchestration engine

Clouddriver keeps track of all deployed resources and realizes the interaction with cloud providers

Front50 is Spinnaker's metadata store. It persists all relevant metadata (application, delivery model, project and notifation metadata).

Rosco produces machine images for the bake stage. As of today, it wraps packer [1]

[1]https://www.packer.io/

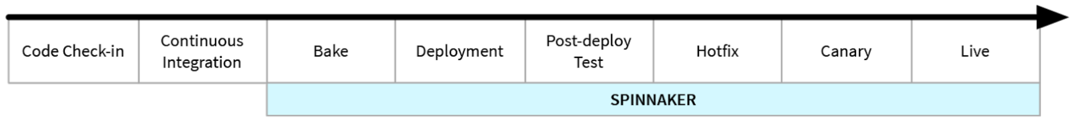


Figure 4.1.: Spinnaker Continuous Delivery (from [Net15])

Igor integrates Jenkins, Travis and Git repositories into Spinnaker. It triggers the delivery process in Spinnaker whenever something has changes remotely.

Echo is the eventing bus of Spinnaker. Igor sends information to Echo if something has changed. Echo is also used for sending notifications to Slack, Hipchat, etc.

Fiat is Spinnaker's identity service. It handles user authentication and authorization.

Halyard is Spinnaker's configuration service. It provides a command line interface for configuring, installing and updating Spinnaker's services.

Spinnaker handles only a part of the delivery process. As figure 4.1 depicts, it is deployment-focused. Therefore a dedicated Continuous Integration Tool (e.g. Jenkins) is required to handle the build stage. Having a binary in place, Spinnaker then handles the remaining activities. The atomic building blocks of a delivery model are called *stages* in Spinnaker. Stages can be sequenced in any order. A stage encapsulates functionality. Users select and parametrize available stages using Spinnaker's web-interface to describe the delivery process. Beside providing abstraction, stages are the means to extend Spinnakers functionality. Developers can create new stages by implementing a new stage class in Orca, by adapting Clouddriver if required and by making front-end changes to Deck. In Deck developers basically provide a view for configuring instances of this new stage. This view is required as Spinnaker delivery models are configured either through the UI or via API calls. The netflix team is actively working on a declarative pipeline description language [Rob17] to allow for *pipeline as code*, but as of today this is still a beta feature.

4.1.1. Requirement Fulfillment

The following briefly discusses which of our architecture requirementd (section 3.2) are met by Spinnaker.

Req-1: Integration of new technology

Rating Fulfilled (+)

Rationale Spinnaker extension mechanism is similar to a plugin-based architecture. Developers need to realize custom stages, which can be used to add new functionality. The functionality needs to be realized in Java and is limited to the execution environment of Orca. Basically, the primarily focus are cloud-provider related operations. All in all, the integration of new technologies is possible in theory, although it might be challenging depending on the use case.

Req-2: Modularity

Rating Fulfilled (+)

Rationale The Netflix team purposefully designed Spinnaker as a Microservice architecture since their previous cloud-deployment focused delivery system was not flexible and extensible enough. Considering the microservice architecture, Spinnaker fullfills the modularity requirement.

Req-6: Model Validation

Rating Hardly fullfilled (−)

Rationale The design of Spinnaker indirectly allows for some validation. Primarily, the delivery process, respectively its stages are configured graphically. The configuration frontend is realized by developers who also implemented the stage functionality. Thus, they can define which input values are acceptable through the user interface. This allows for some basic validation support. But a delivery process can also be configured via API calls. Then, an invalid input parameter would only be detected during execution time. In addition, the compatibility of stages is not validated. So overall, the validation requirement is hardly met.

Req-3: Abstraction

Rating Fulfilled (+)

Rationale Key abstraction are stages in Spinnaker. They encapsulate all technical details to realize cloud provider deployment functionality. Users only need to configure required input parameter, which represent the interface of a stage. Thus, we consider the abstraction requirement to be fulfilled.

Req-4: Self-Organizing

Rating Not fulfilled at all (−−)

Rationale Users need to configure the delivery process stages with all required information. Spinnaker executes these stages with the exact given config and in the exact specified order. Spinnakers design does not allow to dynamically adapt the delivery model.

Req-7: Best practices

Rating Partially Fullfilled (○)

Rationale Spinnaker follows many Continuous Delivery best practices like usage of deployment strategies (e.g. red / black or canary) or immutable infrastructure. But as introduced, so far it has no support for pipeline as code. This violates the best practice to keep everything in version control, as the delivery model is stored internally.

Req-5: Custom PDLs

Rating Not fulfilled at all (−)

Rationale Spinnaker primarily allows to define the delivery process via its GUI. Thus, there only is one (graphical) pipeline description language. In theory, a delivery process can be defined through API calls, which would allow to implement a dedicated software entity that translates custom pipeline descriptions languages. But Spinnaker does not provide such entity, thus we consider this requirement not to be fulfilled.

Req-8: Traceability

Rating Fullfilled (+)

Rationale Traceability in the delivery system context requires to be able to follow what has happened and what the result was. Spinnaker visualizes every step performed and provides means to access the results including produced artifacts. In addition, ts uses Echo to publish important events which aids traceability.

4.2. Concourse

Concourse [Pivd] is a Pivotal sponsored delivery system. Its focus is to be simple and scalable. Thereby, it follows the *pipeline as code* principle. Each delivery process is defined in a single declarative delivery model file. Under the hood, concourse natively uses docker to encapsulates and run all delivery process activities. Overall concourse uses three core concepts [Piva]:

Resource Key abstraction in Concourse are resources. A resource is *any entity that can be checked for new versions, pulled down at a specific version, and/or pushed up to idempotently create new versions* [Piva]. Thus, resources are Concourse's concept to interact with the outside world (e.g. with Version Control Systems or notification tools). They are used to model all external dependencies that enter or exit the delivery process. Each resource must implement a generic interface by means of a container image with three scripts: a *check* script to check for new versions of the resource, an *in* script to download the resource and an *out* script for upload a new version of the resource. Each script basically needs to accept arbitrary JSON-objects (source & version for the check script, source & version & params for the in script and source & params for the out script). Out of the box, concourse officially supports 13 resource types ranging from version control over a time resource for scheduling to S3 buckets. But there are many more community resource types and custom resource types can also be implemented.

Task Tasks are the atomic unit of work in Concourse delivery process. Conceptually, they define a function from input to output. Each task references a shell script to execute in a docker container. Thereby, Concourse pulls the required input resources and mounts them into the docker container working directory. The task script then needs to produce its output in the corresponding output directory also mounted by Concourse.

Job Jobs define actions to execute when dependent resources change. They orchestrate multiple tasks and their depending resources.

Using these three concepts, listing 4.1 provides an example delivery model. It contains a time resource, which emits a new version every minute. The greet job is triggered by this resource and performs the greet task. We defined the task inline. Concourse also allows to define tasks in a separate .yml file, which the job definition then references.

```yaml
resources:
- name: every-1m
  type: time
  source: {interval: 1m}

jobs:
- name: greet
  plan:
  - get: every-1m
    trigger: true
  - task: greet
    config:
      platform: linux
      image_resource:
        type: docker-image
        source: {repository: ubuntu}
      run:
        path: echo
        args: ["Hello World"]
```

Source Code 4.1: Concourse PDL

Defining tasks separately has the advantage of being able to trigger tasks outside the delivery process execution context via Concourse Command Line Interface (see FlyCLI [Piva]).

Since resources are used to model the artifact flow through a delivery process, users do not need to explicitly define the execution order or job dependencies. Instead, Concourse automatically schedules the execution based on the job's resource dependencies.

4.2.1. Requirement Fulfillment

The following briefly discusses which of our top level requirement (section 3.2) are met by Concourse.

Req-1: Integration of new technology

Rating Fulfilled (+)

Rationale Concourse natively uses docker which allows to integrate almost arbitrary technologies by defining the corresponding docker image. The integration of new technologies for the task execution (e.g. new build tools) then is just a matter of providing a docker image which contains the build tool. For the integration of new resource types, one has to provide a docker image which implements the generic resource interface.

Req-2: Modularity

Rating Partially Fulfilled (○)

Rationale In principle, Concourse is build up from two subsystems. One subsystem is the web subsystem comprising the ATC and the TSA. The ATC (*"air traffic control"* [Piva])orchestrates the delivery processes across workers and provides an web-based UI. The TSA is a custom-built SSH server for registering workers with the ATC. The second subsystems are workers which provide the docker container runtime and cache management. Internally, a worker contains Garden for the container runtime management and Baggageclaim for resource caching. Overall, Concourse divides responsibilities into several subsystems and components. But these components do not have the granularity to easily extend or modify the architecture (e.g. when comparing the architecture to the microservice architecture of Spinnaker (cf. section 4.1)). We therefore consider the modularity requirement to be partially fulfilled.

Req-6: Model Validation

Rating Hardly fulfilled (−)

Rationale The concourse command line interface provides a *validate-pipeline* command to validate a given delivery model. The validation primarily comprises syntactic checks. Since tasks perform arbitrary shell scripts and the resource interface (check, in and out) accepts arbitrary parameters no functional validation can be performed. Thus, concourse hardly fulfills the validation requirement.

Req-3: Abstraction

Rating Partially Fulfilled (○)

Rationale At its heart, Concourse is built upon three concepts. Resources, jobs and tasks. Resources are the central abstraction to encapsulate the interaction with external systems. Users do not need to deal with all their details. Instead, they can use the generic resource interface. Task definitions on the other hand contain raw shell commands [2]. Thus, we consider the abstraction requirement to be partly fulfilled.

Req-4: Self-Organizing

Rating Hardly fulfilled (−)

Rationale Concourse automatically plans the delivery process job execution order based on the jobs resource dependencies. So users do not need to define an explicit order which is a first step into fulfilling the Self-Organizing requirements, but not more.

Req-7: Best practices

Rating Partially Fullfilled (○)

Rationale Concourse Resource abstraction allows to easily follow some Continuous Delivery best practices like build once. But Concourse does not enforce these practices. It is up to users to model their delivery process accordingly. But the resource abstraction is a measure in the right direction. We therefore consider this requirement to be partly fulfilled.

Req-5: Custom PDLs

Rating Not fulfilled at all (−−)

Rationale Concourse does not support custom descriptions languages. However, it could be easily extended to support other description languages as it relies on pipeline as code.

Req-8: Traceability

Rating Partially Fullfilled (○)

Rationale Concourse Resources are the only way to archive flow in the delivery process. These resources can be decorated with arbitrary meta data for traceability. The Concourse UI provides a graph-based UI to visualize this flow and meta data.

[2] or a reference to a shell script. But as these scripts need to be part of the project, there basically is no information hiding

But as Concourse does not provide an artifact store, it's up to users to store their artifacts somewhere externally, which hinders traceability in the delivery process context. Overall, we consider the requirement to be partially fulfilled.

4.3. GitLab CI/CD

Initially developed as a web-based git repository manager, Gitlab evolves to - as they call themselves - *the leading integrated product for the entire software development lifecycle* [Gita]. In Q3 2017 the Forrester Wave report elected GitLab as the leading Continuous Integration Tools [Git17]. Beside a git repository, wiki and issue tracking features, GitLab also offers an integrated delivery solution - GitLab CI / CD.

Delivery Processes in GitLab CI / CD are defined declaratively via pipeline as code. Thereby, GitLab expects a gitlab-ci.yml file in the root path of the GitLab project. The delivery model contains a set of jobs with constraints specifying certain execution conditions (e.g. only execute on master branch). Per job shell scripts are defined, which are executed sequentially. To execute such a job, Gitlab uses several independent GitLab Runners. These runners are external agents that register themselves at GitLab. Each Runner can be configured to use one or several executors. Among the available executors are a shell executor, a docker executor, a VirtualBox executor, a kubernetes executor or a ssh executor. The delivery process activities are executed in the selected executor context. Thus, executors provide means to integrate new technology. In case of the docker executor arbitrary docker images can be used to integrate new technologies. In case of the shell executor, new technologies must be installed in this runners context.

Listing 4.2 provides an example gitlab delivery model. It comprises two stages: compile and test. Stages are executed sequentially. A stage then comprises jobs which are executed in parallel. Listing 4.2 defines two jobs: compileDummy and testDummy. Each job definition contains some shell commands under the script directive. The compileDummy job also contains an image definition and a tag. Tags are used to constrain the executor selection. In this concrete case, only executors with the docker tag should execute the job. Since the tags corresponds to a docker executor, the job definition requires an image definition (ubuntu:17.04). The compileDummy job also contains an artifacts section defining file system locations to be stored by the executor on completion of the job execution. GitLab automatically provides depending jobs with the specified artifacts (e.g. the testDummy job).

```
stages:
  - compile
  - test

compileDummy:
  image: ubuntu:17.04
  stage: compile
  script:
    - echo 'Hello World' > hello.txt
  artifacts:
    paths:
      - hello.txt
  tags:
    - docker

testDummy:
  stage: test
  dependencies:
    - compileDummy
  script:
    - cat hello.txt | grep -q 'Hello world'
```

Source Code 4.2: Gitlab - Pipeline Description Language

With release of version 10.0 GitLab introduced a beta feature called **Auto DevOps**. Auto DevOps is their vision of automatically building, testing, deploying and monitoring applications with minimal to zero configuration [GIt]. The idea is to automatically detect the project technologies and to generate an opinionated delivery process based on best practices. Thereby, 7 stages are generated:

Auto Build Auto build bakes the application into a docker image. Either there is a Dockerfile in the project's root directory or GitLab will leverage Herokuish [3] which supports all languages and frameworks that are available as Heroku buildpacks like, for example, Java, Python or Node. Gitlab also supports to use a custom buildpack.

Auto Test Auto tests also leverages Herokuish to run tests appropriate for the project. The tests to be performed must be part of the project. GitLab only uses Herokuish to determine which tests to execute.

Auto Code Quality The auto code quality stage runs the codeclimate docker image [4] which performs static analysis on the projects source code. The report is thereby uploaded as an artifact to GitLab for later usage and analysis.

Auto SAST The auto static application security testing (SAST) stage runs a custom

[3]https://github.com/gliderlabs/herokuish
[4]https://hub.docker.com/r/codeclimate/codeclimate/

gitlab docker image [5] to check for potential security issues using static analysis. The image supports JavaScript, Python, Ruby and Ruby on Rails. Its reports are uploaded as an artifact to Gitlab for later usage and analysis.

Auto Review Apps Auto Review Apps is a stage that uses Kubernetes to temporarily run the docker image built in the build stage. Thereby, each started container is unique accessible. When the corresponding project branch is deleted (e.g. after the merge request), the auto review app stage deletes the container.

Auto Deploy When changes are merged to the master branch, the auto deploy stage uses Kubernetes to deploy the application to a production environment.

Auto Monitoring After deploying the application, auto monitoring allows to collect certain metrics from the running application. The auto monitoring stage thereby uses Prometheus to access Kubernetes metrics directly.

Overall, the Auto DevOps feature is based on a delivery model template. To adapt the delivery process, one can change the template or copy it directly to the projects root directory for usage as a normal gitlab delivery model.

4.3.1. Requirement Fulfillment

The following briefly discusses which of our top level requirement (section 3.2) are met by Gitlab CI / CD.

Req-1: Integration of new technology

Rating Fulfilled (+)

Rationale Depending on the executor used for executing jobs, the integration of new technology is met in different ways. The docker executor or the kubernetes executor allow to integrate new technologies by using a docker image which supports the technology. Both the shell executor and the ssh executor require to install new tools or required dependencies in the runner's context or the ssh context, respectively, to be accessible via shell commands. The virtual-box executor requires a corresponding virtual machine which supports the new technology.

Req-2: Modularity

Rating Hardly Fulfilled (−)

Rationale The GitLab architecture primarily comprises the GitLab Workhouse, Gitaly and GitLab Pages (cf. [Gitb]). They communicate via Unix sockets. We do not consider the Gitlab coarse-grained architecture being modular.

[5]gl-sast: https://gitlab.com/gitlab-org/gl-sast

Req-6: Model Validation

Rating Not fulfilled at all (−−)

Rationale The only validation support provided by Gitlab is by means of a model linter, that allows to validate the delivery model syntactically. But as the delivery model contains arbitrary script sequences, the validation is restricted to basic syntax checks, which we consider as not fulfilling the requirement.

Req-3: Abstraction

Rating Hardly fulfilled (−)

Rationale GitLab barely provides any abstractions. Users need to define almost all delivery process details. Since the pipeline description language is declarative, at least some aspects do not need to be defined. Therefore we consider the abstraction requirement to be hardly fulfilled.

Req-4: Self-Organizing

Rating Hardly fulfilled (−)

Rationale Based on production-ready features GitLab does not offer any Self-Organizing capabilities at all. Users even need to define the execution order in terms of stages and jobs. There is a beta-feature called Auto DevOps (see above). But this feature basically provides an opinionated delivery model as a template. There is no smartness or organization involved. But still, this feature indicates the need for Self-Organizing or assisting the modeling. Since Auto DevOps at least eases the modeling a little bit, we consider the requirement to be hardly fulfilled.

Req-7: Best practices

Rating Hardly fulfilled (−)

Rationale Since GitLab hardly provides any abstraction, it's up to users to define their delivery process in such a way that follows best practices. The Auto DevOps feature at least provides hints how to structure a delivery process. But overall, the requirement is hardly fulfilled.

Req-5: Custom PDLs

Rating Not fulfilled at all (−−)

Rationale GitLab does not support custom pipeline description languages. Though, it could be easily extended to support other languages as it relies on pipeline as code.

Req-8: Traceability

Rating Partially fulfilled (○)

Rationale GitLab visualizes every job and stage in the delivery process. GitLab runners provide logs of currently running jobs almost instantly. In addition, GitLab stores the result and logs of every job. Using its artifact store, GitLab can also provide the artifact results of a job. But as it is up to users to define those artifacts, some intermediate results might be untracked.

4.4. Summary

This chapter provided an overview of related work. Considering the identified issues of existing delivery systems in chapter 3, we decided to only evaluate bleeding-edge delivery systems against our architecture requirements. Concretely, we evaluated Spinnaker, Concourse and GitLab. Table 4.1 summarizes our findings. All things considered, the presented delivery systems only meet individual requirements. In particular, they have difficulties with the usability related requirements, i.e. Req-4: Self-Organizing, Req-5: Custom PDLs and Req-6: Model Validation.

Requirement	Spinnaker	Concourse	GitLab
Req-1: Integration of new technology	+	+	+
Req-2: Modularity	+	o	−
Req-6: Model Validation	−	−	−−
Req-3: Abstraction	+	o	−
Req-4: Self-Organizing	−−	−	−
Req-7: Best practices	o	o	−
Req-5: Custom PDLs	−−	−−	−−
Req-8: Traceability	+	o	o

−− Not fulfilled at all
Hardly fulfilled
° Partially fulfilled
+ Fulfilled

Table 4.1.: Requirement fulfillment analysis

Interesting though, they seemingly identified our challenges, but yet struggle to tackle them. GitLab for example has a *Auto DevOps* - feature to model the delivery process with minimal to zero configuration, which in some aspects is similar to our idea of self organization. Concourse introduces resources as a central abstraction providing a unified interface, hiding technical details and allowing for *scheduling with resources*. Scheduling with resources thereby is Concourse concept to automatically determine the execution order of activities. This concept also is similar to our idea of self organization. Spinnaker relies on microservices to tackle the evolution challenge. Thereby it encapsulates the realization of delivery process activities in dedicated software entities (abstraction). Because of Spinnakers internal structure, these activities then allow for validation at least at the User Interface level. But as the evaluation shows, neither Spinnaker, Concourse or GitLab tackle the challenges adequately. To allow for profound self-organization and validation support, they would require great conceptual and architectural changes. To this reason we think a custom development is reasonable, which the following chapters introduce. Since we primarily need to operate on green-field, we use a Domain-Driven Design approach in the next chapter to gain a solid domain understanding.

5. Domain Driven Design

Contents

Chapter 3 identified challenges delivery systems face in the context of build design. The identified challenges are C1 - Project Evolution and C2 - Modeling Usability. Among the reasons for these challenges is complexity (see section 3.1). The key to controlling (business) complexity is a good domain model ([Eva03]). Based on the principles of Domain-Driven Design [Eva03], the next section therefore introduces the *core domain* and other important subdomains (cf. [Eva03]). With a solid domain understanding chapter 6 then introduces our architecture trying to meet the requirements and tackle the aforementioned challenges.

5.1. Strategic Design

In Domain Driven Design, Strategic Design provides bigger-picture semantics. It highlights what is strategically important to the business, how to divide work by importance and how to best integrate (cf. [Ver16]). In section 5.1.1 we therefore distill the core of our problem domain, then divide the domain by importance (section 5.1.3) and define the integration between these boundaries (section 5.1.4). We then address the technical aspects of strategic design (cf. [Ver16]) and provide resulting, high-level design decisions in section 5.1.5.

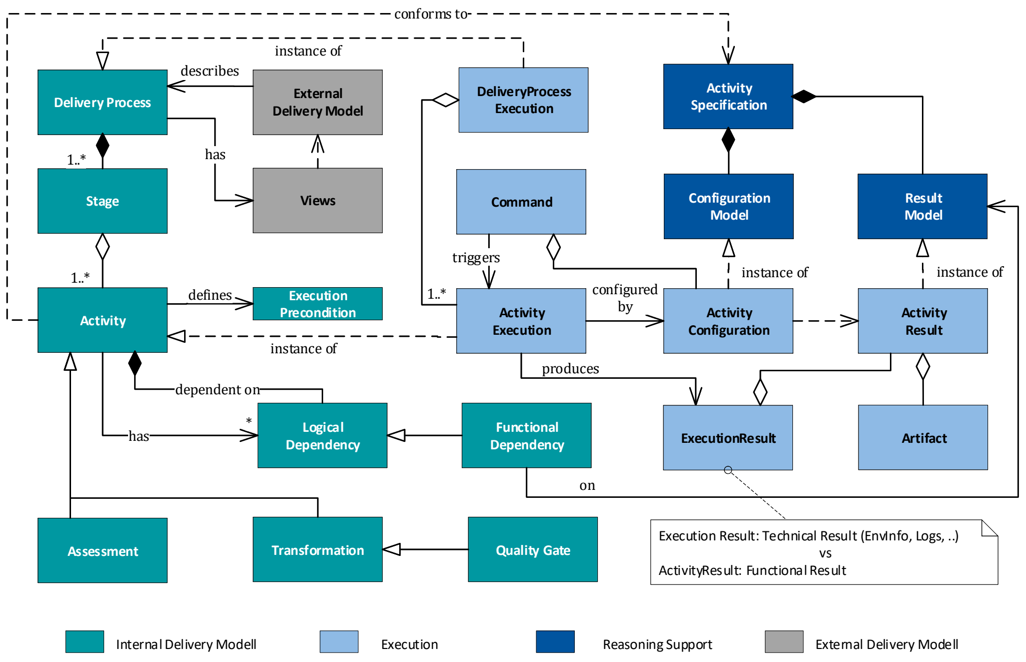


Figure 5.1.: Delivery System Core Domain

5.1.1. Core Domain

Central aim of a domain model is to be an asset in solving and understanding a business problem [Eva03]. As a full-blown domain model for a delivery system would not provide value because of it's complexity, we distill the core, i.e. the essential part of the problem domain (cf. [Eva03]). Following our challenges and the thesis scope (see section 3.1), two aspects are of great importance. First to identify stable concepts that allow for extension (C1 - Project Evolution and second to provide means for reasoning about a delivery model (C2 - Modeling Usability).

Figure 5.1 depicts the core domain. We modeled few cardinalities as we focus on the general concept relations in this chapter. We developed the core domain iteratively starting from our concepts described in section 2.3. Simplifying the core domain figure, it consists of four areas. The turquoise area (Internal Delivery Model) depicts our internal representation of a delivery process. The light-blue area (Execution) is about the delivery process execution. The dark-blue area (Reasoning support) represents concepts for reasoning about a modeled delivery process and the gray area (external delivery model) depicts concepts for the external delivery model. The following details the concepts according to their area in the following order: internal delivery model, reasoning support, execution, external delivery model. Since Section 2.3 already introduced the Delivery Process and stages, we do not detail them here. For the Delivery Process Aggregate we refer to section 5.2.1.

Activity

The units of work in the delivery process are called *activities* (cf. section 2.3). As the project evolution influences the delivery process (cf. chapter 3), the delivery system must be both flexible and maintainable. To design a flexible and maintainable delivery system (C1 - Project Evolution) while yet being able to reason about the delivery process (C2 - Modeling Usability) requires to classify the different activities in a flexible but also meaningful manner. Section 2.4 provided such an activity classification. We modify it as follows:

No fan in/out : J.Hermanns introduced *fan in/out* activities to parallelize the execution. We argue that these technical activities are necessary, but not as part of the core domain. The core domain focuses on the functional relations between activities and their validation. Execution constraints can be easily deduced via activity dependencies (cf. section 5.1.1).

Assessment instead of Measurement : Conceptually, our assessment activity type is identical to Hermanns measurement. Hermanns defined measurement as activities that analyze the degree to which their input artifact possesses a attribute. This is the characteristic of a metric ("*a quantitative measure of the degree to which a system, component, or process possesses a given attribute*" [IEE02]). A metric can either be a base metric, i.e. independent of others, or a derived metric, i.e. combination of multiple base metric quantify attributes that cannot be directly measured [LL10]. The application of a metric is called measurement [Via16]. Thus, this activity type performs measurements to be able to make a statement about certain attributes of the input artifact. Overall, we therefore think that assessment is more adequate to express this activity type.

Quality Gate as a special transformation : Hermanns defined quality gates as a control activity, which aborts the execution if the input artifact does not meet certain quality criteria. But quality gates not only abort the execution, i.e. reject artifacts, they also *promote* artifacts in the positive evaluation case. As the quality gate output, i.e. the promoted artifact, conceptually differs from the input artifact, we consider quality gates as a special transformation that alway produces an artifact, although input and promoted artifact are equal on a binary level. With this classification, an exit condition of a delivery process execution would be a missing output artifact, which conceptually matches an interruption of the software delivery value stream [HF10].

Overall, this results in three different activity types, which figure 5.2 summarizes.

Transformations are the core activity type of a delivery process. They take one or multiple artifacts as input and transform these artifacts, i.e. mutate, translate or

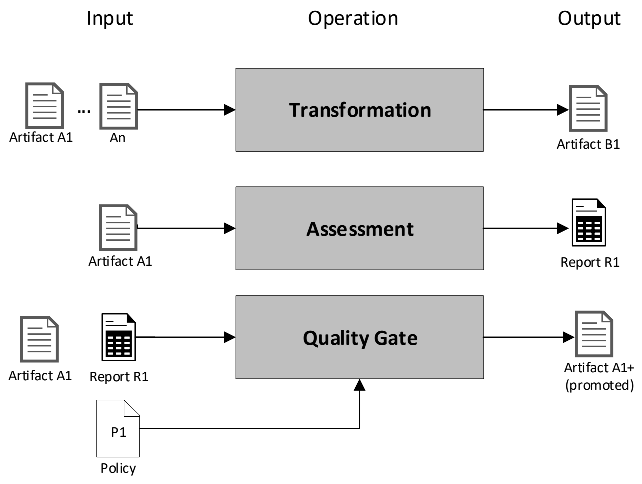


Figure 5.2.: Activity Classification - Overview

merge them into a new artifact, which is the output of the transformation. An example for a typical transformation is *compilation*, which transforms source code into executable machine code. As transformations are the only activity type to participate in the delivery value stream, a delivery process must at least comprise one transformation. Some activities do not require a binary artifact as input to produce another artifact (e.g. checkout from version control) or their result does not reference a binary artifact (e.g. the output of a deployment transformation might just be an URL). Conceptually, we consider these activities still to be a transformation.

Assessments perform measurements to be able to evaluate certain attributes of the input artifact. They publish these results as a report. The assessment realization decides which measurements are performed and to which scale type (allowed metric scale types are ordinal, interval, rational or absolute scale (cf. [LL10])) they are mapped. An example is a unit test assessment which performs unit tests on its input to calculate passed test rate and test coverage. Each assessment accepts exactly one artifact (to keep the quality gate promotion simple) and produces a single report for this artifact. The report might contain multiple measurement results.

Quality Gates represent decision points in the delivery process to ensure quality criteria are met. They promote or reject transformation artifacts either by a manual user approval or automatically based on a given policy and corresponding assessment reports. In case of a manual approval, a user interprets the reports and promotes or accepts the input artifact. In case of an automatic evaluation, the quality gate interprets the result and evaluates if the reported quality characteristics fulfill the expected values specified in a policy. If so, the artifact is promoted. Otherwise, the

Activity	Activity Type	Input	Output
Compile	Transformation	Source Code	Compilation
Unit Tests	Assessment	Compilation	Unit Test Report
Code Analysis (compliance)	Assessment	Source Code / Compilation	Analysis Report
Bake	Transformation	Compilation	Deployable Image
Acceptance Testing	Assessment	Deployed System	Acceptance Report
Integration & System Tests	Assessment	Deployed System	Integration Report
Performance Testing	Assessment	Deployed System	Performance Report
Security Testing	Assessment	Deployed System	Security Report
Environment provisioning	Transformation	Env specification	Provisioned Env
Deploy to Env	Transformation	deployable image	deployed system
User Acceptance Testing	Assessment	Deployed System	User Acceptance Report
Setup operation services	Transformation	Base System	Provisioned System
DNS Modification	Transformation	DNS Spec	Updated DNS
Data Migration	Transformation	Migration Spec	Migrated data
Undeploy from Env	Transformation	Deployed System	Clean system

Table 5.1.: Delivery Process Activities

artifact is rejected and the execution aborted. As the delivery process follows the idea of a stage-gate process, quality gates are typically performed at the end of a stage.

After introducing our different activity types, it remains to show these abstractions are suitable for modeling a delivery process. To this reason table 5.1 classifies delivery process activities identified in section 2.3.1 according to our task classification.

Activity Dependencies

Activities encapsulate delivery process behavior. According to the classification above (section 5.1.1) they either *transform*, *assess* or *promote* artifacts. To represent flow between activities, there are two dependency types:

Logical Dependencies are dealing with the control flow. They express execution conditions. If activity a_2 has a logical dependency on activity a_1, activity a_1 needs to be executed before activity a_2.

Functional Dependencies represent data flow. They realize data passing. After an activity has been performed, its output is passed as input to the next activity. Functional Dependency also implies logical dependency: If activity a_1 has a functional dependency on activity a_2, the output of activity a_1 is passed as input to a_2 and a_2 is logical dependent on a_1.

Execution Precondition

Activities are not necessary side-effect free. A typical example are deployment transformation. It is therefore necessary to be able to reverse the effect of activities or to provide counter measures, e.g. in case of a failure during execution. Such reversal operations could be realized by means of the same activity. But we want to provide more flexibility in freely deciding on counter measures. To this reason, our core domain knows the concept of an *execution precondition* which defines a condition under which the activity is executed. Possible variants are:

onFailure The activity should be performed if the previous activity failed. If there is no previous activity, this activity should not be performed

onSuccess The activity should be performed if the previous activity was successful. If there is no previous activity, this activity should be performed

allways The activity should be performed independently of the result of the previous activity.

In combination with the activity dependency concept, the execution preconditions supports fine-grained control when to execute an activity.

Activity Specification

The delivery system requires detailed information about all available activities in the system to be able to tackle C2 - Modeling Usability. As introduced in the previous section our activity classification differentiates between assessment and transformation activities. Conceptually, they can be both described as functions. Therefore, the delivery system must at least know the interface of its activities to reason about them. The interface of an activity is described through the corresponding activity specification. The activity specification consists of an **activity identifier**, a **configuration model** and a **result model**. The activity identifier allows to uniquely identify activities. The configuration model specifies the admissible configuration range and the result model defines permitted results.

Using the activity specifications the delivery system can reason about the compatibility of consecutive activities in the delivery process. It also allows to automatically calculate functional dependencies between activities (assuming a single activity provides the required value type). The details of the activity specification are provided in section 5.2.5. Section 7.3.1 provides some background on the automatic dependency calculation.

Activity Execution

Activities are units of work in a delivery process. Their performance is represented as a dedicated concept, namely **activity execution**. This explicitness is both advised ([Ver12]) and reasonable as the activity and its execution are conceptually different. In principle, an activity execution is an active instance of an activity. The activity execution is configured by an **activity configuration**, which must be an instance of the activity specifications configuration model. The activity execution produces an **execution result**. The execution result contains both technical information and the functional result. The technical information contain environment information and logs. The functional result, represented as **activity result** in the core domain model must be an instance of the activity specification result model.

Other activities can reference the result of an activity. From a model perspective, this functional dependency references the corresponding result model. During execution, the activity execution gets the referenced result as part of its activity configuration.

Delivery Process Execution

Similar to the activity and activity execution instance-of relation, there is an instance-of relation between the delivery process and the delivery process execution itself. The delivery process execution comprises several related concepts like a state and the activity executions. Section 5.2.1 details the delivery process execution aggregate.

Artifact

An activity execution typically produces one artifact. Thereby, an artifact can be either a binary artifact or a machine interpretable. Our core domain does not differentiate between them, as the artifacts are transparent to the delivery system and only relevant for individual activities. To provide a consistent terminology, we nevertheless introduce **Report** as a specific artifacts here. Reports are machine-interpretable files that are produces by assessments. Quality Gates use Reports for their decision making.

Command

To bridge the gap between activity and activity execution, our core domain provides the command concept. Each command encapsulates all information required to trigger the activity execution. This allows for an unified execution interface as detailed in section 7.5. Section 7.5.2 provides more details about both the activity and its execution.

External Delivery Model

An important Continuous Delivery principle is to keep everything in version control (cf. 2.2.5). This also includes the delivery model (cf. pipeline as code chapter 1). Existing delivery systems use the same delivery model for both modeling and execution concerns, i.e. they directly execute the model provided by the user. This conflicts with Req-3:

Abstraction, it makes Req-6: Model Validation challenging and couples the delivery system to a certain model type making Req-5: Custom PDLs impossible. Our core domain therefore provides the concept of an internal and an external delivery model to separate the modeling and execution concerns.

The internal delivery model represents the delivery process as depicted in the core domain, i.e. a graph-based representation consisting of activities with dependencies on each other. The external model can be an arbitrary model optimized for the concrete use case. The only requirement for the external model is to be translatable into the internal model. Given the quite generic structure of our internal model, this requirement only has a small impact on external models. Section 5.2.1 details the external model aggregate.

Overall, the separation into external and internal model then allows to meet Req-6: Model Validation and Req-3: Abstraction since small, concise external models with an explicit language scope can be defined. Section 8.2 provides an example language to describe external delivery models.

Views

To visualize a delivery process in its external model representation, the core domain introduces different views. Each view corresponds to the representation of the delivery process in a certain model - including the internal model. Although not part of the core domain, we briefly want to mention one aspect here: The delivery process itself is represented by means of the internal delivery model, thus a representation in an external model might be challenging since the internal to external model transformation might be lossy.

Constraints

This section briefly introduces important business rules not possible to express in the UML class-diagram representation of the domain model (figure 5.1). We formulate these rules in OCL ([OAB12]).

At least one transformation A valid delivery model must perform at least *something* to contribute the value stream. As section 5.1.1 elaborated, transformations are the designated activity to supply the value stream. Thus, a minimal delivery process must at least comprise a transformation activity.

```
context Activity
inv: Activity.allInstances()->exists(self.oclIsTypeOf(
        Transformation))
```

Source Code 5.1: At least one transformation must exist

No circular dependencies between activities The internal delivery model uses a graph structure to represent a delivery process. Based on this structure, the execution order of the modeled activities is determined. To be able to determine a valid execution order, the activity graph must be a directed acyclic graph (DAG).

```
context Activity
inv: self->closure(activity |
    activity.dependencies->select(dependentOn))->isUnique(self)
```

Source Code 5.2: No circular dependencies between activities

Logical dependency constrains stage Analogously to the delivery process, stages are executed sequentially. Thus, if activity a_1 has a logical dependency on activity a_2, a_2 must belong to the same stage as a_1 or to a previous stage.

```
context DeliveryProcess
inv: Activity.allInstances()->forAll(activity |
         activity.dependency.dependentOn->forAll(dependentOn |
           self.stages->indexOf(dependentOn.stage)
             <= self.stages->indexOf(activity.stage)
         )
   )
```

Source Code 5.3: Logical dependency constrains stage

5.1.2. Explanatory Model

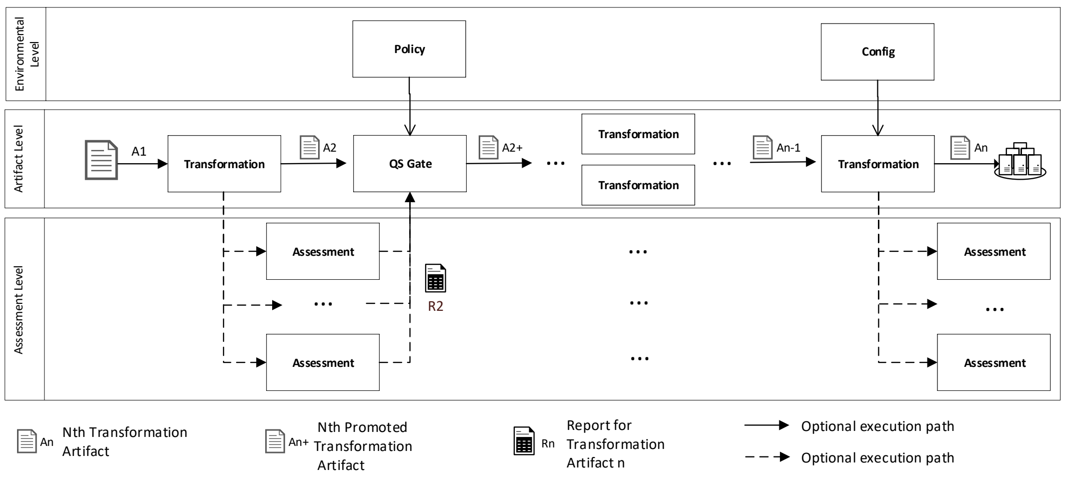


Figure 5.3.: Core Domain - Explanatory Model

To ease understanding of a domain model, Evans suggests to present the concepts in an explanatory model [Eva03]. This model must not correspond in every detail with the domain model [Eva03].

Figure 5.3 provides an explanatory model for our core domain presented above (cf. section 5.1.1). It thereby focuses on the general delivery process building blocks and dynamics. Although they might be represented differently, these elements must be part of every external delivery model.

At its heart the modeled delivery process consists of a series of transformations, which constitute the artifact value stream. The transformation activities can be parametrized with configuration from the environmental level. In case of a quality gates, this configuration typically comprises a policy defining artifact acceptance criteria. Following our activity classification, the quality characteristic of an artifact are evaluated by assessments in the assessment level. Based on their reports, the quality gate decides to promote or reject the artifact. Although not depicted in the explanatory model, an assessment can of also be configured. But as assessments are optional, a minimal delivery process consists of at least one transformation.

Overall, each external delivery model must at least to contain a representation of the presented elements to be mappable to our internal delivery model.

5.1.3. Subdomains

The previous section introduced our core domain for a delivery system. It provided concepts and their relations essential for this thesis scope. Cohesive parts that are required for the full expression of the model are factored out into (supporting) sub domains if they add complexity without communicating specialized knowledge (cf. [Eva03]). The following briefly introduces different subdomain types as defined by Eric Evans. Afterwards we decompose the delivery system domain into subdomains.

Subdomain Types

Evans differentiates three types of subdomains [Eva03]:

Core Domain The core domain reflects the part of the overall problem domain that is central to the purpose of the intended application. It is where the most investment is made as a well-defined core domain provides the means to have an advantage over all competitors.

Supporting Subdomain Supporting subdomains assist the core. The core domain cannot be successful without them. But supporting subdomains do not require such heavy investment as the core domain. They might be outsourced as they are strategically not as important as the subdomain. However, they still require custom development.

Generic Subdomain For generic subdomains an off the shelf solution might be considered. They reflect a modeling situation which requires the least investment amount as they do not require custom development.

Domain Decomposition

Our delivery system domain can be decomposed into five subdomains. Besides the already introduced core domain (see above), the following subdomains exist:

Model Subdomain (Supporting) The core domain differentiates between two delivery model types. An internal delivery model which is contained in the core domain (activities & dependencies) and an external model which the core domains defines as an abstract concept. That is because concepts related to the external model are factored out into a subdomain - the modeling subdomain. Thus, the modeling subdomain is concerned with the import of external models, different external model languages and the translation of external to the internal delivery model. As custom development is required in this domain, the model domain is a supporting subdomain.

Activity Subdomain (Supporting) Key concept in the core domain model are activities. They represent delivery process building blocks. While the concept of an activity and its specification is stable, each concrete activity has several related concepts not relevant for the core domain. If we take a deployment activity to Amazon EC2 for example, there might be concepts like buckets. Another example would be a java compile activity which might have maven specific concepts. Overall, the activity specifics span the activity subdomain, which also is a supporting one as it must be compatible with the core concepts.

Orchestration Subdomain (Generic) The core domain models different execution aspects of delivery process activities. But it does not explicitly introduce process control related aspects, i.e. the orchestration and monitoring of delivery process activities. These aspects are factored out into an orchestration subdomain. Since they represent well-known concepts which are not tight to the delivery system domain, an off the shelf solution can be considered (cf. process manager [HW03]). Thus, the orchestration subdomain is a generic one.

Artifact Storage Subdomain (Generic) The core domain model introduced the concept of an artifact. Related to artifacts are storage concepts. Thus we consider a the artifact storage a dedicated subdomain. As we do not have storage-specific requirements, the artifact storage subdomain is a generic one.

5.1.4. Bounded Contexts

Related to subdomains is the concept of a bounded context. Vernon describes bounded context as *a conceptual boundary where a domain model is applicable* [Ver12]. A bounded context is part of the solution space, i.e. where the software is implemented, while a domain model is part of the problem space, i.e. the business challenge to be solved (cf. [Ver12]). Thus, a domain model can have multiple bounded contexts. But when applying DDD, a bounded context should align one-to-one with a subdomain [Ver16], i.e. to build the software in a way we understood the domain. Sometimes they diverge because legacy software is being used [Lev14]. Considering the greenfield character of our delivery system, the bounded contexts align one-to-one with the aforementioned subdomains. Figure 5.4 visualizes the different bounded contexts and their relations.

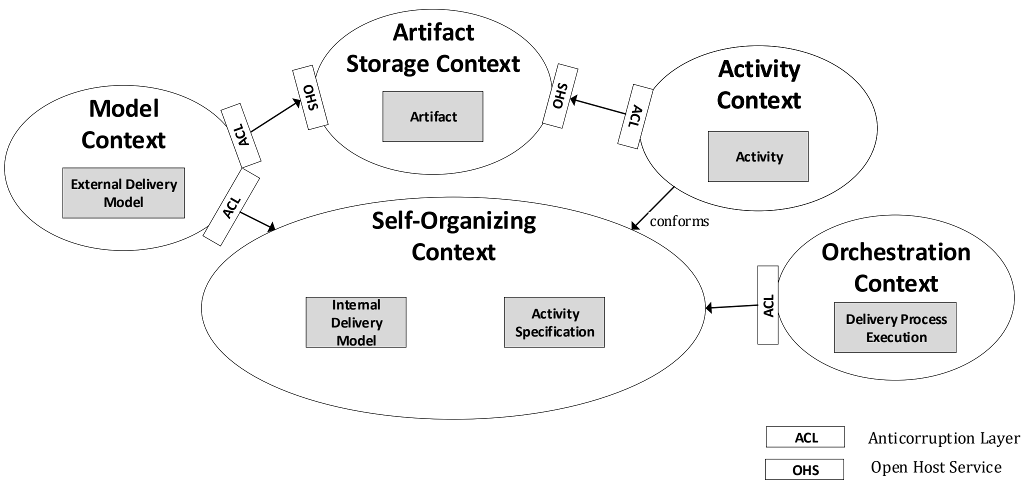


Figure 5.4.: Delivery System - Context Map

Self-Organizing Context The Self-Organizing context encapsulates the core subdomain and ensures its consistency. Components of the self-organizing context realize the activity specification concept and provide the internal delivery model.

Model Context The modeling bounded context defines the boundary of the modeling subdomain and ensures that the language is consistent inside this boundary. For example, the core domain provides the concept of an external delivery model but the details of the external model are out of scope of the core domain. The bounded context thus ensures that the scope and meaning of an external model is consistent inside this boundary. Main responsibility of components in the model context is to import different external models. To make use of these models across the delivery system, the model context integrates with the storage context to be able to store these models. This integration is realized by means of an anti-corruption layer [Eva03], that translates the language of the storage context into the language of the model context. In addition, the model context integrates with the self-organizing context to provide the internal delivery model by translating its external model via an anti-corruption layer.

Activity Context The activity context encapsulates the activity subdomain. Components in the activity context realize the activity concept. The activity context heavily rely on core concepts, e.g. on the activity specification. Because of this close relation and because the activity context has no special requirements regarding the core concepts we modeled the relationship between the self-organizing context and the activity context as a conformist relation [Eva03] to have a reasonable implementation effort. A conformist relation indicates that the activity context uses the activity specification and other core models as they are. Since components in the activity context realize activities, they also need to deal with storage concerns (retrieval or storage of artifacts produced during an activity execution). This integration is realized by a more defensive approach, namely an anti-corruption layer as the activity context should not deal with all the storage model details.

Orchestration Context The orchestration context is the conceptual boundary of the orchestration subdomain. Its components realize process-control related aspects, i.e. the monitoring and triggering of executions. The underlying domain is a generic subdomain. Thus, the language of the orchestration context is translated to the language of the self-organizing context via an anti-corruption layer.

Artifact Storage Context The artifact storage context defines the boundary where the storage domain is applicable. Its components offer services related to the storage and retrieval of artifacts. The model context and the activity context both integrate with the storage context through an anti-corruption layer.

5.1.5. Resulting Design Decisions

The previous section introduced several bounded contexts. This section provides an overview of top-level design decisions that help to cleanly map these boundaries to the delivery system.

Microservices

Given the focus on providing an framework architecture focusing on flexibility and maintainability (cf. section 3.1.1), it is important to provide strong boundaries enforcing autonomy. Because a monolithic architecture can only provide logical boundaries, we want our delivery system to be polylithic. It should be decomposed by means of cohesive services that reflect our bounded contexts. This approach is known as *microservice architectural style.* Martin Fowler and James describe it as [MJ14]:

> In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

This architectural style naturally fits with important design principles that improve both flexibility and maintainability when applied. These principles are summarized under the acronym SOLID. SOLID subsumes the Single Responsibility Principle, the Open-Closed Principle, Liskov Substitution Principle, the Interface Segregation principle and the Dependency Inversion principle [Bai09]. Although these principles originate from object-oriented programming, some of them can be applied to (macro) architecture.

James Hugh describes the architecture when applying these princinples exactly as a microservice architecture [Jam13]:

> Micro Service Architecture is an architectural concept that [...]. Think of it as applying many of the principles of SOLID at an architectural level, instead of classes you've got services.

Overall, the microservice architectural style seems to be a natural fit for tackling our requirements. Another key advantage of a microservice-based architecture is its support of heterogeneous technology. Each service can be realized with the technologies best suited for its task. This allows to meet Req-1: Integration of new technology in a way that supports maintainability and flexibility. Section 5.2.5 provides more details about this decomposition. The general rationale is that the functionality offered by a single tool is closely related while the functionality across different tools has little in common. Thus, introducing a microservice per tool leads to high cohesion in a service and low coupling across services, which supports our goals of maintainability and flexibility.

Chapter 6 provides an overview of the architecture decomposed by means of microservices.

Messaging

Section 5.1.4 provided a context map detailing the different bounded contexts. Some of these contexts have integration points. In the context of DDD basically three [1] integration types are possible: Remote procedure call (RPC), RESTful HTTP and messaging [Ver16]. These communication style can be differentiated by two dimensions. One dimension is about the nature of the protocol, i.e. if it is synchronous or asynchronous. The other dimension reflects the receiver semantics, i.e. if there is a single or if there are multiple receivers. RPC is synchronous with a single receiver, RESTful HTTP has a single receiver and is typically synchronous, but can also be realized asynchronously. Messaging is asynchronous with multiple receivers.

Communication with a single receiver couples sender and receiver. The sender needs to know the receivers location and depends on its availability. In addition, it is difficult to introduce changes or extension. In the delivery domain many processes could also be long-running. Given the focus on flexibility and maintainability, we therefore rely on messaging [HW03] for the integration between the context reflecting the core domain and the other dependent contexts. In the context of DDD the messages are Domain Events [Ver16]. Section 5.2.2 provides more details about the domain events.

For the communication between components of either the model context or the activity and the storage context, we decided to use RESTful HTTP as the underlying storage domain is a generic one and off the shelf solutions typically offer a REST-based interface.

[1] file system or database integration are considered bad practice

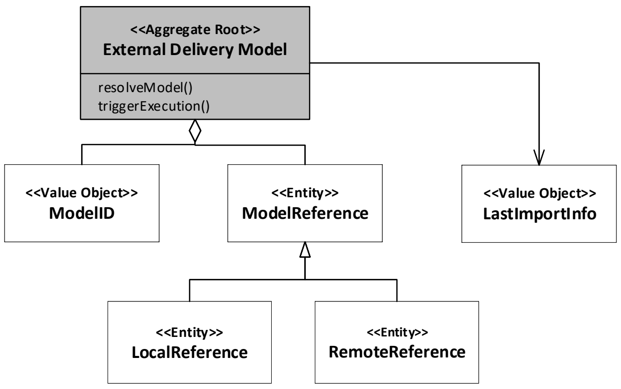


Figure 5.5.: External Delivery Model Aggregate

5.2. Tactical Design

After the previous section provided a coarse-grained overview of important concepts and their relation, this section details these concepts by using an important DDD tool: the aggregate pattern (see section 5.2.1). Since DDD is all about modeling the domain as explicit as possible (cf. [Ver16]), section 5.2.2 introduces Domain Events that help to model explicitly and share business-relevant incidents. Subsequently, section 5.2.3, section 5.2.4, Section 7.5.2, section 5.2.5 and section 5.2.5 introduce more fine-grained design decisions.

5.2.1. Aggregates

An aggregates in DDD is a pattern that clusters related domain objects [Mar]. Each aggregate has a single root object which might be referenced by other objects in the domain. Any access must go through the aggregate root. That way, the aggregate root can ensure consistency of the whole aggregate. Following the DDD terminology, an aggregate is composed of entities and value objects [Eva03]. An entity represents an individual thing that has a unique identity. Often an entity is mutable and changes state over time. A value objects represents an immutable encapsulation of attributes. Its identity is determined by comparing the attribute values. Typically, entities are composed of value objects. [Ver16].

This sections introduces important aggregates from the core domain in the following.

External Delivery Model Aggregate

The external delivery model aggregate clusters domain objects related to the external delivery model concept as presented in the core domain (cf. section 5.1.1). Figure 5.5 depicts the aggregate root with its related domain object. Since the delivery system follows Continuous Delivery best practices (Req-7: Best practices) to keep everything

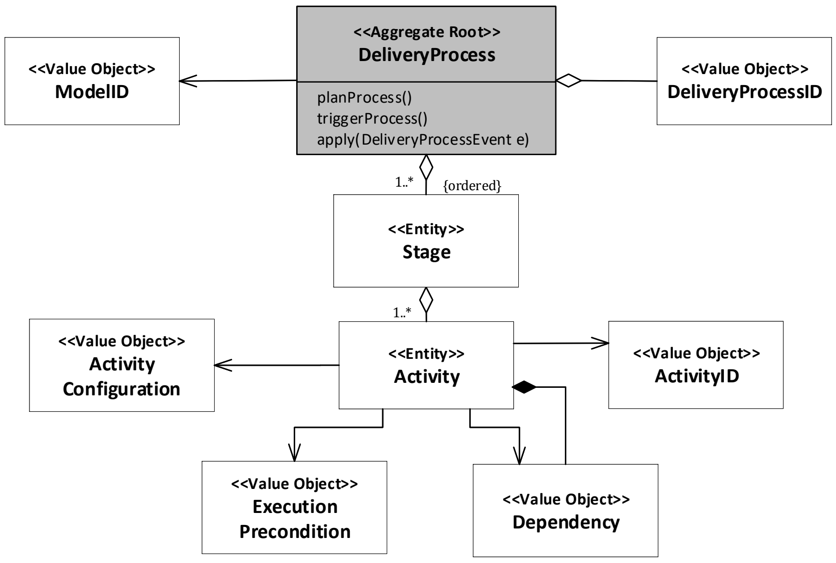


Figure 5.6.: Delivery Process Aggregate

in version control, the external model aggregate primarily consists of a *modelReference* entity. The modelReference expresses that external delivery models are stored somewhere else and not necessarily inside the delivery system context. The reference can either be a *LocalReference* meaning that the external delivery model is stored somewhere natively accessible by the delivery system (e.g. file system) or it is a *RemoteReference* expressing that application services are required to retrieve the model. Central responsibility of the external model aggregate is to retrieve the external delivery model. The external delivery model describes a concrete delivery process. As the external model is not controlled by the delivery system, the resulting delivery process might be different on every import of the external model. Thus, we decided to invert the control and enable the model aggregate to trigger the process execution by providing the external model. Section 7.2 provides more details about the inversion of control and the external model aggregate.

Delivery Process Aggregate

Figure 5.6 depicts the delivery process aggregate. It clusters domain objects that relate to the internal delivery model as presented in the core domain (cf. section 5.1.1). To this reason, we do not detail the concepts here. Important about the delivery process aggregate is that it is only concerned with the static, modeling-related aspects. A dedicated aggregate - the delivery process execution aggregate - deals with the execution of the delivery process. This clear separation of modeling and execution concerns improves maintainability and allows for greater flexibility. To allow for Req-8: Traceability the delivery process aggregate has a reference on the external model aggregate. Also, each

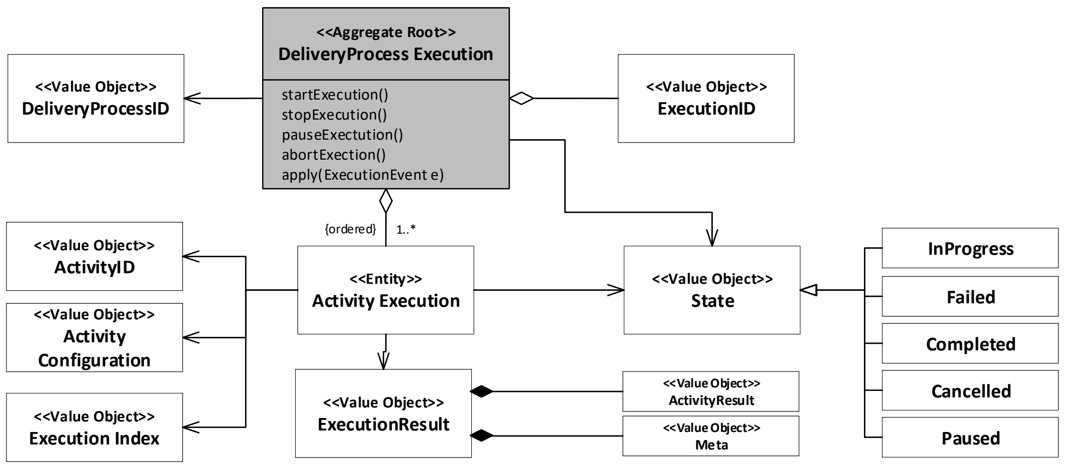


Figure 5.7.: Delivery Process Execution Aggregate

activity in the delivery process aggregate references the activity specification via the *activityId*. This allows the delivery process aggregate to meet Req-4: Self-Organizing. Section 7.3 details this model optimization.

Delivery Process Execution Aggregate

The delivery process execution aggregate is depicted in figure 5.7. It encapsulates domain objects related to the delivery process execution concept as presented in the core domain (cf. section 5.1.1) and ensures their consistency. Central part of the delivery process execution is the activity execution. We designed it as an entity and not a separate aggregate to avoid eventual consistency issues. Both the delivery process execution and the activity execution have a state. The different states already indicate what business operations on the delivery process execution are possible: The execution can be started, stopped, paused and aborted. Similar to the activity entity in the delivery process aggregate, the activity execution comprises configuration values and references the corresponding activity through the activityId. The delivery process execution should not be concerned with logically structures like stages, therefore it only comprises activity executions. The order when to execute an activity is defined through the execution index. This allows to execute activities in parallel while still providing a flexible data structure. Each executed activity has a result which comprises meta data (e.g. logs, metric). If the activity did not fail, its result additionally contains an output value object. Section 7.4 details these execution aspects.

5.2.2. Domain Events

Section 5.1.5 motivated the use of messaging to integrate our bounded contexts. In DDD the messages are *Domain Events* [Eva03], i.e. the integration of bounded contexts is

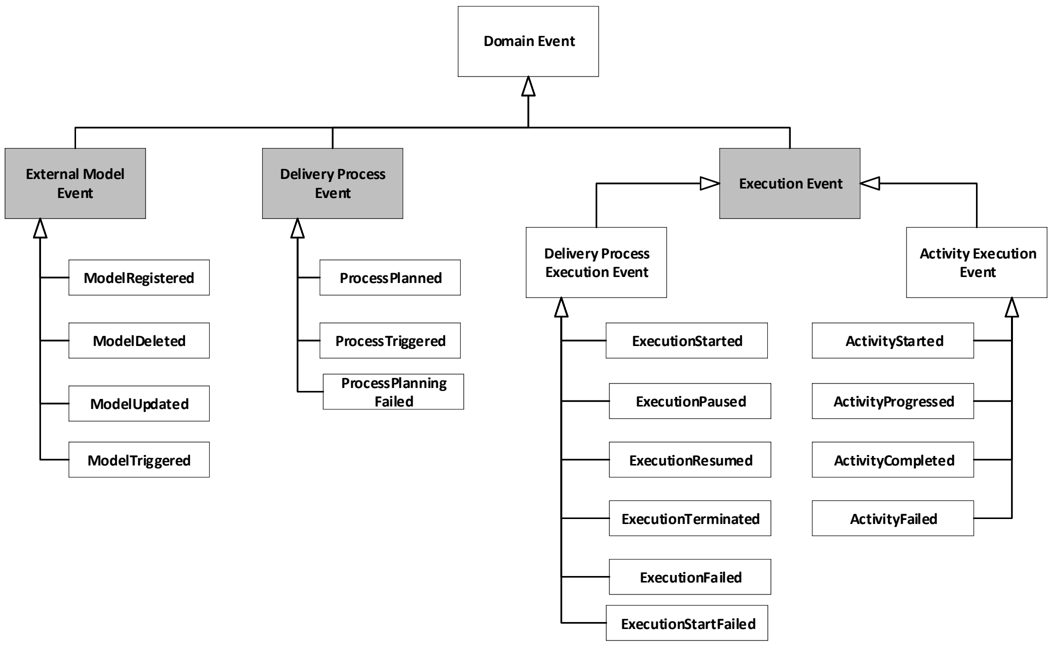


Figure 5.8.: Domain Event Hierachy

realized by means of event notification (cf. Observer Pattern [Gam+02]). Vernon defines a Domain Event as *"a record of some business-significant occurence in a Bounded Context"* [Ver16]. Business-relevant processes are realized by means of aggregates, thus aggregates produce domain events. Interesting parties receive domain events and trigger an action in response. These actions are triggered by means of commands. Semantically, the difference between a command and an event is historical. A command expresses the intention to perform something, while an event indicates that something has happened. Thus, a command can be rejected, an event cannot. Syntactically, a command is formulated in present tense, while an event is formulated in past tense.

The previous section introduced central aggregates. As they are the domain event producers, the different domain events provided in figure 5.8 correspond to them.

External Model Events External Model events relate to the external model aggregate. Thus, they basically reflect the external delivery model life cycle, i.e. they comprise of a *ModelRegistered Event*, which signals the successful registration of an external model reference at the delivery system, a *ModelDeleted Event* notifying the deletion of a previously registered model reference and a *ModelUpdated Event* signaling that the model reference (see section 7.2) has been updated. In addition a *ModelTriggered Event* exists indicating that the model should be planned and executed.

Delivery Process Events Delivery Process Events correspond to the delivery process aggregate. Two different delivery process events exist. A *ProcessPlanned Event* being triggered when an internal delivery model has been planned and a *ProcessPlanned Event* notifying when the execution of a previously planned internal model has been triggered. For the failure handling (cf. section 6.3.7) there is an *ProcessPlanningFailed Event.*

Execution Event The execution domain differentiates between to different execution event types. Both are produced by the delivery process execution aggregate. *DeliveryProcessExecution Events* notify observers about updates regarding the overall delivery process execution (i.e. started, paused, resumed, terminated and failed events) and *ActivityExecution Events* signal updates of a single command execution as part of the delivery process execution. Beside the obvious *ActivityStarted*, *ActivityCompleted* and *ActivityFailed* event, there is an *ActivityProgressed Event* which is published whenever the activity execution (cf. section 7.5) has progress while still running. This enables the delivery system to trace the activity execution and to provide snapshot mechanisms even on activity execution level.

5.2.3. Event Sourcing

Event Sourcing is the concept of persisting all domain events which have occurred for an aggregate [Ver16]. Thus, in contrast to traditional state handling techniques (e.g. Active Record [Fow02]) where only the aggregate state as a whole would be persisted, this results in a sequence of state-changing events that determine the current aggregate state. This *event stream* allows to have a complete history over every aggregates state change.

Considering our requirements, event sourcing allows to meet Req-8: Traceability. In addition, event sourcing allows to put the system in any prior state by replaying the events to a certain moment in time, which greatly eases debugging. Given the importance of a delivery system for the whole organization the ease of debugging is quite important in the delivery system context and also improves maintainability. Another great advantage of event sourcing is that no information is discarded. Traditional state handling techniques only persist information which are required for current or expected business interest. If unforeseen business interests occur, the persisted state must be extended to accommodate the new required information. In the traditional approach the business interest can only be answered starting from the point of time when the persisted state has been extended. In the event sourcing approach the event projection must also be adapted, but the business interest can then be answered from the beginning of time by replaying all persisted events. Transferring this benefit to our problem domain, event sourcing enables great possibilities for example in improving the self-organizing capabilities (see section 7.3) by applying machine learning to all stored events (see section 10.2).

Overall, we decided to apply event sourcing as a means to meet Req-8: Traceability while simultaneously easing the reproducibility (event replay) to improve maintainability.

5.2.4. CQRS

Command-Query Responsibility Segregation (CQRS) is an architectural pattern that applies the command-query separation (CQS) [MA89] design principle on architecture level [Ver12]. The CQS principle states *"every method should be either a command that performs an action, or a query that returns data to the caller, but not both"*. Transferring this idea to our domain, aggregates normally would have command and query methods. As described in the detailed design (chapter 7) there also would be repositories [Fow02] with different methods to query and filter the data. CQRS segregates the concerns of reading and writing by means of separated models, i.e. the aggregate only has command methods and the corresponding repository only a simple find method that returns the aggregate by id. This is the *command model*. The query concerns are handled by a dedicated *query model*, which may be denormalized and optimized for queries.

The rationale behind this approach is as follows: In complex domains users typically are interested in data that is spread across several aggregates. This requires clients to query multiple repositories and condense their information into a data transfer object [Fow02] which requires them to have detailed domain knowledge. In addition it introduces a strong coupling and requires aggregates to also consider query concerns making them less optimized for their write concerns. CQRS allows aggregates to concentrate on the write concerns, while their domain events are projected into one or multiple optimized query models. Of course, this separation into two models adds complexity, thus it really depends on the use case whether applying CQRS provides benefits.

In our case, users are primarily interested in the delivery process as a whole, i.e. an integrated view of the external delivery model, the planned process and its execution. Thus, there only is one central query model. Applying CQRS then allows to integrate the different domain events to provide a unified view, while simultaneously reducing the complexity of each aggregate. Meeting Req-5: Custom PDLs, i.e. visualizing the delivery process in multiple external models can then be easily archived by providing a specialized projection of the domain events. Overall, we therefore apply CQRS in the design of the delivery system.

5.2.5. Delivery Process Activity

Given the importance of our decision to encapsulate the delivery process activities into typed activities to meet Req-3: Abstraction and Req-6: Model Validation and to improve the overall architecture of a delivery system (C1 - Project Evolution), this section explains the design decision and provides details.

As already motivated, existing delivery systems have few semantic in their activities. Their activities represent arbitrary shell commands which the delivery system delegates to the operating system. While this approach offers maximal flexibility, the activity basically is transparent to the delivery system. Thus, the delivery system hardly can validate the shell commands (Req-6: Model Validation) neither provide meaningful execution feedback nor hide technical details (Req-3: Abstraction). Basically, it heavily couples the delivery system and the delivery model to its environment. This leads to very fragile systems, as a change in the delivery model or the environment can immediately break the delivery process.

We therefore decided to encapsulate the functionality into activities, which encode the required steps to be performed. To provide more semantics, we introduced an activity classification improving the reasoning capabilities of the delivery system. Thus, beside describing it's interface (activity specification) each activity conveys its abstract functionality (transformation or assessment). Overall, as everything required to perform an activity is encapsulated in the activity itself, they realize *information hiding* [LL10], thus providing a stable interface making the delivery system more robust and less sensitive to changes. The interface allows a unified way of executing an activity, which allows for extensibility of new activities.

Service Encapsulation

The previous section motivated our decision to encapsulate functionality into typed, self-describing activities. These activities need some software components to realize their functionality. The functionality typically is provided by third-party tools or service providers. Thus, the architecture needs to allow their easy integration (Req-1: Integration of new technology). In section 5.1.4 we identified the delivery activity context as a boundary for the activity details. Strictly speaking, each tool or service provider has its own bounded context, as the concepts and software components might be totally different. Similar to our microservice decomposition argumentation (cf. section 5.1.5) this suggests to realize each tool, respectively service provider context, by means of a dedicated microservice, which then also allows to support heterogeneous technologies. Each service then offers cohesive activities relating to this tool or service provider. Taking a look at some service principles [Erl05] and mapping them to our scenario, suggests that microservices are a natural fit to integrate the different tool and service providers:

Abstraction Services introduce abstraction, which means that (implementation) details not required for the usage of the provided functionality are hidden. This reduces coupling, therefore improves maintainability and eases the delivery process modeling as fewer knowledge is required.

Standardized contract Services have standardized contract. This allows to consume the offered activities and provide means for Req-4: Self-Organizing.

Discoverability Services are discoverable. To be able to use the offered activities, they need to be discoverable in the first place.

Autonomy Services have a high level of control over their execution environment. This allows to reduce the coupling between the delivery system and its environment, thus reducing fragility.

The service encapsulation then also allows to reuse the offered activities outside the software delivery context.

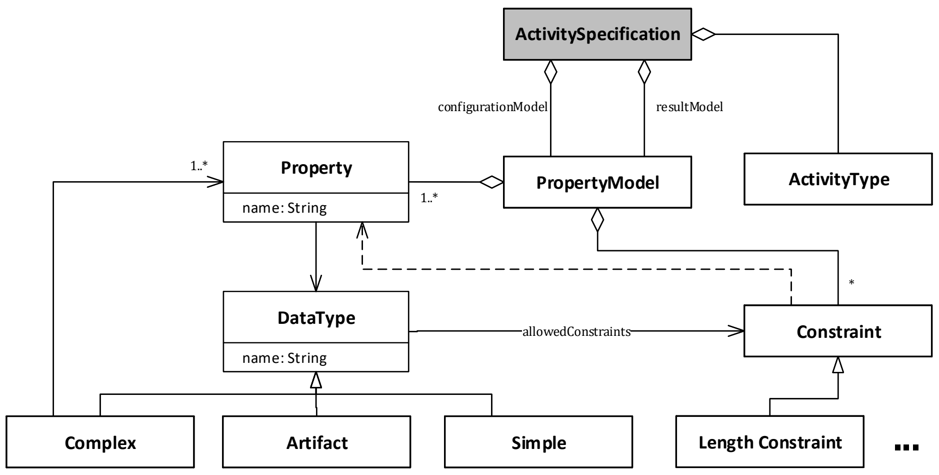


Figure 5.9.: Activity Specification

Activity Specification

Important concept to allow for Req-6: Model Validation and Req-4: Self-Organizing it the activity specification. It describes both the possible configuration values of an activity and its result scheme. Figure 8.5 provides the activity specification design. In principle, it is all about describing properties with their data type. Our activity specification provides three different data types. A simple data type expresses basic data types. The delivery system already provides *String*, *Numeric* and *Boolean*. But of course, custom data types can be defined. Complex data types represent objects consisting of other properties. The special *artifact* data type is due to the fact that conceptually the delivery system focuses on the artifact delivery stream, thus transformations consume and produce different artifact. The propertyModel also contains constraints to not only allow for syntactically validation (property names & data type), but also semantically, for example, if a string has a certain length. These constraints are used for the self-organizing capabilities. Especially for the artifact type, as we differentiate them based their concrete artifact type. Section 7.3 provides more details about the self-organizing capabilities.

5.3. Summary

This chapter used Domain Driven Design to model the Software Delivery Domain as explicitly as possible. We decided to use Domain Driven Design since this thesis tries to provide a framework architecture for Software Delivery Systems. Designing a framework requires to have a deep domain understanding in order to incorporate appropriate hot-spots. As chapter 4 identified, existing delivery systems have fundamental problems to meet our requirements. Therefore, we started from scratch by modeling the domain using Domain Driven Design.

First, Strategic Design tools assisted in identifying coarse-grained concepts and responsibilities. Most importantly, we identified and designed our core domain during the strategic design. Thereby, we introduced an delivery process activity classification. Each activity either is a transformation, an assessment or a quality gate (which in fact is a special transformation). Transformations mutate, translate or merge their input artifacts into a new artifact. Assessments perform measurements to be able to evaluate certain attributes of their input artifact, which they publish as a report. Quality Gates consume artifacts, corresponding reports and policies to decide which artifact is promoted or rejected. Each activity thereby has an activity specification that defines both the input and the output model. This specification allows to validate activities defined in a delivery model.

Based on our core domain, we identified several supporting domains which we restricted by defining bounded context. The next chapter uses these contexts to map them to software-technical units.

In addition to strategic design, we detailed some identified concepts in tactical design and made important design decisions. Most importantly, we decided to use the microservice-architectural style because it provides great flexibility and allows to easily integrate new technology. To decouple these services, we use messaging. Following our Domain Driven Design the messages correspond to domain events of our core aggregates identified during tactical design. Another important decision we made is to use Event Sourcing as a means to meet traceability and CQRS to reduce the complexity of each core service. The next chapter uses these design decisions to present an architecture for our delivery system framework.

6. Architecture Overview

Contents

Central part of this thesis is to introduce an architecture which tackles the identified challenges and meets the requirements. The previous chapter introduced important concepts following domain driven design principles. Based on these concepts, this chapter provides an overview of our architecture. Thereby it is structured according to the 4+1 View Model [Kru95]. We address the following views:

Logical View The logical view provides high level abstractions of the architecture based on functional requirements and logical concepts identified in chapter 5

Development View The development view presents the architecture from a technical perspective. It contains the different layers and their components.

Process View The process view focuses on important dynamic aspects of the architecture, i.e. it describes how the different components of the architecture interact.

We do not discuss the physical view [1] because considering our microservice design (cf. section 5.1.5) this mapping can be arbitrary and thus without providing any value in this thesis context. All microservices could be deployed on the same node, each on it's own node or any combination in between. Also we do not discuss scenarios, as the scope of this thesis is to tackle C1 - Project Evolution and C2 - Modeling Usability. The first challenge basically deals with improving the architecture of existing delivery system, thus the scenarios should be sufficiently known. The second challenge is a concrete scenario, which does not require further explanation in our opinion.

[1]The physical view describes how the different components are mapped to physical nodes in the network

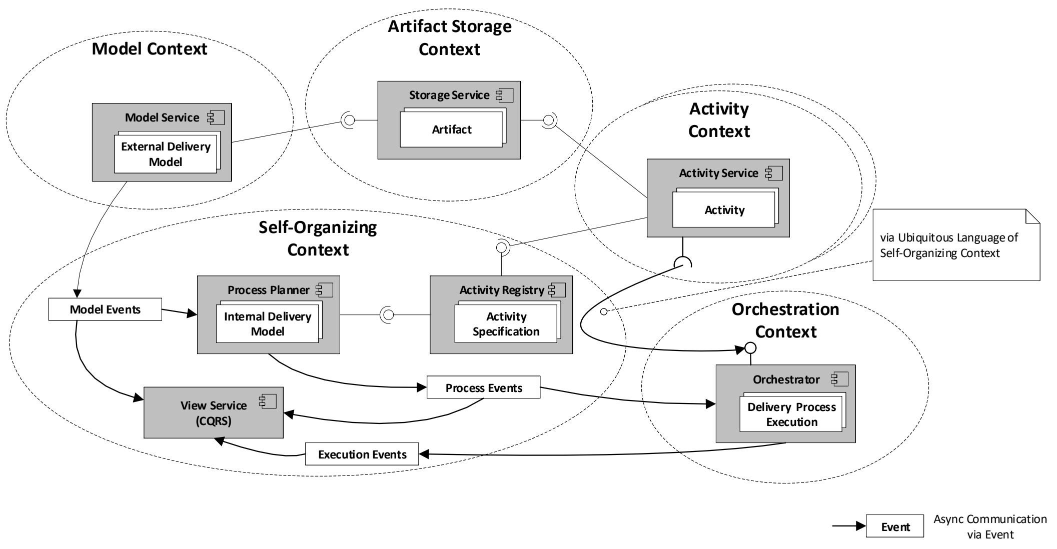


Figure 6.1.: Logical View - Bounded Context Service Mapping

6.1. Logical View

Chapter 5 applied Domain Driven Design and presented strategic and tactical design decisions. Main contribution were several bounded context that decompose the overall domain. This section maps these bounded contexts to software entities. Thereby, we share the view of some others that a microservice might be much smaller than a DDD bounded context [Ver16]. A microservice might only model one concept of the bounded context. Following this idea, multiple microservices will still logically be in the same bounded context.

Figure 6.1 provides the context mapping. As depicted, most bounded contexts map one-to-one to a microservice. The model context is realized through the model service, the storage context through a storage service and the orchestration context through an orchestrator service. As discussed in section 5.2.5 there is one activity context per tool (cf. Req-1: Integration of new technology), thus there are multiple activity contexts with each one comprising a single activity service. The Self-Organizing context is realized by means of a process planner which deals with the internal delivery model, an activity registry which provides the activity specification concept and a view service which projects the events of our core aggregates (cf. 5.2.1). As stated, these services logically correspond to the same bounded context and share its concepts. But considering the focus on maintainability and flexibility, design practices (single responsibility principle) suggest to decompose the context into multiple services. Planning the delivery process and managing available activity specifications are separate concerns.

Figure 6.1 also depicts the service relations. These relations are similar to the bounded context relations (cf. section 5.1.4). We won't go into much details here, as section 6.3

provides the dynamic aspects. But for clarity reasons, we want to highlight the following: Taking a look at the relation between the activity service and the orchestrator service and comparing it to the relations of their bounded contexts, there is a difference: While the contexts have no relation, there is a relation between the activity service and the orchestrator. This is on purpose. Since the context map provides the integration points of bounded contexts, i.e. an overview about how a context uses concepts of another context and figure 6.1 depicts the communication between services. Such a scenario can occur, if the communication is based on the language of another bounded context. In this concrete case, the activity service communicates with the orchestrator via the language of the self-organizing context. This has the following rationale: We designed the orchestrator to be generic. If the activity services would use the language of an off the shelf orchestrator, we cannot change the orchestrator without adapting the activity services. Therefore they both use the core language, which conceptually provides loose coupling between them.

6.2. Development View

Figure 6.2 (page 63) provides an overview of our technical architecture. It highlights several key components. As discussed in section 5.1.5 and indicated above, the architecture uses the Microservice Architectural Style which meets Req-2: Modularity.

Because the development view also contains technical services, we couldn't arrange the already identified services according to their bounded contexts while still keeping the model readable. Instead figure 6.2 (page 63) organizes the microservices in layers [Fow02]. To prevent misunderstandings we explicitly want to highlight that each depicted service is isolated and individually deployable.

The following introduces our different layers from bottom to top. The lowermost layer (*External Provider*) is not part of the architecture itself. To provide an holistic overview, figure 6.2 (page 63) nevertheless depicts this layer.

Activity Layer The activity layer houses *Activity Services* that realize coherent activities in a self-contained manner (cf. section 5.2.5). Typically, each activity microservice encapsulates the functionality of a tool (Req-1: Integration of new technology). But it might also be reasonable to split the functionality into multiple activity services. Each activity service registers its offered activities during startup by means of publishing his activity specifications (cf. section 5.2.5) at the activity specification registry (delivery process management layer). Activity services might interact with external tools or services to realize their activities. As these providers are external, we do not discuss them in further details. Following our core domain (cf. section 5.1.1) each activity either is a transformation or an assessment. Typically, an activity service provides only activities of a certain type. If an activity service only provides transformations it is a *transformation service*. Analogously, it's an *assessment service* if he only provides assessment activities. The service can also provide activities of both types. We then consider him a *hybrid service*.

Overall, the activity layer provides means to tackle Req-1: Integration of new technology because integrating new technology is a matter of providing a new activity service (cf. section 5.2.5). An activity service itself tackles Req-3: Abstraction (cf. section 7.5.2. They also enable Req-6: Model Validation by means of providing activity specifications. Section 7.5 details the architecture of activity services.

Delivery Process Management Layer The delivery process management layer contains services required for managing and executing delivery models. From a domain driven design perspective this includes the model context, the orchestration context and the self-organizing context (cf. section 5.1.4). Thus, we consider the management layer as our core layer. It houses the *model service* which imports, validates and translates external delivery models into the internal delivery model (cf. section 5.1.1), the process planner which plans and optimizes a given internal delivery model, and the *orchestrator* which is a process manager (cf. [HW03]) that controls the execution of the aforementioned activity services. The activity specification registry (cf. registry pattern [Fow02]) from the Self-Organizing context is also part of this layer. It keeps track of all available activities and their activity specifications provided by the activity services. As only the model service, the process planner and the orchestrator deal with our central aggregates (cf. section 5.2.1), we abbreviate these three services (and not the registry) as *core services* in the remainder of this thesis. Summing up, the separation of concerns between the import, the planning and the execution control increases the robustness and allows to easily extend and modify each aspect and to use off-the-shelf solutions for example for the orchestrator. To employ isolation, loose coupling and location transparency between these services, they communicate asynchronously via domain events as motivated in section 5.2.2. Overall, the delivery process management layer is responsible for Req-6: Model Validation, Req-4: Self-Organizing and Req-5: Custom PDLs. Section 7.1 provides details about the activity specification registry, section 7.2 details the model service, section 7.3 details the process planner and section 7.4 details the orchestrator.

Infrastructure Layer This layer provides microservice that offer general purpose functionality (for transport and persistence) required by other microservices in the architecture. Beside the artifact storage service contained in this layer which was functionally motivated (cf. section 5.1.4), all other services in this layer are technically motivated. They comprise of an event store, which allows the core services to use event-sourcing (cf. section 5.2.3) and event notification (cf. section 5.2.2), a monitoring service to assist the operation and a discovery service to allow peer to peer communication without knowing the location of every service (e.g. for the API gateway).

Access Layer Motivated by microservice best practices [New15], the access layer contains an *API Gateway* to provide clients in the visualization layer a single entrypoint into the system and to deal with crosscutting concerns like caching and security. In addition, this layer contains the view service (cf. 6.1). As motivated in section 5.2.4, our core services apply CQRS. The view service maps and integrates domain data

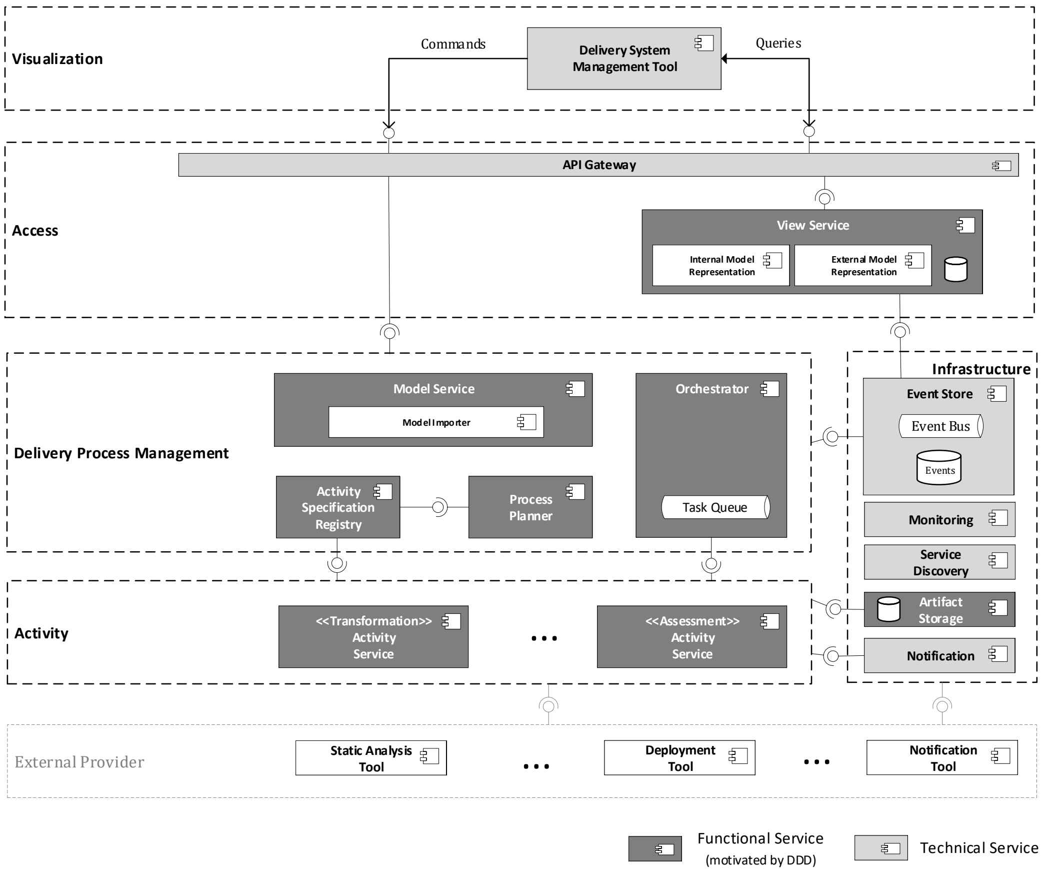


Figure 6.2.: Layers and services in the delivery system architecture

across our central aggregates section 5.2.1. Thus, it subscribes to external model, delivery process and execution domain events (cf. section 5.2.2) to build a query model integrating all relevant aspects across the delivery domain. To support Req-5: Custom PDLs different model representation services or components can be provided that provide a projection of domain events into their dedicated model.

Visualization Layer Similar to a 3-tier design [Gor06] our top layer is the visualization layer. It houses frontend services. As no special requirements regarding the visualization exist, only one frontend, the *Delivery System Management Tool* is necessary right now. Following the CQRS approach of our core service, the management tool offers a task-based user interface which allows to import new delivery models, visualizes existing ones and allows to trigger their execution. The future work discusses another possible visualization tool, namely a graphical delivery model modeler (see section 10.2).

Summing up, our microservice-based architecture for delivery systems provides several

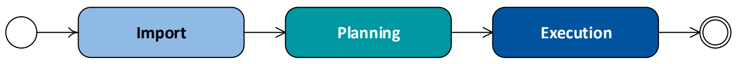


Figure 6.3.: Architecture Control Flow

hot spots. New technology can be integrated by means of adding an activity service, additional pipeline description languages can be added by providing a corresponding model service, new Self-Organizing capabilities can be realized by means of additional process planner services, different off-the-shelf orchestrator can be added and even new delivery process representations can be provided.

6.3. Process View

This section provides insights into the dynamic aspects of the delivery system architecture. First, section 6.3.1 defines the overall control flow, which affects all our core aggregates (cf. section 5.2.1). Section 6.3.2 then introduces how this flow is integrated into the delivery system architecture. Section 6.3.4, Section 6.3.5 and Section 6.3.6 respectively then provide an overview of each individual phase of the control flow. The control flow thereby describes the execution of a delivery process starting with the import of a delivery model. Prerequisite for execution a delivery model is to register it at the delivery system first. Section 6.3.3 details how external delivery models are registered at the delivery system.

6.3.1. Architecture Control Flow

As section 6.2 stated, the delivery system architecture provides several hotspots to adapt functionality because of its microservice structure. To control the flexibility at individual service level, the architecture defines a higher-level control flow, which manifests from the Domain Driven Design (cf. chapter 5). We call this flow *Architecture Control Flow*. Figure 6.3 depicts its three phases. It starts with controlling the import of an external delivery model, i.e. the download from version control or wherever it is stored. The import process is realized by the model service (model context), which downloads and transforms the external delivery model to the internal delivery model. After importing, the internal delivery model gets planned and optimized by means of process planners (self-organizing context). Then the orchestrator starts executing the delivery process defined by the optimized internal delivery model.

Overall, the Architecture Control Flow is at the heart of the delivery system architecture. It extends the architecture towards a delivery system framework by realizing *Inversion of Control* [Fow05].

Following the event driven approach, the architecture realizes the Architecture Control Flow by means of a *Choreography* [New15], i.e. each core service knows how to react on certain events. This has one central implication: Each core service can influence the

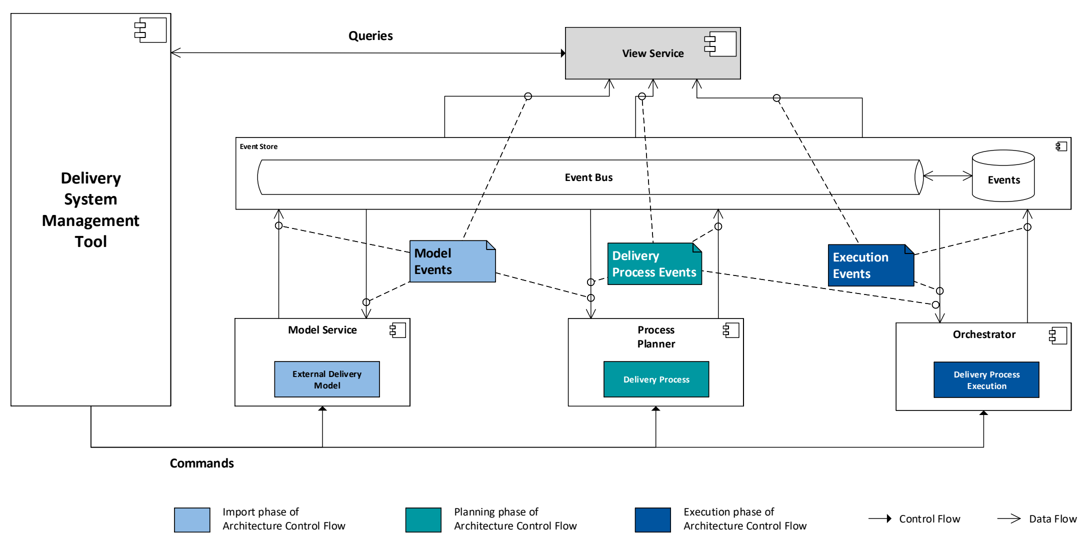


Figure 6.4.: Simplified Message Flow in the Delivery System

control flow by not not publishing the corresponding event. Similar to a *Detour* [HW03], a core service could integrate an arbitrary sequence of steps in between or could abort the flow completely.

To enforce the control flow, one could use a designated central component that monitors progress and instructs services to execute certain commands. We decided against such an orchestrated approach to allow for tailoring by means of integrating custom services. To be able to enforce the control flow, the core services need to be fixed. Other planners or model importer then can be added to each individual core service by means of an adapter [Gam+02]. Each core services then orchestrates its adapters or internal components.

6.3.2. Message Flow Overview

Figure 6.4 provides an overview how the architecture control flow is integrated into the delivery system architecture. Recapitulating chapter 5 we use messaging for publishing domain events of our core aggregates, i.e. of the external model aggregate, the delivery process aggregate and the delivery process execution aggregate. Section 5.2.2 provided an overview of their domain events.

Considering how our aggregates are distributed across the core services, the model service publishes and consumes model events, the process planner consumes both model events and delivery process events and publishes delivery process events and the orchestrator consumes both delivery process events and delivery process execution events and publish delivery process execution events.

The architecture control flow then is realized as follows [2]: The model service publishes

[2]see section 5.2.2 for the individual event types

an ModelTriggered event, the planner service consumes this events, plans the delivery model and publishes a ProcessTriggered event. The orchestrator consumes this event and starts executing the modeled delivery process, thereby publishing delivery process execution events.

Figure 6.4 also illustrates the CQRS pattern applied across our core services (cf. section 5.2.4): The delivery system management tool triggers state changes via CQRS commands. The core services perform the changes and publish domain events in response. Instead of querying the core services, the management tool requests these information from the view service, which uses the published events to build an integrated, read-only query model. The event store depicted in figure 6.4 provides the required infrastructure for this approach, i.e. an event bus (cf. message bus [HW03]) to communicate events via messaging and a persistence mechanism to store these events. The core services use the stored events to determine their aggregate state (cf. event sourcing section 5.2.3).

6.3.3. Model Registration

Clearly, most important scenario for a delivery system is the execution of a delivery process as this provides value to the user. But such a delivery process must be defined in the first place. Meeting Req-7: Best practices requires to keep everything in version control which also includes the delivery model. Recalling chapter 5 we call this model *external delivery model*. As this model is not controlled by the delivery system, future changes would be ignored, if the model is imported to the delivery as a whole. Thus, the delivery system only stores a reference onto this external model. This reference then allows to trigger the delivery process execution while reflecting the latest model content. We call this process *model registration*. This section provides an overview of the model registration dynamics. Section 6.3.4 then provides an abstract view on the import process in context of the fundamental architecture control flow.

Three services are involved in the import process. Starting with the user requesting the import of an existing external model, the management tool issues an importDeliveryModel command to the model services. This command comprises all information required to fetch the external model (e.g. file path, url, etc.). The model services validates the command. If all required command parameters are set and in their bounds, the model service tries to reach to specified model location. If the referenced model can be reached, the model service publishes a modelRegistered event which contains meta information about the model (location, hash sum, etc.). The view service listens to this event and creates a corresponding representation for the management tool (cf. CQRS section 5.2.4).

6.3.4. Import

The first phase initiated by the architecture control flow is to download the latest model version and to convert it into the internal delivery model. We call this process *import*. Prerequisite for importing a model is that the delivery system knows about it.

Figure 6.6 depicts the high-level steps of the import process as an UML activity diagram. These steps are performed by the model service. Section 7.2 provides more

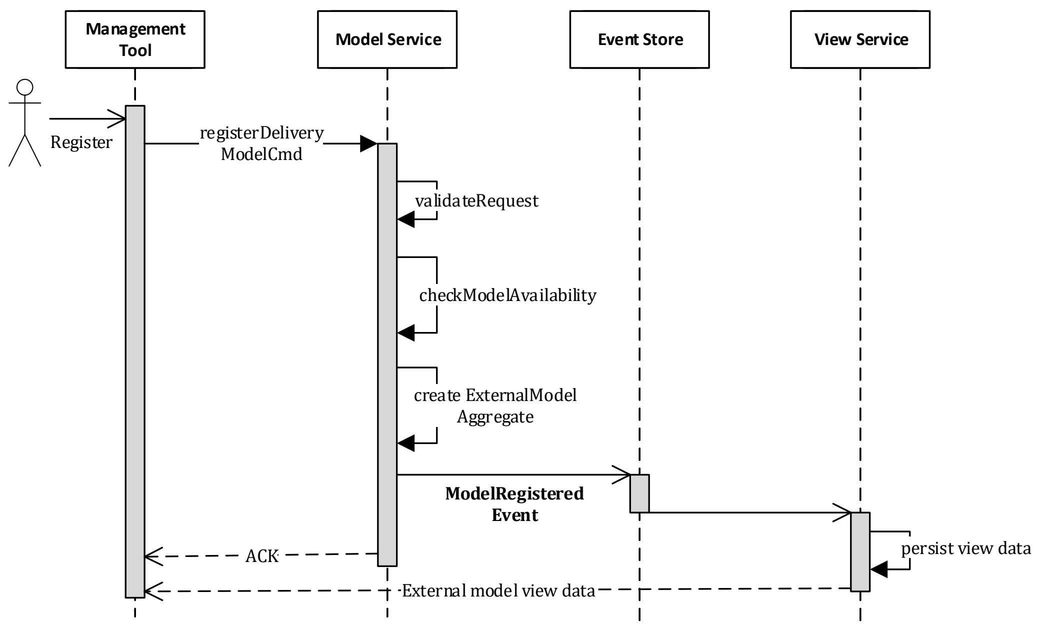


Figure 6.5.: External Model Registration Process

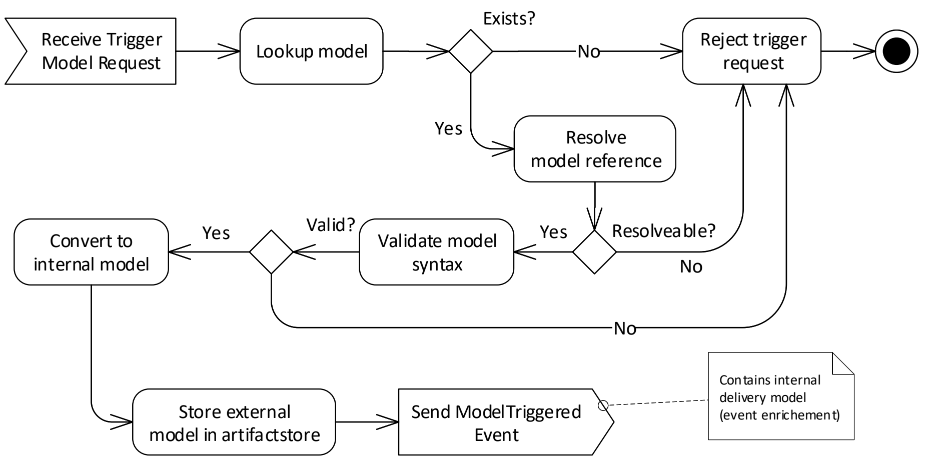

Receive Trigger Model Request
Lookup model
Exists?
No
Reject trigger request
Yes
Resolve model reference
Resolveable?
Yes
No
Validate model syntax
Valid?
Yes
No
Convert to internal model
Store external model in artifactstore
Send ModelTriggered Event
Contains internal delivery model (event enrichement)

Figure 6.6.: Abstract Import Process

details on the model service. The import process starts on receiving a delivery model trigger command from the delivery system management tool. The trigger command contains the id of the corresponding external model aggregate (cf. section 5.2.1). If the referenced aggregate exists, the model aggregate downloads the referenced delivery model and validates it syntactically (the process planners provide more sophisticated validation). Afterwards, the model aggregate converts the external delivery model into an internal delivery model, to allow process planners to plan independently from the concrete model. For meeting Req-8: Traceability, it then uploads the external delivery model to the artifact store, and publishes an modelTriggered event. This event contains the internal delivery model (we decided to use event enrichment (cf. section 6.3.7)) and the artifact id of the uploaded external delivery model.

6.3.5. Planning

The second phase initiated by the architecture control flow is planning, i.e. the completion and optimization of an internal delivery model. The first phase imported the external delivery model and translated it into the internal delivery model. On receiving this internal delivery model, the planning process starts. Figure 6.7 depicts the high-level steps of the planning process as an UML activity diagram. These steps are performed by the process planner service. Section 7.3 provides more details on the process planner service. Following defensive programming practices, the planner service first validates the received delivery model. Since the model typically is incomplete at this early stage in the process, the validation only is of technical nature (e.g. syntax). Key requirement for planning is a knowledge base. Thus after basic validation, the planner service fetches the available activity specifications. Next, matching planners are selected. Following section 7.3.2, the process planner service internally uses multiple planners. Thereby we

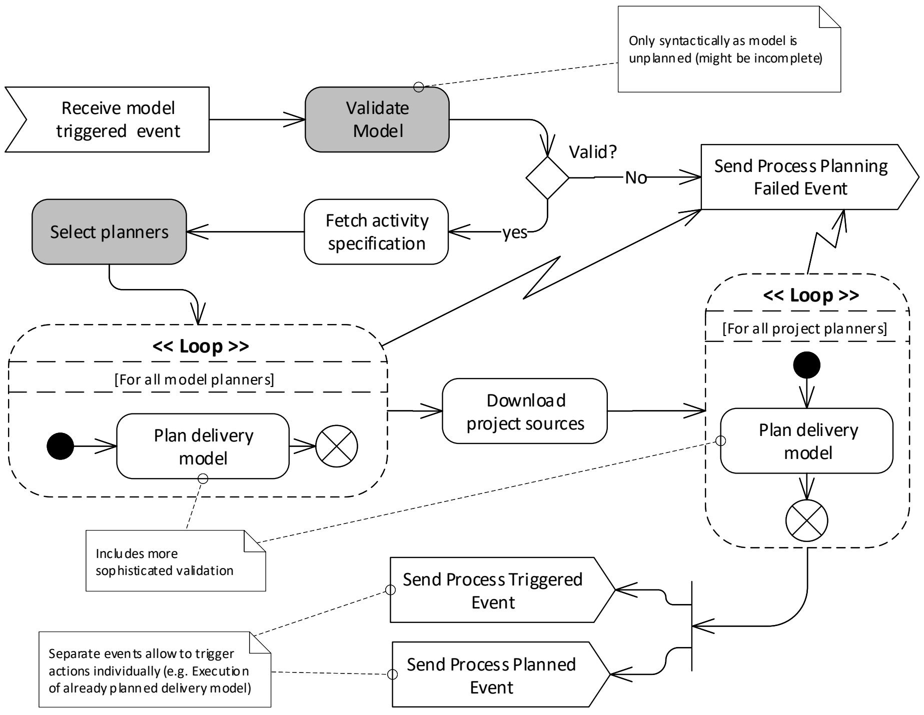

Only syntactically as model is unplanned (might be incomplete)
Receive model triggered event
Validate Model
Valid?
No
yes
Send Process Planning Failed Event
Select planners
Fetch activity specification
<< Loop >>
[For all model planners]
Plan delivery model
Download project sources
<< Loop >>
[For all project planners]
Plan delivery model
Includes more sophisticated validation
Send Process Triggered Event
Send Process Planned Event
Separate events allow to trigger actions individually (e.g. Execution of already planned delivery model)

Figure 6.7.: Abstract Planning Process

differentiate between model planner and project planner. Depending on the delivery model different planner are applicable. During the planning process matching planners are therefore dynamically selected. After matching planners have been selected, the actual planning starts. Thereby, the planning is divided into two phases, which results from project planners being more expensive then model planners. Project planners require the project sources for planning. After the model planner have finished their planning, the planner service therefore downloads all project sources (which are referenced by the delivery model). Then, all selected project planner start their planning. After the planning has finished, both an process triggered and an process planned event are published to decouple both operations from each other and allow to trigger each one individually later on. Section 7.3 provides further details on the planning.

6.3.6. Execution

Last phase initiated by the architecture control flow is the execution of the previously planned delivery model. Figure 6.8 provides an overview of the execution process as an UML sequence diagram. Recalling section 5.1.4 the delivery model is expressed by means of the core language, i.e. in the internal delivery model. The orchestrator translates the internal delivery model into his orchestrator model. This indirection is required as

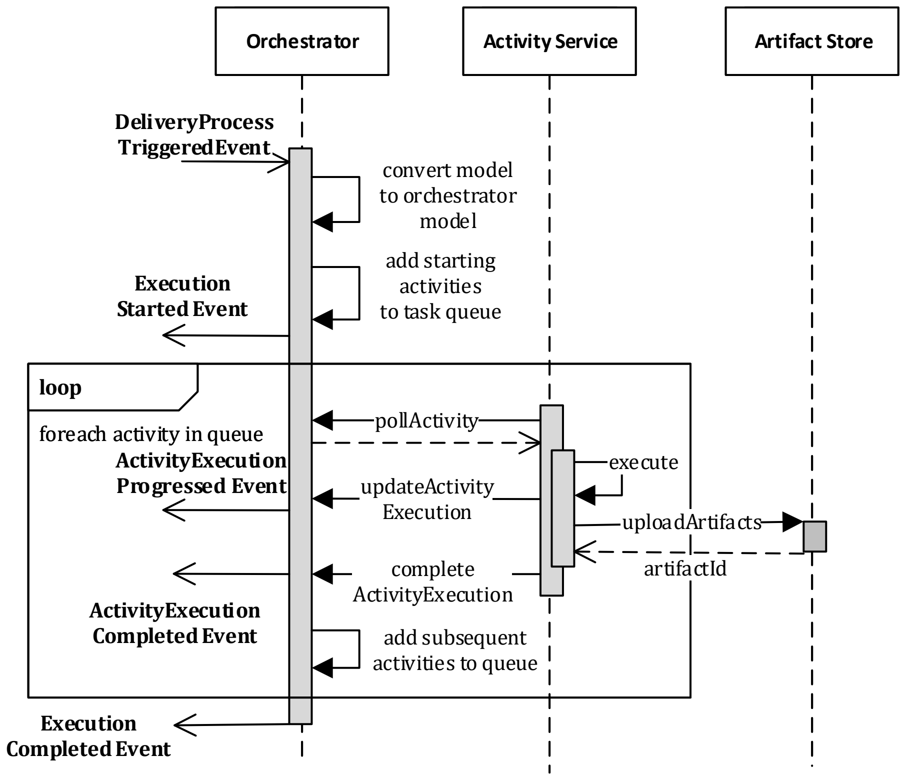

Orchestrator
Activity Service
Artifact Store
DeliveryProcess TriggeredEvent
convert model to orchestrator model
add starting activities to task queue
Execution Started Event
loop
foreach activity in queue
pollActivity
execute
ActivityExecution Progressed Event
updateActivity Execution
uploadArtifacts
artifactId
complete ActivityExecution
ActivityExecution Completed Event
add subsequent activities to queue
Execution Completed Event

Figure 6.8.: Delivery Process Execution Process

we want to support different orchestrator realizations (we classified the orchestration subdomain to be a generic one). A different orchestrator can then be used by simply providing a corresponding adapter. Section 7.4 provides more details on this. Having translated the internal model to the orchestration model, the orchestrator starts the execution. Following the execution order the orchestrator adds activities to its queue. The activities services consume these activities and execute them. Section 7.5 details why we designed the activity services as polling consumers. During execution the activity services can update their progress at the orchestrator at any time. This allows to provide users with feedback during long running activity executions. Any update to the orchestrator leads to the corresponding event (ActivityStarted, ActivityProgressed, etc.). If all delivery process activities have completed, the orchestrator publishes an ExecutionCompleted event.

6.3.7. General Design Decisions

The above description of the architecture control flow contained some design decisions, which we want to briefly discuss.

Query-back vs. Event Enrichment

When dealing with domain events, there are two possibilities [Ver16], regarding the amount of information included in a domain event: Either the domain events are kept thin which requires the consumer to query back for more data or the domain event is enriched with enough data to satisfy the needs of its consumers. From the autonomy perspective, enrichment is favorable as the dependent receivers can only rely on the event itself. On the other hand it is difficult to predict which information might be needed. Providing too much information then also imposes a security risks as the access cannot be as effectively controlled as in the querying back approach. Overall, both approaches have their pros and cons. Vernon therefore advocates to individually decide per use case [Ver16].

In our concrete use case one driving requirement is autonomy, as we focus on maintainability and flexibility. Moreover, the amount of information required is clearly defined, namely the internal delivery model. Thus, we decided to use the event enrichment approach for the aforementioned events.

Failure Handling

So far, we primarily considered successful execution branches in the process descriptions. Of course, failure will happen (especially in a distributed system) and one should explicitly design for it ([Jin11]). As providing alternative paths in the sequence diagram (figure 6.8) would have lead to a complexity explosion, we decided to present our general approach to failure handling in this section.

In principle, there are two ways of triggering an action. Either by issuing a command directly to a core service or by publishing an domain event which in response causes a core service to react by performing an action (cf. choreography [New15]). In the first scenario the command request can simply be rejected if something fails (e.g. by throwing a meaningful exception). In the second scenario we are already in the context of a running business process. Anything that happens therefore is of interest for the domain. Thus, our core services publish a corresponding failure event in that case. Considering the delivery process execution process, a failure during the running business process can happen through an error in the process planning or through an error in the delivery process execution (orchestrator). Our domain events (cf. section 5.2.2) provide a ProcessPlanningFailed and a ProcessExecutionStartFailed event for these cases.

6.4. Summary

This chapter provided an overview of our architecture by means of the Logical View, the Development View and the Process View from the 4+1 View Model. In the Logical View we mapped bounded context identified in chapter 5 to microservices. Overall, we introduced a model service that tackles Req-5: Custom PDLs, by importing different external delivery model types and translating them to the internal delivery model. An artifact storage service, which in combination with our event-sourced approach allows for Req-8: Traceability. An orchestrator to manage several activity services, which allow toReq-1: Integration of new technology by encapsulating delivery process activities, which also allows for a unified execution interface (Req-3: Abstraction), thereby hiding technical details and providing activity specifications in order to meet Req-6: Model Validation. We also introduced a process planner that completes and optimizes internal delivery models to allow for Req-4: Self-Organizing and Req-7: Best practices. In the process view we then provided an overview of the architecture dynamics. The dynamics highlighted the architecture framework characteristic. At its core, the architecture controls a process consisting of a resolving, a planning and an execution phase. Its microservice architecture thereby provides many hotspots to change each phase realization. But the overall process is given by the architectures inversion of control. In the next chapter, central components and services are detailed.

7. Core Components & Services

Contents

The previous chapter provided an overview of the architecture. This chapter details central services. In principle, these services correspond to the identified bounded contexts (cf. section 6.1). We do not detail the artifact store, as its subdomain is a generic one, thus detailing the realization wouldn't provide much value regarding the thesis scope.

7.1. Activity Specification Registry

The activity specification registry provides the delivery system with the means to meet Req-6: Model Validation and Req-4: Self-Organizing. It thereby realizes the **ActivitySpecification** concept from the core domain section 5.1.1. Like a service registry (cf. [Erl05]), it stores descriptions for discovery of the available capabilities. But instead of services descriptions it stores the specifications of all delivery process activities available

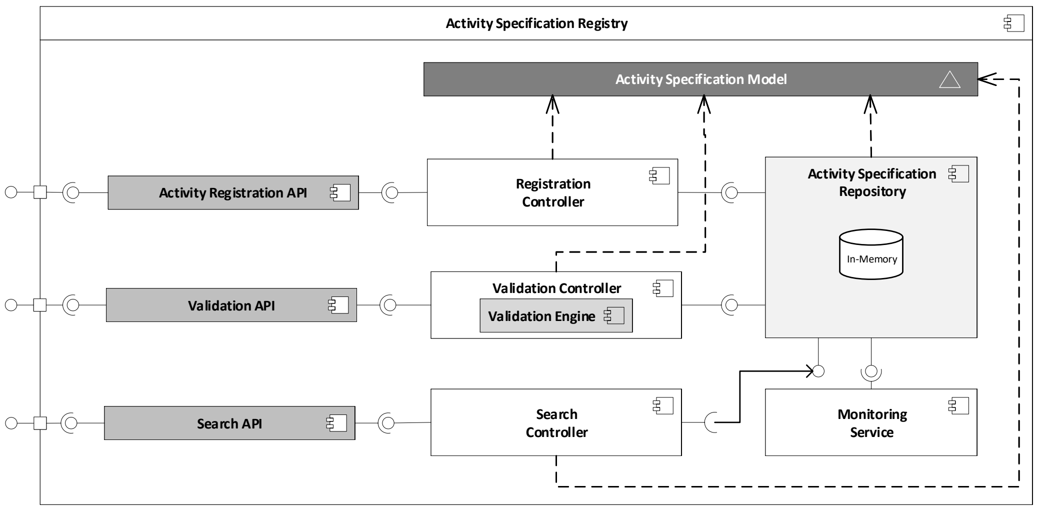


Figure 7.1.: Activity Specification Registry - Component View

in the delivery system. Storing the specifications at a central location has the great advantage of decoupling specification consumers and specification providers. Recalling section 6.2 the specifications are provided by activity services which realize the corresponding activity. The process planner consumes these specifications to reason about a given delivery model. In future, more consumers are possible. Because of the activity specification registry, these consumers are autonomous from activity services, which improves flexibility and maintainability.

The idea of a registry is of course not new. Having references service registry from the SOA world above, the following briefly justifies why we designed our own registry. The solution for service discovery in SOA is Universal Discovery, Description and Integration (UDDI) [Erl05] which contains a xml-based registry. The registry stores information about the discovered services using their WSDL description. As Vianden [Via16] pointed out, UDDI was quite successful in the early 2000 year but several large companies (Microsoft, SAP and IBM) discontinued their UDDI support in 2006. Following Vianden's argumentation, UDDI failed because it tried to solve the discovery and integration problem on a too general level. In our scenario we do not need to discover and integrate arbitrary services. We are only interested in discovering activity services, thus we know their APIs and can ensure their technical compatibility. Consequently, we decided to design a solution specifically tailored to our needs.

The following provide an overview of the activity specification registry components.

7.1.1. Component View

Figure 7.1 depicts the internal components of the activity specification registry. Centerpiece is an implementation of the activity specification model (see section 5.2.5).

All other components provide functionality regarding this model. Overall, the activity specification registry is designed as a microservice, thus providing only limited, tailored functionality. This functionality first an foremost comprises an *Activity Specification Repository* which handles the persistence concerns. To keep the stored information up to date, the *Monitoring Service* continuously verifies if the corresponding activity service is alive and yet provides activities matching the stored specification. Activity services can register new activity specifications via the *Activity Registration API*. The *Search API* allows consumers (e.g. the process planner) to find available activity specifications matching certain criteria (e.g. service name).

The activity specification registry also provides validation support. The *Validation API* allows to check a given object against the activity configuration model respectively the activity result model. We decided to integrate the validation capability in the activity specification registry as this functionality is closely related to the activity specification itself. If an off-the-shell solution for the registry should be used, this aspect could be easily separated in another microservice. Right now, the separation wouldn't provide much value. Overall, the validation support not only is used by the process planner but also allows to realize for example a graphical delivery process modeler (see section 10.2).

7.2. Model Service

The model service is the entry point for external delivery models into the delivery system. It realizes the **ExternalDelivery Model** concept from the core domain (section 5.1.1). Req-7: Best practices requires among others to keep everything in version control including the delivery model. The external model aggregate therefore contains a model reference. Using this reference the model service resolves the model on demand, i.e. when the delivery process execution is triggered. To meet Req-5: Custom PDLs, the model service translate the external model into the internal delivery model. Supporting a new external model type can then be realized locally by extending the model service and without affecting the remaining delivery system.

The following sections detail the model service both from a static (section 7.2.1) and a dynamic perspective (section 7.2.3).

7.2.1. Component View

Figure 7.2 depicts the components of the model service. At it's heart is the *ExternalDeliveryModel aggregate* from the core domain (section 5.1.1). As a service dealing with a central aggregate, it applies the CQRS pattern and uses Event Sourcing for persistence (cf. section 5.2). CQRS Commands enter the service through the *Delivery Model Command API*. Section 7.2.2 details this API. In principle, two CQRS commands are supported: An import command to add a new model reference and a command to trigger the execution of the delivery process described by the external model. Each command is handled by a dedicated *Command Handler* that validates the command request and orchestrates the command execution. Following our domain driven design approach, the command handler

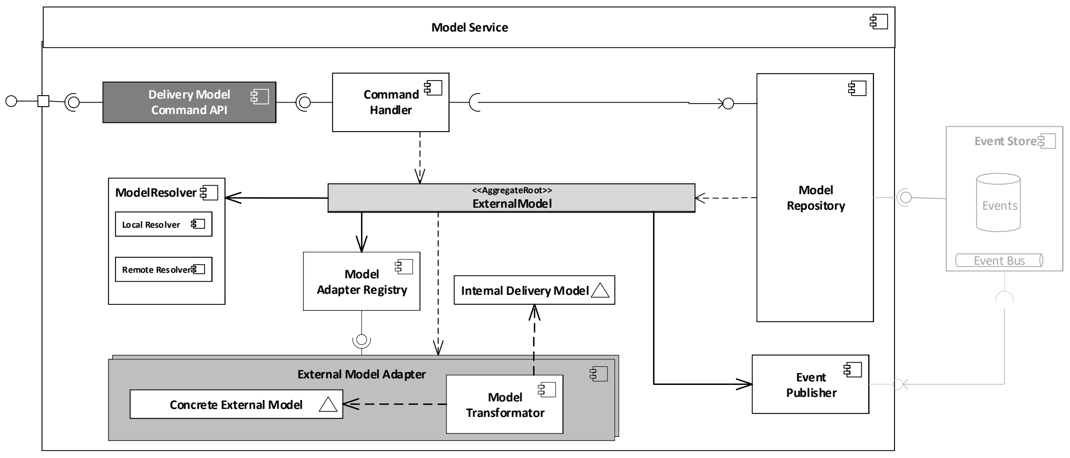


Figure 7.2.: Model Service Component View

loads or creates the corresponding *ExternalModel* aggregate via the *ModelRepository* and performs the requested operations on it which produces domain events. The command handler uses the *ModelRepository* to load the aggregate. The *Event Publisher* allows to publish domain events in the event store (cf. section 5.2.3). Section 7.2.3 details the dynamics based on the model import scenario.

As mentioned above, central entity of the external model aggregate is the model reference. The model service uses the reference to resolve the external model on demand. This behavior is encapsulated in the *ModelResolver* component which provides different resolving strategies (cf. strategy pattern [Gam+02]) depending on the model reference type. The referenced model can have different types (Req-5: Custom PDLs). The model services uses corresponding model adapters managed in the *model adapter registry*. This follows the *Open-Close principle* [LL10] and eases flexibility and maintainability. Each adapter encapsulates the specifics of a concrete external model and has the means to translate it into the internal delivery model for usage independent of the external model.

7.2.2. Delivery Model Command API

The model service provides both the means to import a delivery model into the delivery system and to trigger the execution of such a model. Section 6.3 outlined this process. We therefore consider model service the entrypoint into the delivery system. As such, it API should be detailed. The following specifies the command API of the model service using a well known interface description language, Corba IDL ([Obj17]). Each method is described after the listing.

```
module DeliveryModelAccess {
    interface DeliveryModelCommandAPI {
        string importDeliveryModel(in ModelImportRequest
            importRequest);
        void triggerDeliveryModelExecution(in string modelId);
    };
    interface ModelRegistrationRequest {
        readonly attribute string modelName;
    };
    interface LocalModelRegistrationRequest:
       ModelRegistrationRequest {
        readonly attribute string path;
    }
    interface RemoteModelRegistrationRequest:
       ModelRegistrationRequest {
        readonly attribute string serviceName;
        readonly attribute string commandName;
        readonly attribute Map parameters;
    }
};
```

Source Code 7.1: DeliveryModel Command API specification

registerDeliveryModel triggers the creations of a new delivery model aggregate based on the registration request and returns the aggregate id to enable the caller to trigger further commands (e.g. the model execution) on the aggregate. The command handler either instantiates a local or a remote model reference depending on the import request and persists a ModelRegistered event (cf. section 5.2.2).

triggerDeliveryModelExecution invokes the execution of a delivery model. The command handler loads the aggregate, resolves the model reference (i.e. downloads and stores the references external model in the artifact store) and publishes an ModelTriggered event (cf. section 5.2.2) which then triggers the delivery process execution.

7.2.3. Model Registration

Figure 7.3 provides an overview of the model registration dynamics. For clarity reasons some details are not modeled (e.g. the command validation performed by the registrationCommandHandler).

The model registration is triggered with a corresponding registration request at the *API* (see section 7.2.2). The API delegates the call to a dedicated application service, the *RegistrationCommand Handler*, which orchestrates the command execution. Since a registered model logically represents an aggregate, the registration command handler instantiates a new external model aggregate, which handles all domain logic regarding the external model (registration) and ensure its consistency. For the import of a new

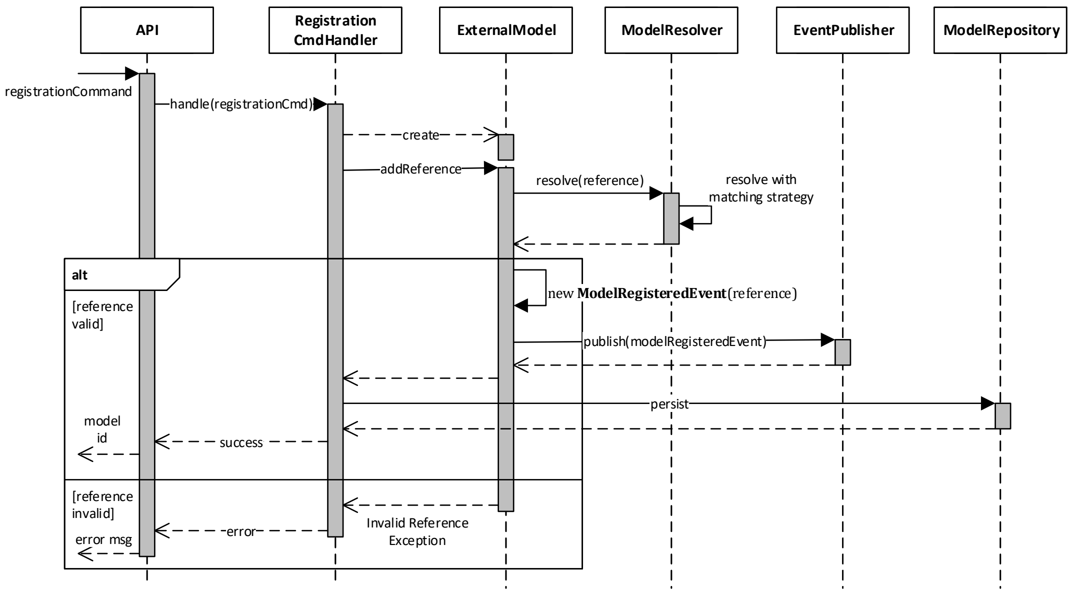


Figure 7.3.: Model Service - Model Registration

model the domain logic is straightforward. As long as a physical file exists at the location referenced by the registration request, the model reference is added to the delivery system. The model service does not perform a model validation upfront or does not verify if a corresponding model adapter exists. The rationale behind this decision is that the referenced model is subject to change. Thus, all validation related operations are performed when the model is fixed, i.e. when the execution of a model is triggered.

Concretely, the external model aggregate uses the *ModelResolver* domain service to verify if the reference model exists. If the model exists, the aggregate publishes an *ModelRegisteredEvent* containing the reference. If the reference is invalid, i.e. if the specified location cannot be accessed or no physical file exists at the specified location, the external model aggregate throws an exception. If the aggregates domain operation executes successfully, the registration command handler persists the aggregate using the *ModelRepository* and returns the outcome to the API. In case of an invalid model reference, the registration command handler returns the corresponding error to the API.

7.2.4. Model Execution

Figure 7.4 details the model execution operation of the model service. For clarity reasons some details are not modeled. Recapitulating section 6.3.6 the model execution resolves an external model references by an external model aggregate and thereby triggers the execution of the delivery process described by this model.

The model execution is triggered with a request at the *API* (see section 7.2.2). The request needs to contain the id of an existing model aggregate. The API then delegates

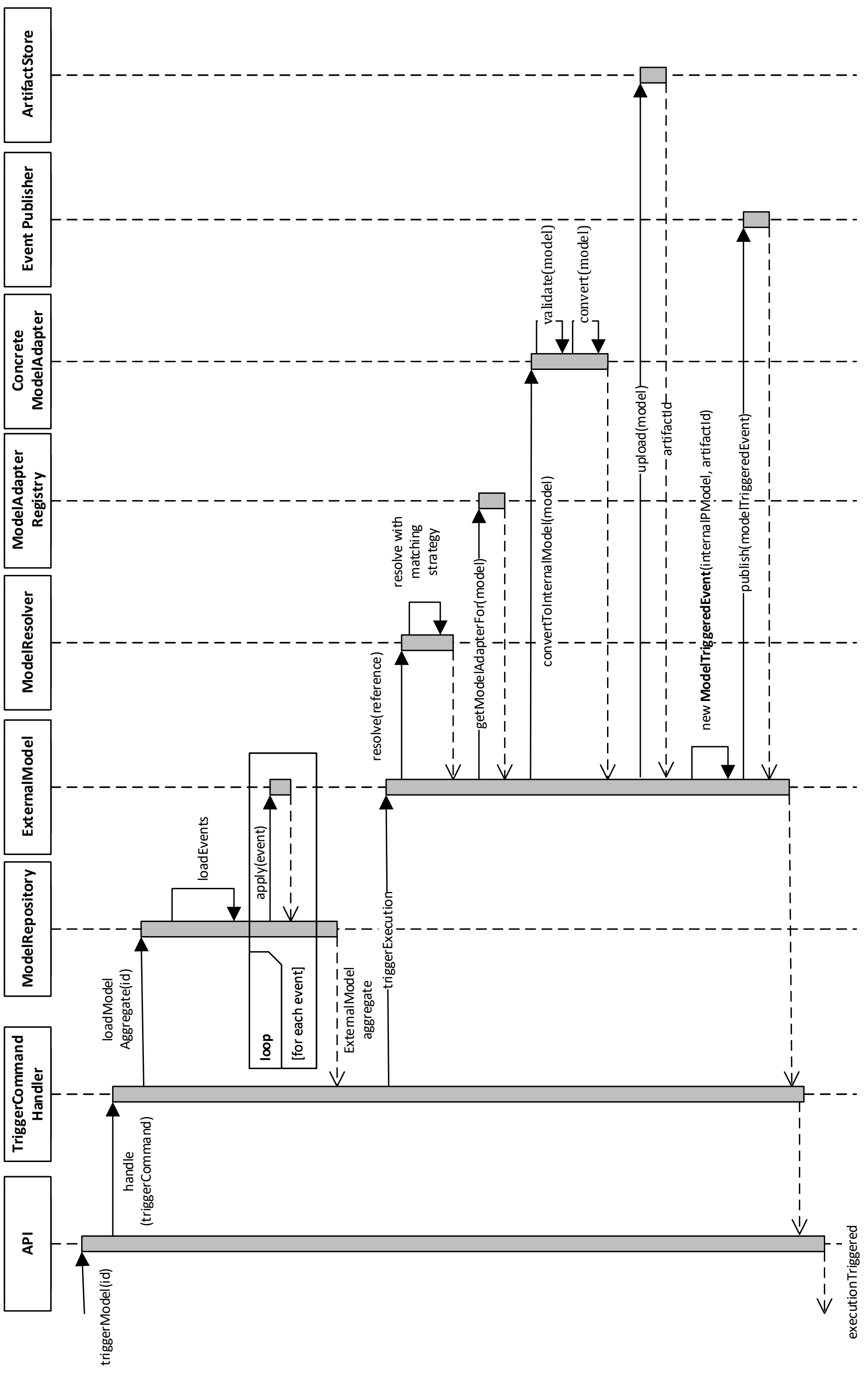


Figure 7.4.: Model Service - Model Execution

the call to the *TriggerCommand Handler* application service that orchestrates the domain-independent execution logic. Following our DDD approach, the domain logic is handled by the *ExternalModel* aggregate. To handle the request, the command handler first instructs the *ModelRepository* to load the aggregate referenced by the request. Since were are using event sourcing (cf. section 5.2.3) the aggregate state needs to be constructed from the event stream. Thus, the model repository loads the events and applies them in the order of their occurrence to the aggregate.

The trigger command handler then passes control to the external model aggregate root to handle the business logic of the model execution. Similar to the model registration (cf. section 7.2.3 the model aggregate resolves it's model reference using the *ModelResolver*. It then uses the *ModelAdapter registry* to determine the correct model adapter for the resolve (binary) model. The adapter provides both the syntactically validation and the conversion to the internal delivery model. The adapter does not perform any semantic validation, e.g. if the modeled delivery process activities are compatible. This decision is made by the process planner (see section 7.3) since the semantic validation contains business rules outside the model service context. For example, the modeled delivery process might only consist of placeholders that are substituted by the planner, such that an semantic validation would be not possible. After the adapter has syntactically validated the external model and successfully converted it into the internal delivery model, the external model aggregate stores the binary external model artifact in the artifact store to provide means for Req-8: Traceability. It then publishes an *ModelTriggeredEvent* contained the internal delivery model and the artifact id of the upload model by using the *Event Publisher*. Finally, control is passed to the trigger command handler which returns success or failure to the API depending if the aggregates execution was successful or threw an exception.

Overall, Figure 7.4 only modeled a successful execution path. For insight into the failure handling we refer to section 6.3.7.

7.3. Process Planner

The process planner is the key component that enables the delivery system to tackle C2 - Modeling Usability. The process planner thereby realizes the **DeliveryProcess** aggregate from our core domain (cf. section 5.1.1). The general idea is to separate the concerns of delivery process modeling and delivery process execution, i.e. to use different models (external and internal model, cf. chapter 5) for them. This not only allows to support Req-5: Custom PDLs for describing a delivery process, but also allows to keep the external models concise and easy maintain as many information might be automatically deduced and do not need to be modeled in the external model. But the process planner not only reduces the amount of information required to describe a delivery process. It also allows the delivery system to meet Req-7: Best practices, as the delivery system respectively the planner can freely adapt a given delivery model. Of course the delivery system thereby still needs to meet Req-8: Traceability.

As expressed by Req-4: Self-Organizing we call the capability of the delivery system to adapt a given delivery model *self-organizing*. The following section details the concept of self-organization and provides concepts for the process planner. With this conceptual background section 7.3.2 briefly introduces related work. Afterwards we introduce our process planner design both from a static (section 7.3.3) and from a dynamic perspective (section 7.3.5).

7.3.1. Self-Organizing

One important challenge this thesis tries to tackle is C2 - Modeling Usability (cf. section 3.1). This means that the delivery system should make the delivery process modeling as straightforward and as easy as possible. Section 3.2 introduced several aspects to tackle this challenge. One important aspect is by providing self-organizing capabilities (cf. Req-4: Self-Organizing). In the context of artificial intelligence and system engineering self-organizing is defined as *"global order emerging from local interactions"* [HG03]. Thereby, self-organizing has the goal to reach a stable (optimal) state. In the delivery system domain, we define *self-organizing* as the process consisting of

1. Information retrieval through Self-Analysis
2. Self-stabilization using the retrieved information, i.e. to reach a valid delivery model
3. Self-optimization, i.e. adaption of a delivery model towards a specific goal

Self-Organizing then allows to fulfill Req-7: Best practices by specifying the corresponding goal. From a software-technical perspective, a dedicated component, the process planner, realizes the self-organization capability. We define *planning* as the process of self-organizing. Thereby, we differentiate two planning strategies, which have different scopes:

Model-based planning The model-based planning only relies on meta information (e.g. the delivery model, activity specifications) to plan the delivery model.

Project-based planning The project-based planning additionally uses project sources and other external sources (e.g.external services) as the source of truth.

Both planning strategies operate on the internal delivery model. Thereby, the following operations are allowed:

Adding a stage New stages can be added to the internal delivery model

Modifying a stage Planners may rename a stage or change related activities

Removing a stage Planners can completely remove a stage. Removing a stage does not necessarily imply the removal of related activities. Planners may move them to another stage

Adding an activity New activities may be added to the delivery model

Modifying an activity The activity configuration or the executionPrecondition can be modified

Adding a dependency Planners may add dependencies to existing activities in the delivery model. This includes both logical and functional dependencies

Removing an activity Existing activities can be removed from the internal delivery model

Postconditions of the planning process is a valid, i.e. executable, delivery model. Because of the architectural focus of this thesis, we do not provide formal foundations here (see future work (section 10.2).

7.3.2. Planner Types

Section 7.3.1 classified two planning strategies, namely the *model-based* and the *project-based* planning. They both differ in their knowledge base. While model-based planning only considers meta-information directly or indirectly contained in the delivery model, the project-based planning also uses information derived from for example project sources. Planners realizing either of the strategies have different characteristics as described in the following:

Model Planner use the model-based planning strategy. More precisely, they analyze the modeled delivery process activities, compare them to their activity specification and derive dependencies between those activities by e.g. performing constraint solving based on the artifact constraints defined in the activity specification (cf. section 5.2.5). That way, they eventually arrive at a valid delivery model. Overall, model planner are independent of the project and thus technology-agnostic.

Project Planner apply the project-based planning strategy. As such, they are typically technology-specific. An example for a project planner would be a maven multi-module planner that first detects if the project is a maven multi-module project and then substitutes parent project activities with subproject activities according to the project dependency tree. Each project planner requires the project sources a priori. Executing a project planner therefore is more expensive than model-based planning. On the other hand a project planner is also more powerful as it uses a larger knowledge base (project sources).

Related Work

To further motivate project planners (see above), this section briefly introduces a tool proposed by Google [Vak+15]. They identified a dependency problem of software builds, which they call *underutilized targets*. Typically, a software build comprises multiple targets that form a dependency structure. An underutilized target is a build target that contains files not required by some of its dependents. As such, it reduces modularity and negatively affects the build performance. To tackle the issue, they proposed two

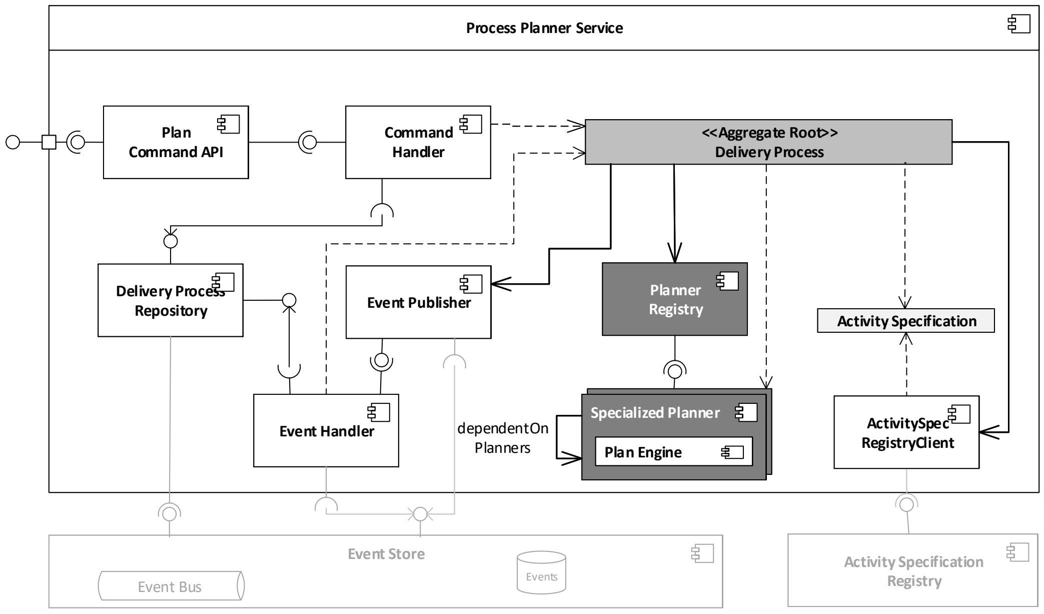


Figure 7.5.: Process Planner Component View

tools *decomposer* and *refiner*. Decomposer identifies underutilized targets and proposes decompositions for them and refiner provides the means to perform the refactorings. Using their tools Google found that at least 50% of the total execution time of tests could be saved as many test triggers could be saved.

Decomposer and refiner actively refactor the project sources (semi-automatically). They are therefore not directly applicable for our delivery system. But as they operate on project sources (build files), a project planner could incorporate their findings to decompose delivery process (build) activities into cohesive subactivities e.g. one per submodules which then help to meet an important delivery system principle: to provide fast feedback (cf. section 2.2.5).

Overall, their proposal suggests the great optimization potential project based planning can provide beside tackling C2 - Modeling Usability.

7.3.3. Component View

Figure 7.5 provides an overview of the process planner components. At it's heart is the delivery process aggregate (cf. section 5.2.1) which realizes the business logic. As a core service (cf. section 6.2) the process planner service applies the CQRS pattern (cf. section 5.2.4) and uses event sourcing (cf. section 5.2.3) for the persistence of delivery process aggregates. Thus, it's structure is similar to other core services. The *Delivery Process Repository* loads delivery process aggregates and applies the stored event stream on them. An *Event Publisher* enables the aggregate to publish new domain events and

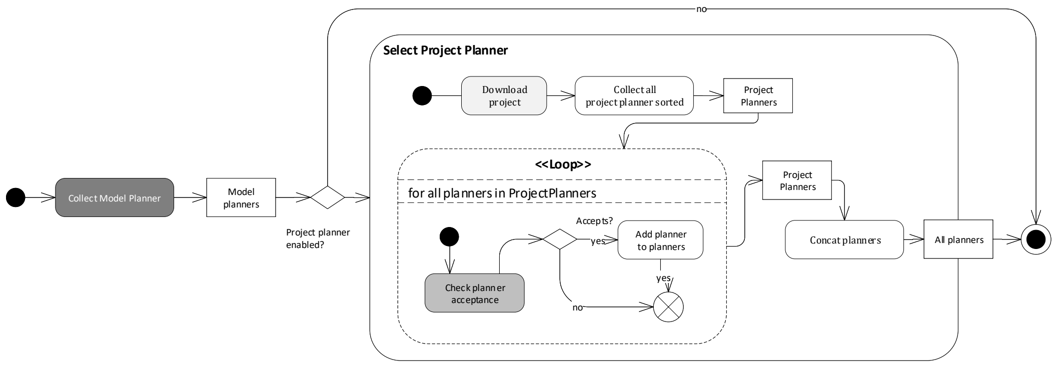


Figure 7.6.: Process Planner - Planner Selection

dedicated *Command Handler* orchestrate the non-business logic related execution of commands (e.g. transaction handling). As the process planner microservice is specifically tailored to our needs, it offers two commands: The planning of an internal delivery model and the execution trigger of an already planned delivery model.

Overall, central concern of the process planner obviously is to plan, i.e. organize and optimize a given delivery model. To keep the planning flexible and maintainable, we designed it to be hierarchical (cf. hierarchical planning [BC89]). The idea is to distribute the planning across a dynamically determined hierarchy of planners that are specialized in a certain area. The planners need to be determined dynamically as it depends on the project which planners are applicable. In general, there is a *Planning coordinator* that acts as the root planner. It is a model planner (see section 7.3.2) that calculates eventually missing dependencies between the modeled activities and checks the graph spanned by the activity dependencies for satisfiability. Considering our DDD approach, the delivery process aggregate itself acts as the coordinator. It then delegates the planning to specialized (project) planners. Each *Specialized Planner* can act as a coordinator too and use other sub-planners, thereby building a hierarchy with multiple levels. Section 7.3.5 provides more details about the dynamic planner selection and the planning process itself.

Since the design and the components of each planner depend on it's concrete planning scenario, figure 7.5 does not provide details for specific planners. Instead, each planner contains a *Planning Engine* which can be realized differently. One approach for the realization of planners are expert systems ([HMM88]). A planner would then comprise of a knowledge base and an inference engine. For simple process planners an alternative might also be a hard-coded handling, it depends on the use-case.

7.3.4. Planner Selection

Figure 7.6 details the planner selection process as performed by the planning coordinator (cf. section 7.3.3). In principle, it consists of two phases. First to collect all model planners (cf. section 7.3.2) from the planner registry and then to select matching project planners. The planning coordinator uses all model planner because they are project agnostic and therefore always applicable. Of importance is is their ordering. As depicted in figure 7.5 each concrete planner can specify other planners on which the planner depends. The planning coordinator must obey these dependencies while collecting the model planners. Technically, *topological sort* can be used to order the model planners in a correct sequence. After the model planners have been collected, matching project planners need to be selected. We divided the planner selection into these two phases as model and project planners have completely different runtime characteristics. While model-based planning only has minimal influences on the execution time, project-based planning can take a long time depending on the projects size. To this reason each delivery model can selectively enable or disable project-based planning to improve the planning performance.

If project-based planning is enabled, the procedure continue as follows: The planning coordinator initiates the project resource download. After all resources have been downloaded and extracted, the planning coordinator builds a file tree and collects all available project planner in the same way as described above. Each project planner then decides using the file tree if he is applicable for this project. In the positive case, the planning coordinator adds the project planner to the list of selected planners. Overall, the planning coordinator arrive at a valid planner sequence in the end.

7.3.5. Planning

Figure 7.7 provides an overview of the planning process. To have a holistic view, it also contains the execution trigger of a planned delivery model. Of course, the planning of a delivery model can be triggered without triggering it's execution (e.g. via the API). The event handler triggers both as this is the required behavior. It reacts to the ModelTriggered domain event of the external model aggregate and initiates the corresponding actions. Following our DDD approach, the business logic is encapsulated in the delivery process aggregate. The event handler acts as a process manager (cf. [HW03]) and orchestrates the business process activities. The modelTriggered business process requires to first plan the delivery process represented by the model and then to trigger the execution of this planned delivery model. Since the delivery process aggregate is scoped to a single delivery model, the event handler instantiates a new delivery process aggregate based on the received internal delivery model and initiates the planning, i.e. the event handler passes control to the delivery process aggregate. The delivery process aggregate acts as the planning coordinator (see above). First of all, it retrieves all activity specifications to allow for model-based planning. It then selects all applicable planners as described in the previous section. During this selection process the delivery process aggregate triggers the download of all project resources to allow for project-based planning.

The actual planning is divided into two phases. The first phase is devoted to model-based planning, as the model-based planning is far superior performance-wise compared to project-based planning. The delivery process aggregate hierarchically delegates the

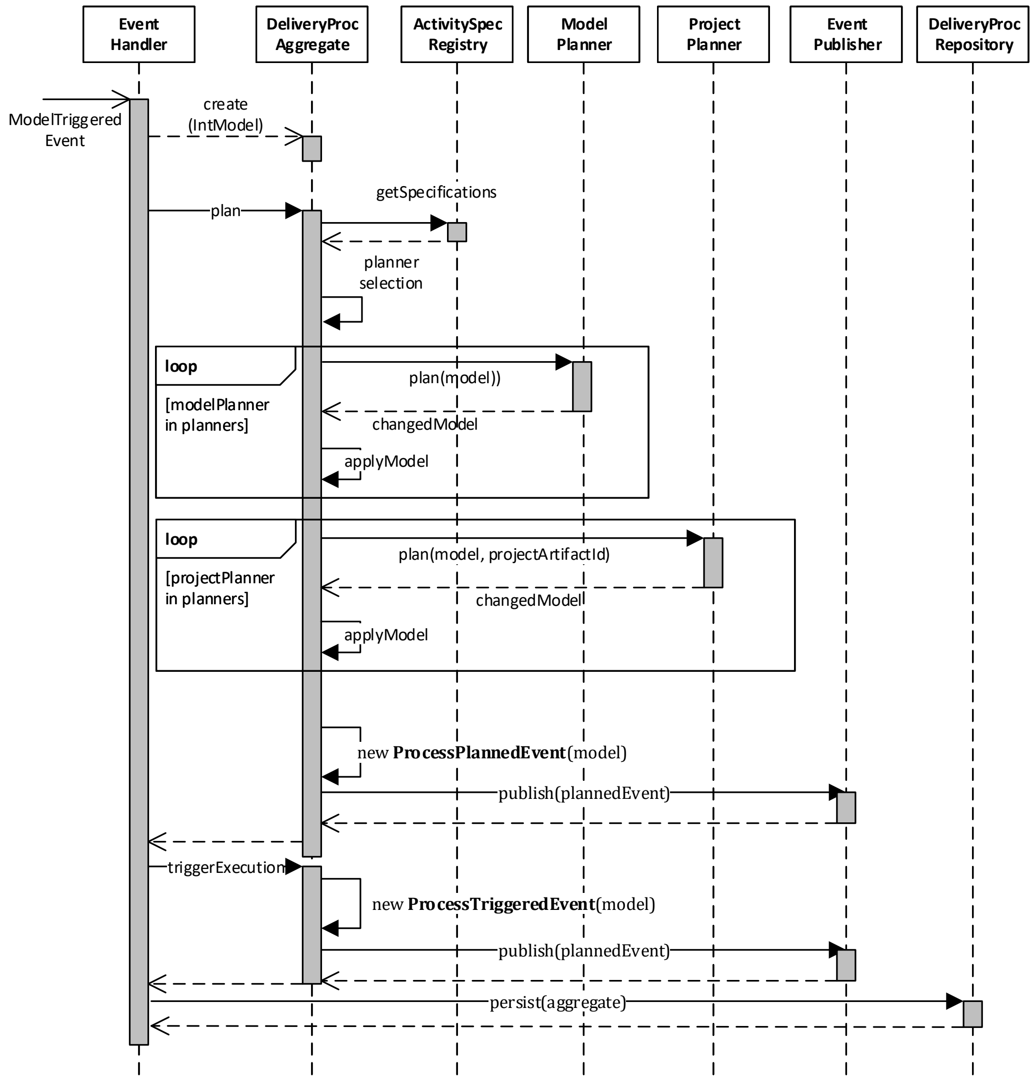


Figure 7.7.: Process Planner - Planning

planning to its model planners. Each planner gets the currently planned delivery model and the available activity specification. It is completely up to the planners to which scope they perform the planning. The delivery process aggregate only requires them to return a list of changes. The planners do not apply these changes themselves, they only return an updated model. This way, the delivery process aggregate can ensure consistency. Moreover, the delivery process aggregate also could publish planning domain events to allow for Req-8: Traceability even for the planning itself. As this is not required right now, we didn't modeled this aspect. After all model planners are done, the delivery process aggregate continuous planning using all applicable project-based planners. The planning itself works analogously to the model-based planning. When the project-based planning has finished, the delivery process aggregate published a *ProcessPlanned* event (cf. section 5.2.2) to indicate the successful planning. The delivery process aggregate does not need to validate the delivery before publishing the event as it consistently ensures validity when applying proposed changes by the planners.

After planning the event handler triggers the next activity which is the execution trigger of the just planned delivery model. Since the execution only need to be signaled, the delivery process aggregate publishes a *ProcessTriggered* event. Finally, the event handler persists the aggregate using the model repository.

Figure 7.7 only provided a successful model trigger sequence. In case of a failure, the event handler would publish a *ProcessPlanningFailed* event (cf. section 5.2.2). Section 6.3.7 provides more details on failure handling in general.

7.4. Orchestrator

The orchestrator provides the delivery process execution backbone. It thereby realizes the **DeliveryProcessExecution** aggregate from the core domain (section 5.1.1). Conceptually, the orchestrator is a *Process Manager* [HW03] that maintains the execution state of delivery process respectively of their individual activities and determines the next execution steps based on intermediate results. Instead of deciding for a centralized design, we could have opted for a decentralized, *choreographed* (cf. [New15]) approach. Section 9.3.5 discusses the rationales behind this decision and compares their advantages respectively disadvantages. The following details the internals of the orchestrator both from a static (section 7.4.1) and dynamic perspective (section 7.4.2).

7.4.1. Component View

Figure 7.8 depicts the orchestrator components. At it's heart is the *DeliveryProcessExecution* aggregate (cf. section 5.2.1), which encapsulates the delivery process execution business logic. In section 5.1.3 we identified the underlying subdomain as a generic one. Therefore the orchestrator is specifically designed to exploit this characteristic. Its design comprises three layers. One layer meets our specific requirements (e.g. Req-8: Traceability), another layer realizes the generic orchestration aspects and the third layer is

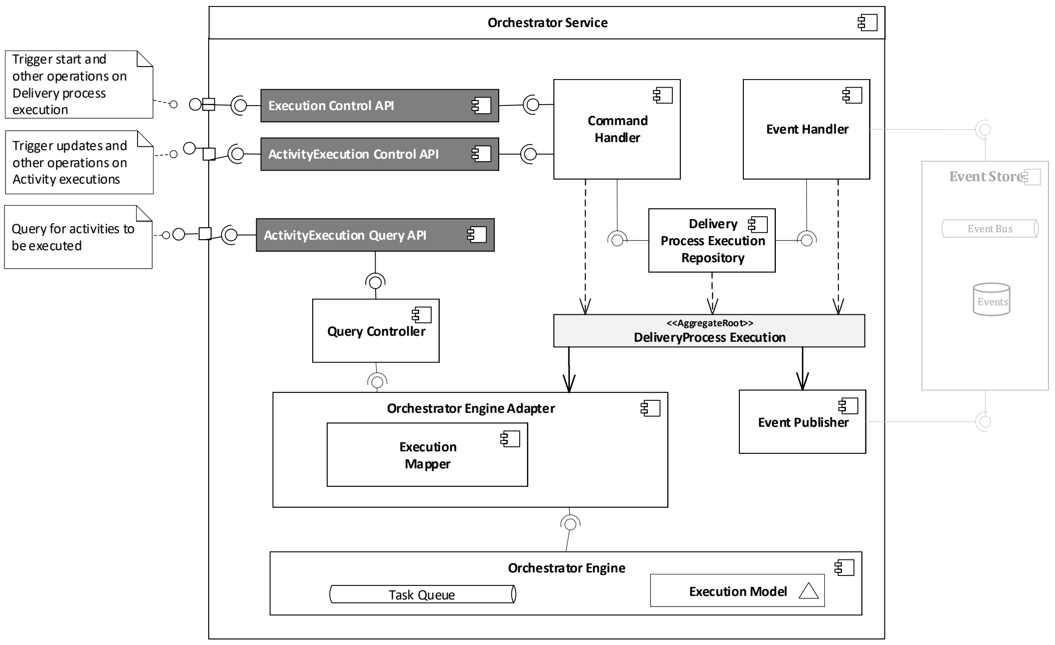

Orchestrator Service
Trigger start and other operations on Delivery process execution
Trigger updates and other operations on Activity executions
Query for activities to be executed
Execution Control API
ActivityExecution Control API
ActivityExecution Query API
Command Handler
Event Handler
Delivery Process Execution Repository
Query Controller
<<AggregateRoot>> DeliveryProcess Execution
Orchestrator Engine Adapter
Execution Mapper
Event Publisher
Orchestrator Engine
Task Queue
Execution Model
Event Store
Event Bus
Events

Figure 7.8.: Orchestrator Component View

an anti-corruption layer mediating between them. This design allows to use an arbitrary [1] orchestration engine, while still meeting our requirements. Figure 7.8 details our specific layer and the mediation layer. The mediation layer is realized by means of an *adapter* [Gam+02] depicted at the bottom of figure 7.8. Above the adapter are components belonging to the specific layer.

Everything enters the orchestrator through the first layer. It is structured similar to the CQS pattern ([MA89]). On one side commands trigger state changes. On the other side queries, i.e. read-only operations are provided. Although this sound like the CQRS pattern (section 5.2.4), internally the same model (with data transfer objects (DTO [Fow02]) for the query side) is used since all read and write concerns deal with the delivery process execution aggregate.

The command side comprises two API component. The *Execution Control API* allows to trigger operations regarding the delivery process execution itself, e.g. start, stop or pause. The *ActivityExecution Control API* allows to trigger updates on the activity execution, e.g. to start or complete an activity. Recapitulating chapter 6 activity services perform the activity execution, thus they are the primary consumers of this API. To query for new activities to be executed they consume the *ActivityExecution Query API*. This API directly is fed by the orchestrator engine adapter. This allows to let the specific orchestration engine handle concern like retry strategies or parallel execution. Since the

[1]as long as the concepts of the first layer can be mapped to concepts of the orchestration engine

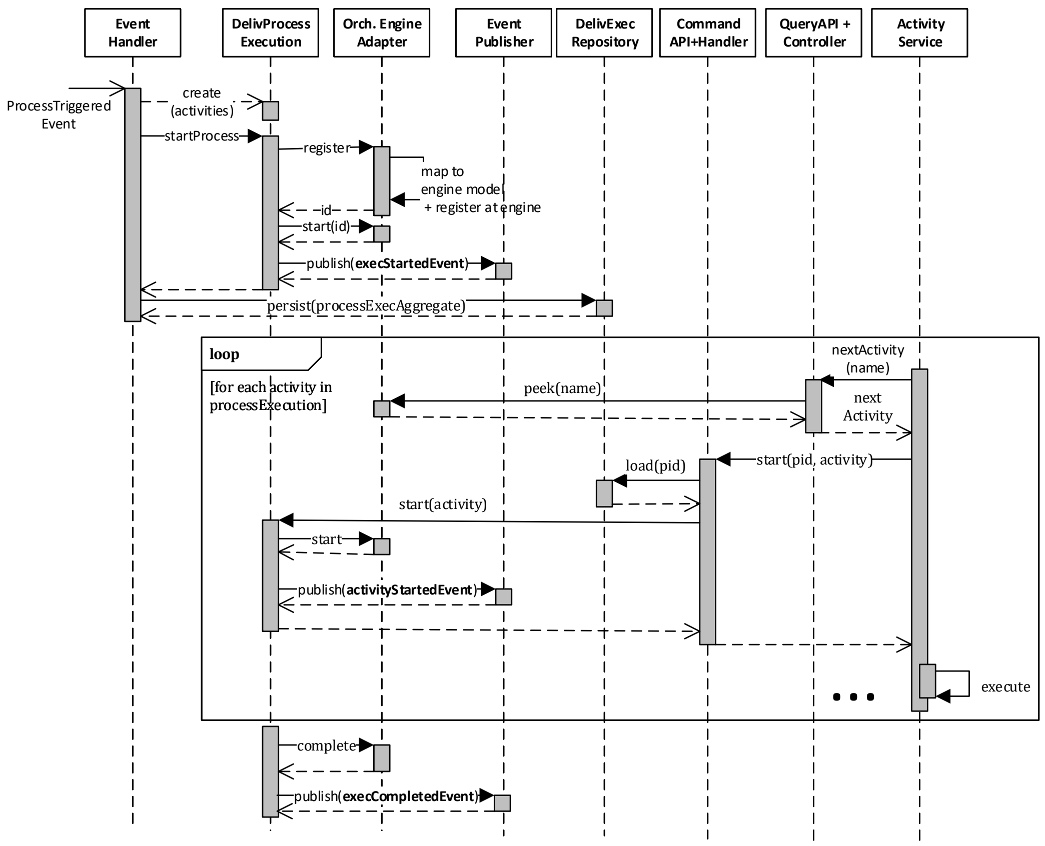


Figure 7.9.: Delivery Process Execution Overview

API is read only, the delivery process execution aggregate still can ensure consistency and use event sourcing, while the query side uses concepts for fast access (e.g. a queue).

Overall, the orchestrator design is decoupled from a concrete orchestration engine while meeting Req-8: Traceability which improves both flexibility and maintainability.

7.4.2. Delivery Process Execution

Figure 7.9 provides an overview of the delivery execution process. Following the inversion of control approach, the ProcessTriggeredEvent (section 5.2.2) containing the planned delivery model triggers its execution. Each execution of a delivery model corresponds to a new instance of a delivery process execution aggregate (cf. chapter 5). Thus, the corresponding event handler listening on processTrigger events creates a new aggregate instance when he receives such an event and delegates the business logic related to this aggregate. The delivery process execution aggregate first adds the model process to the orchestration engine by using the corresponding adapter and then starts the execution. Since the activity execution itself is performed by activity services, starting the execution basically means to assign activity service with their activities. Since the orchestration service is independent of a concrete orchestration engine, we do not provide details here. Typically, the first activities of the delivery process will be added to some kind of activity queue. After the delivery process execution has been started, the delivery process execution aggregate publishes a corresponding *ExecutionStarted* domain event and returns control back the the event handler. The handler finally persists the aggregate via the repository.

As mentioned above, the activity services are designed as polling consumers. Section 7.5 provides rationales for this decision. As such, they continuously request the orchestration service if some of their activities should be performed. Following our domain driven design approach, all state changing operations are handled by the delivery process aggregate. Polling for activities therefore does not change the state. The orchestration engine influences which activities should be performed next. The delivery process execution aggregate does not and shouldn't know if the engine supports parallel execution for example. The activity services therefore directly poll the orchestration engine adapter. If they decide to execute an activity they need to update (acknowledge) the corresponding activity execution at the delivery process aggregate. That way, all state changing operations are handled through the aggregate which ensures consistency. When an activity service requests to update an activity execution, the corresponding command handler loads the matching delivery process aggregate via its repository and triggers the appropriate action on the aggregate which publishes domain events in response. As these events range from starting to progress over completion events, figure 7.9 does not detail them. When all activity executions have been completed, the delivery process execution aggregate publishes an *ExecutionCompleted* domain event. In case of a failure the execution aggregate publishes corresponding failure events.

7.5. Activity Service

Activity services are the central architectural components to meet Req-1: Integration of new technology while supporting Req-3: Abstraction. As motivated in section 5.2.5 each activity (micro-) service encapsulates the coherent functionality of a single tool or service provider, thus focusing on a single concern providing loose coupling and high cohesion.

The activity services provide these functionalities based on the **Activity** concept from the core domain (section 5.1.1). Each activity is realized by means of the command pattern ([Gam+02]). It thereby either implements the **Transformation** or **Assessment** concept from the core domain. Section 7.5.2 provides more details about the activity realization.

Depending on the offered activities, an activity service is either a transformation, an assessment or a hybrid service.

Section 7.5.1 details static aspects of activity services and section 7.5.4 detail dynamic aspects.

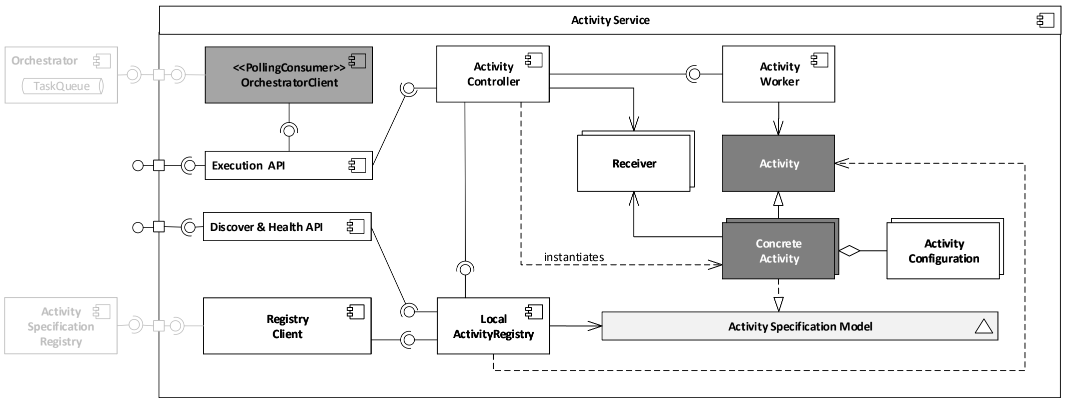


Figure 7.10.: Activity Service Component View

7.5.1. Component View

Figure 7.10 provides an overview of the activity service components. At it's heart are *Activities* (section 5.1.1). Each activity encapsulates a concrete functionality (e.g. git checkout) and has an *activity specification* (section 5.1.1) describing it's input and output schema. The activity service is responsible for providing the delivery system (more precisely the activity specification registry) with these activity specifications and to execute a concrete activity on demand. To be able to execute multiple activities in parallel the activity service uses *Activity Workers*. Each worker executes an activity in its own thread. The concrete activity thereby is transparent to the worker. A worker only knows the general activity interface. To archive this decoupling, an activity is realized by means of the command pattern. Thus, each activity contains all required information for execution including a realization of the *ActivityConfiguration* concept (section 5.1.1) and its *Receiver*, i.e. the object providing the required functionality. Section 7.5.2 provides more details about the underlying command pattern. Overall, it allows to easily extend an activity service with new functionality (activities).

To handle activity execution requests, an activity service has an *ActivityController*. The activity controller instantiates the requested activity and passes it to the activity worker for execution. Therefore, the activity controller needs to know all available activities and their specifications. The *ActivityRegistry* is responsible for this concern. It stores all activities the activity service provides. Using this registry, the *RegistryClient* registers all activity specifications at the ActivitySpecification registry (cf. section 6.2). Overall, the local activity registry decouples components that need to know all activity specifications from the concrete activities allowing to easily add new activities (cf. Open-close principle [LL10]).

The activity service has two interfaces to trigger the execution of activities. One is the *Execution API* to manually trigger the execution of activities. Section 7.5.3 details this API. The other one is the *Orchestrator Client*. It continuously polls the orchestrator

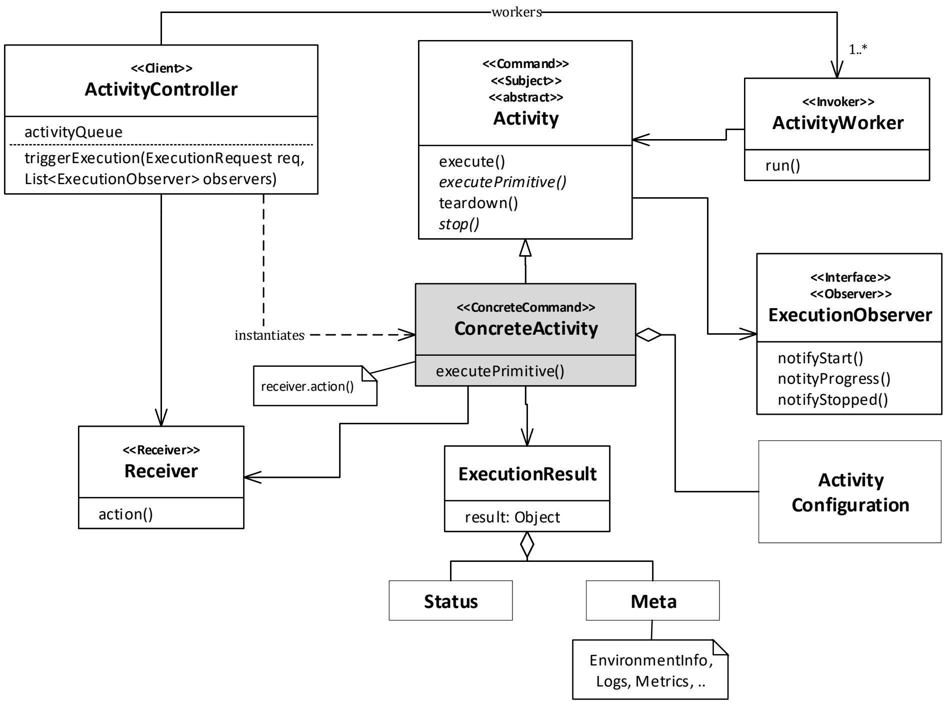


Figure 7.11.: Activity Service - Activity Design

if some activities should be executed. We decided to use polling, as this enables the activity service to decide on his own, when he is ready to accept new execution requests (cf. Polling Consumer [HW03]).

Since it is of great importance that the activity specifications stored at the Activity specification registry are up-to date, activity services offer an *Discover & Health API*. This API allows for health checks by the activity specification registry and to manually request the offered activity specifications.

7.5.2. Activity Design

The different activities of an activity service are realized by means of the command pattern [Gam+02]. Figure 7.11 details the design and provides the corresponding stereotypes. Each concrete activity encapsulates an execution request, e.g. git checkout. It contains the required configuration data (e.g. for our git example: repository URL and credentials) and knows what steps to perform. It then delegates the step execution to the corresponding receiver (e.g. gitClient). Overall, the encapsulation of an execution request with all required dependencies prevents hard-wiring the execution logic (decoupling of invoker and concrete command), allows to compose activities from others, provides an unified execution API and the possibility to asynchronously execute activities by using a queue for example. This property is utilized by the *ActivityController*. The activity controller handles execution requests triggered at the execution API or triggered by the polling orchestrator consumer. It translates the requests into the corresponding activity thereby validating the request against the activity specification and providing the activity with its requirements (receiver & configuration). The activity controller itself doesn't execute the activity. The execution is performed by means of *ActivityWorkers* which process the activity controller's activity queue. Beside instantiating the concrete activity, the activity controller also acts as a guard to prevent to much executions from being requested. If the activity queue reaches a certain threshold, the activity controller rejects incoming execution requests.

We designed the activity execution asynchronously since activities in the delivery process are typically long-running. Important aspect of asynchronous execution is to provide feedback. Especially in our concrete scenario the activity service needs to communicate execution updates to the orchestrator. Each activity therefore accepts one or multiple observers (cf. observer pattern [Gam+02]) to notify them about updates. The observers are passed from the calling context to the execution request at the activity controller. This allows to decide dynamically whom to notify about updates (e.g. orchestrator). This dynamic handling is required because it depends on the context whom to notify. In case of a manual trigger via the Execution API, i.e. outside a delivery process context, the orchestrator shouldn't be notified. Instead, the user might want to receive feedback directly. In case of a trigger through the OrchestratorConsumer, other observers are required to notify the orchestrator on updates. To ensure that the orchestrator is notified about the start and stop of an activity, the abstract activity base class applies the template method pattern ([Gam+02]). It's *execute* method notifies potential observers about the start, then calls the hook method of the concrete activity (executePrimitive) and finally notifies about the execution finish. That way, the concrete activity only needs to deal with the progress notification and it is ensured that the orchestrator gets notified independent on the concrete activity realization. Section 7.5.4 provides more details on this.

The execution of an activity produces an *ExecutionResult* which directly implements the ExecutionResult concept from the core domain (section 5.1.1). As depicted in figure 7.11, the ExecutionResult provides structured data, which eases its processing. Beside the result payload corresponding to the result schema of the activity specification, it also has a status (scheduled, running, failed, completed) and provides meta information. These meta information can contain environmental infos (e.g. for the git checkout scenario: commit message, author, etc.), metrics (e.g. checkout duration, repository size, etc.) and logs. All these information provide value to the user and follow directly from the core domain (section 5.1.1).

Beside offering an execute method, each activity could also implement a tear-down operation. This fact is especially useful for non side-effect free transformations (especially deployment related).

7.5.3. Execution API

Central functionality of the activity service is to provide and execute activities. These activities can either be triggered through the orchestrator consumer (cf. section 7.5.1) or manually through the execution API. The manual trigger allows to use the offered functionality outside the delivery system context e.g. to support developers in their daily work. The following specifies the execution API using a popular interface description language, Corba IDL ([Obj17]). Each method is described after the listing. Overall, the execution API allows to trigger activities as a remote procedure call ([HW03].

```
module ExecutionAPI {
    interface ActivityServiceExecutionAPI {
        ExecutionResult executeActivity(in
           ActivityExecutionRequest executionRequest);
        string triggerActivityExecution(in
           ActivityExecutionRequest triggerRequest);
        ExecutionResult getProgress(in string executionId);
    };
    interface ActivityExecutionRequest {
        readonly attribute string activityName;
        readonly attribute Map configuration;
    };
};
```

Source Code 7.2: Activity Service Execution API

executeActivity triggers the activity execution specified in the executionRequest and blocks the request until the activity has been completed. Thereby, the activity is identified by its activity name. Since the activity execution always is performed asynchronously (cf. section 7.5.2), the API acts as an observer to block the request until the activity has finished. It then returns the activity result (cf. section 7.5.2).

triggerActivityExecution triggers the activity execution specified in the executionRequest and returns an execution id to allow for progress tracking.

getProgress allows to track the progress of the execution corresponding to the given execution id. It returns the execution result. Recalling section 7.5.2 the execution result can also be an intermediate result if the execution is still running.

7.5.4. Execution

This section details important dynamic aspects of activity services. Main concern of activity services is to provide activities for execution in the delivery system context. Relating to the execution is the announcement of activity specifications for the offered activities. As the announcement is relatively self-explanatory (scanning for activities & calling the registration API (section 7.1.1)) we do not provide further details here. Instead, we focus on the activity execution aspects. As described above, we use the command

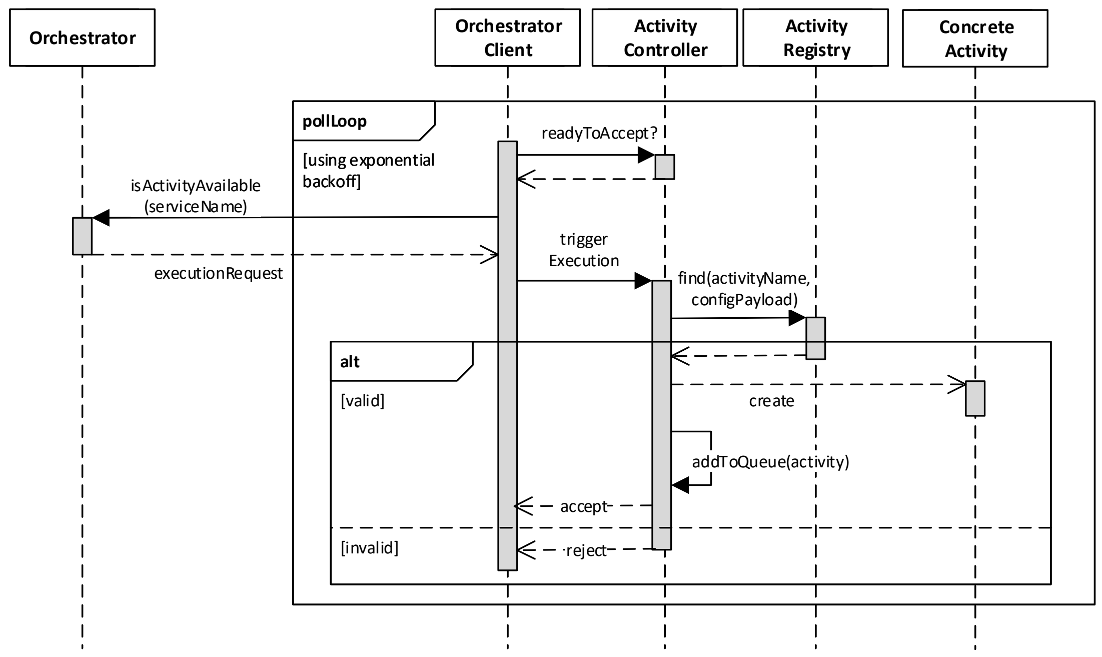


Figure 7.12.: Activity Service - Orchestrator Polling Consumer

pattern for the activity realization. This not only decouples the relevant software components, but also allows to describe the initialization and the execution procedures for the same reasons separately. The activity execution can be triggered both manually and by the orchestrator (cf.). Since the procedure works analogously, Section 7.5.4 details the initialization of activities (commands) triggered by the orchestrator. The manual triggering works analogously. Section 7.5.4 then describes how these activities are executed.

Activity Initialization

Figure 7.12 details the orchestrator polling process. The activity services thereby continuously checks via the orchestrator's Query REST API (section 7.4.1) if new activities to be executed are available. We designed the communication to be synchronous and in a polling based manner for two reasons. One the one hand we want the activity service to decide both the interval and the point of time when to poll, namely when the service is ready. Hohpe and Woolf define this type of consumer *Polling consumer*. On the other hand we want to have minimal requirements when developing an activity service because activity services are the central architectural unit which allow to extend the offered functionality of the delivery system. Given the popularity of REST-based APIs ([New15]) most developers are familiar with this communication style.

As section 7.5.1 mentioned and detailed in the following, the activity services uses an worker queue internally to manage the activity executions, thus the orchestrator communication could easily be adapted to use message or another asynchronous style.

The polling process is depicted in figure 7.12. The orchestrator consumer ensures that the activity controller is ready to process new activity execution requests. In the positive case, the orchestrator consumer queries the orchestrator for new activities to be executed. Thereby the activity service of course only queries for activities targeted to himself by providing it's service name. The orchestrator uses this name to filter matching activities. If the activity controller is not ready to process further requests, the orchestrator consumer aborts the aborts polling loop and tries the same procedure again in the next next loop. To prevent overflowing the orchestrator, the orchestrator consumer uses exponential back-off in case of a failure of the orchestrator.

If the orchestrator activity query returns an activity execution request, the orchestrator consumer triggers the execution at the activity controller. The activity execution request contains the same data as the manual request (cf. section 7.5.3), i.e. activity name and configuration payload. The controller queries the activity registry if an activity exists with the given name and with a configuration model (cf. activity specification section 7.1 accepting the given payload. If an activity matching the request exists, the activity controller instantiates the corresponding activity and provides it with all required dependencies (configuration and receiver, cf. section 7.5.1). The activity controller then adds the activity to his activity queue for processing by activity workers. Section 7.5.4 details the worker execution process.

If no matching activity (different name or configuration model) exists, the activity controller rejects the execution trigger. Since the orchestrator query API is read only and does not actually remove execution request from the orchestrator queue, the orchestrator consumer does not need to initiate further actions. Another activity service with the same name (maybe in another version) might accept the execution request. Otherwise the orchestrator detects that a request is not being processed and reacts accordingly.

Activity Execution

The previous section detailed the process on how incoming activity execution requests are handled. The following resumes this process from the activity worker perspective. In the concrete scenario depicted by Figure 7.13 we assume that the activity to be executed has an orchestrator client as observer, i.e. it was requested by the orchestration consumer. This allows to model the interaction with the orchestrator.

The process is as follows: Each activity worker tries to take an activity from the activity controller execution queue. The take operations blocks until an activity is available. When an activity is available, the corresponding worker starts the execution by invoking the activities execute method. As described in section 7.5.1 the activity execute method is designed as a template method to ensure that observers are notified about start and finish. Thus, the activity first notifies its observer about the execution start. The observer tries to acknowledge the activity execution start at the orchestrator. The observer could reject the start since another activity service might have already been started the execution. In case of a rejection, the observer immediately stops the activity execution by calling the

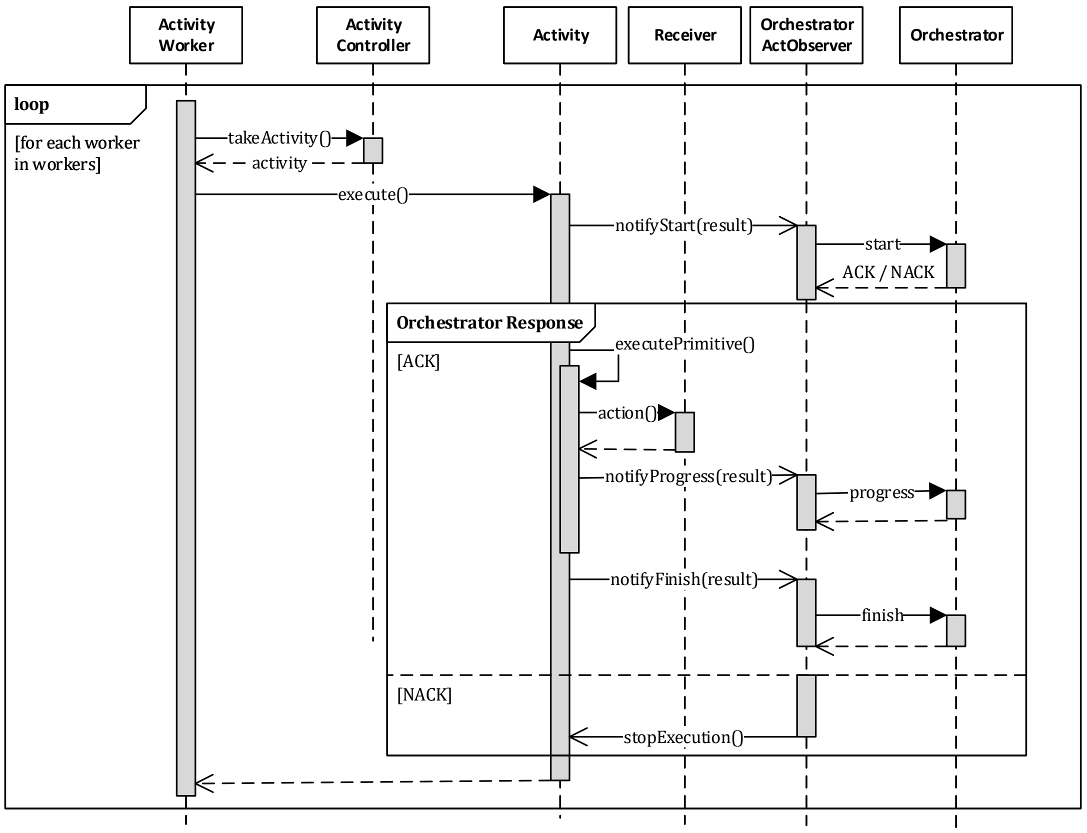


Figure 7.13.: Activity Service - Activity Worker

activity stop method to save resources. If the orchestrator acknowledges the execution, the observer does not intervene. Since typically the observer will acknowledge the request, we decided not to wait for the confirmation to improve the execution performance. In case of a rejection, the activity reverts its changes, thus not waiting on the orchestrator provides more benefits in our opinion. The approach is similar to *optimistic execution* [Get+04].

After having notified its observers, the activity base class calls the hook method of its specializations. During the execution of the hook method, the activity might notify it's observer about updates. Contrary to the observer pattern, we decided to pass the data along with the notification as each activity produces the same result respectively has the same result data structure. This then ensures that observers get consistent data. In our modeled scenario, the observer communicates the progress to the orchestrator. When the hook method is finished, the activity base class notifies about its observers and passes control back the the activity worker, which starts querying the activity queue again [2].

7.5.5. Quality Gate Service

Our core domain provides the concept of a **Quality Gate** (section 5.1.1). Conceptually, a quality gates is a relatively generic function: It accepts an artifact, a report containing interpretable characteristic of this artifact and a policy defining admissible characteristics (cf. section 5.1.1). The function then either accepts or rejects its input artifact based on the characteristic interpretation.

As part of our architecture we provide a default quality gate service, which can handle four different interpretations. It thereby assumes, that the characteristics are at least on an ordinal scale, i.e. they can be ordered. The quality gate service supports the following interpretations:

- Equality: The reported value must be equal to the given policy value
- Non-Equality: The reported value must not be equal to the given policy value
- Threshold$_{(+)}$: The reported value must be larger (with respect to the scale type) than the given policy value.
- Threshold$_{(-)}$: The reported value must be smaller (with respect to the scale type) than the given policy value.

From an implementation point of view, the quality gate is a normal activity service, thus we don't provide further details here. The quality gate activity input thereby is realized by means of the *ActivityConfiguration* concept.

[2] figure 7.13 does not contain this loop to prevent cluttering the figure

7.6. Summary

The chapter detailed important core components & services. Most importantly, it provided insights into our core microservices, i.e. the model service, the process planner and the orchestrator. These service assemble the core of the architecture. They all rely on Event-Sourcing for their persistence, which eases Req-8: Traceability. To reduce their complexity, they apply the CQRS pattern, which allows them to only focus on write, i.e. state-changing concerns. The view concerns are handled by a dedicated service not detailed is this chapter that integrates all their domain events. Beside the advantages of being able to easily add another view representation without changing the core services, this approach reflects the user demands best in our opinion, namely to have an integrated representation across the whole delivery process.

Main responsibility of the model service is to import external delivery models into the delivery system. It thereby converts these models into internal delivery models to decouple the process planners from concrete external delivery models. To prevent information loss during this model transformation, the process planner can still access the external model via project-planners. Project planners were introduced as specific components of the process planner that uses the project sources as their knowledge base for planning. Beside project planners there are model planners. Model planners only rely on meta information, i.e. the internal delivery model and activity specification to perform planning. Overall, the planning process then is a two phase - process consisting of a model-based planning phase to rudimentary plan a given delivery model a project-based planning phase that allows for more sophisticated optimizations. After planning, the orchestrator microservice, which wraps a generic orchestration engine, executes the modeled process. The execution thereby is perform by means of activity service, which this chapter also introduced. Activity Service provide abstraction for activities of the delivery process. Each activity service thereby encapsulates the coherent functionality of a single tool, thus focusing on a single concern providing loose coupling and high cohesion. The functionality is exposed by means of activities that can be referenced in the delivery model. Since activity services provide the architectural hotspot to integrate new technologies, our design tries to expose minimal requirements. Therefore, activity service neither use messaging nor event sourcing in their design. Only requirement is a REST-based communication with the orchestrator and the activity specification registration, which both can be accessed through a discovery service.

Using the presented design of the core services, the next chapter will describe our prototypical implementation. In addition it details a pipeline description language to define external delivery models and provides a java-based activity service framework to quickly bootstrap new activity services.

8. Implementation

Contents

Following a Domain Driven Design approach Chapter 6 and chapter 7 presented an architecture design for a Self-Organizing Delivery System. This chapter presents an implementation of these concepts. Thereby, we not only built a prototype for evaluation purposes in chapter 9, we also defined a pipeline description language (section 8.2) and realized a framework for quickly bootstrapping new activity services in a Spring (Java) context. Section 8.4 provides details the framework. Section 8.1 details the scope and technical details of our prototype.

8.1. Prototype - Scope & Technologies

This section briefly introduces our prototype we implemented based on the design detailed in the previous chapters. Section 3.1.1 defines the prototype scope, as we did not implement all concepts and section 8.1.2 presents the technologies we used for the realization.

8.1.1. Scope

section 3.1.1 Because of the limited scope of this thesis, we could not implement all concepts and components presented in the previous chapters. We focused on being able to

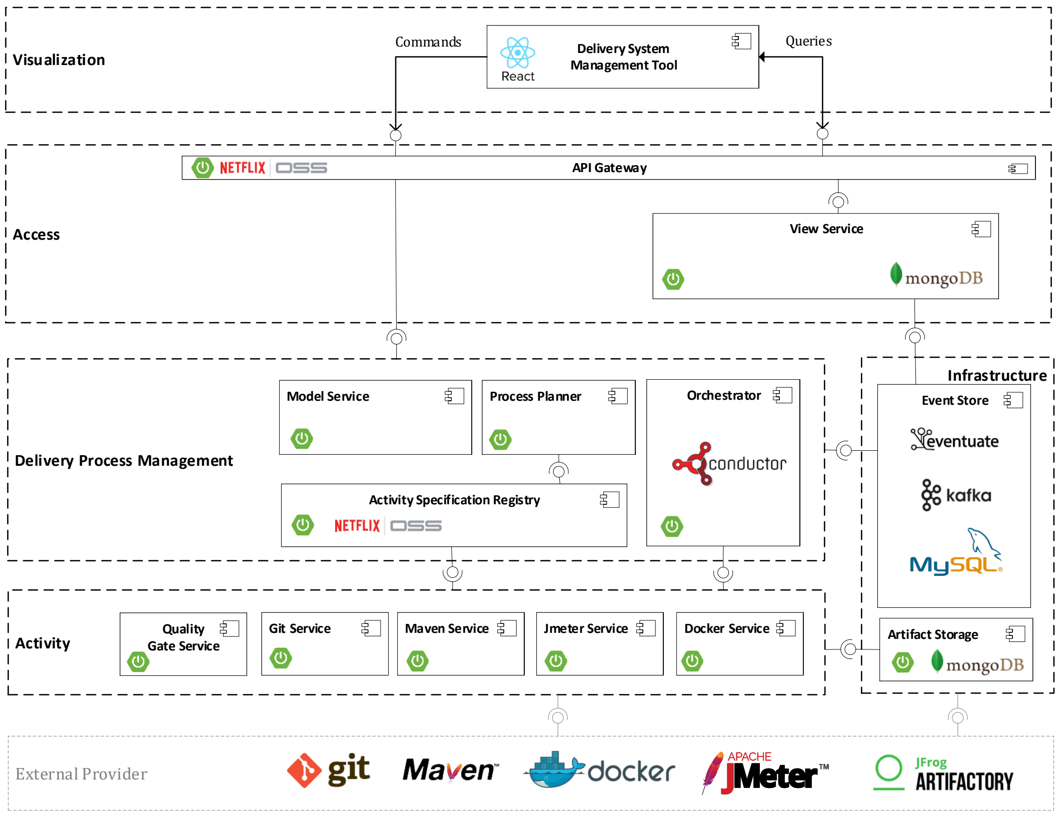


Figure 8.1.: Prototype Overview - Technologies and Frameworks

evaluate our requirements. Therefore, we did not implement advanced concepts like the execution precondition (section 5.1.1), which for example provides semantics to tear-down provisioned system in case of a failure. We also did not implement a notification service. For the realization of the activity specification we use JSON schema [1], which is similar but not equivalent to our activity specification design. Regarding the Self-Organization made several assumptions to ease implementation of planners. We assume that each delivery model only has one project repository and that the providing activity of a specific artifact datatype (e.g. java-classes) is unique. Overall, some design concepts emerged during development of the prototype or later during the thesis, thus the realized prototype should not be seen as a one-to-one realization of our concepts.

8.1.2. Technologies

Figure 8.1 provides an overview of our prototypical delivery system. Primarily, the microservices are realized with Spring Boot [Pivb]. Each service is packaged in a docker

[1]http://json-schema.org/

[Doc] container. The following details each service from a technological perspective.

Delivery System Management Tool The delivery system management tool is the react.js [Facb] based frontend application of our delivery system. We chose react.js as it allows to create responsive, interactive user interfaces. Since delivery process activities might be long running we use websockets to provide the user with up-to date information without distracting the users workflow. Following the CQRS approach (cf. section 5.2.4) the UI provides a task-based structure. Section 8.5 provides screenshots of the delivery system management tool.

API Gateway An API Gateway is a typical pattern in a microservice architecture to provide clients with a single entry point. We decided to use components from the well-known and battle-tested Netflix OSS stack. Thus, we use Netflix Zuul as our our API gateway and the service registry is realized by means of Netflix Eureka.

View Service The view service is a spring-boot based application. It uses mongoDb for persistence since document-oriented storage fits nicely with aggregates and provides good performance. So far, the view service only supports the delivery process visualization in the internal delivery model.

Model Service As all our custom developed microservices, the model service is realized with Spring Boot. In the current configuration stage, the model services supports the import of external delivery models from local file system and from git. But since the remote resolving is implemented by means of activity services, any transformation service could be used to resolve a remote model. So far, the model service supports our external model described in section 8.2.

Process Planner The Spring Boot based process planner service comprises several planner components. So far, we realized two model planners and one project planner (cf. section 7.3.2). One model planner automatically determines the parameter mapping between two given activities and the more sophisticated model planner automatically determines dependencies between activities without requiring the user to specify them. The project planner is a maven planner that detects sub-projects and automatically expands the delivery model for these sub-projects. In addition, the realized process planner service provides validation capabilities based on the activity specification. Section 8.3 provides more insight into both the different planner types and the validation.

Activity Specification Registry Foundation of the activity specification registry is Netflix Eureka from the Netflix OSS [Netc] which we extended by our activity specification. Because of the limited scope of this thesis we utilize the activity specification registry also as the discovery service (cf. section 6.2).

Orchestrator The Spring Boot based orchestrator uses Netflix Conductor [Netb] as its orchestration engine (section 7.4). Conductor is a java application, thus we embedded conductor into the orchestrator service to save resources. In a productive scenario we advice to run a dedicated conductor instance.

Event Store For the event store we rely on the Eventuate Local stack [Eve]. It provides an event bus realized with kafka [Apaa] and a mysql database for persisting the events [Orab]. We chose eventuate as is provides a spring framework which allows to build application based on event-sourcing and CQRS.

Artifact Store The artifact store relies on mongoDB for both persisting file meta information and the files itself. We rely on mongoDB's GridFS to store the binary data. Although there is no limit in the file size, we advise to store files directly on the file system for better performance in production. In this prototypical context we used GridFS for convenience reasons.

Quality Gate Service The quality gate services also is a Spring Boot based microservice. Right now, it supports the same policy interpretations as specified in the external model (treshold+, treshold-, equality, non-equality cf. section 8.2).

Git Service The Git Service is a transformation service. It is based on Spring Boot and uses JGit internally to provide git-related functionality. So far, it provides a preview activity (get content of file in remote repository), a search activity (find file matching the glob pattern in a remote repository) and a checkout activity (checkout specified branch and upload it to the artifact store).

Maven Service The maven service is a hybrid service, i.e. it both provides transformations and assessment activities. Internally, it uses MavenEmbedder for the maven related functionality. It offers three transformation: A compile activity, an assemble activity and a deploy activity. For the assessments it provides a code coverage activity.

JMeter Service The jmeter service is an assessment service based on Spring Boot. It provides both a general purpose *jmeter* activity to perform specified jmeter tests against a specified target and a specialized *jmeterTT* activity which only requires some special parameter. We developed this tailored activity for our case study in chapter 9.

Docker Service Another transformation service we developed is the spring boot based docker service. It requires a docker socket to perform it's functionality. Right now it offers two activities tailored for chapter 9. *BuildTTGateway* builds several docker containers for our evaluation project and *provisionTTGateway* starts all required container while waiting for dependent containers. Chapter 9 provides more details.

8.2. Pipeline Description Language

Section 5.1 identified the separation of a delivery model into an external and an internal model as an important concept to allow for self-organizing capabilities (Req-4: Self-

Organizing) and to support different pipeline description languages (Req-5: Custom PDLs).

This sections details such an external model by presenting a pipeline description language (PDL). We choose to align it narrowly with the core domain to be able to more easily reason about the system in chapter 9. The following defines the language in BNF (Backus-Naur-Form):

⟨deliveryprocess⟩	⊨	stages: ⟨stages⟩ transformations: ⟨activities⟩ | stages: ⟨stages⟩ transformations: ⟨activities⟩ assessments: ⟨activities⟩ qualityGates: ⟨qgates⟩
⟨stages⟩	⊨	- ⟨stage⟩ | - ⟨stage⟩ ⟨stages⟩
⟨stage⟩	⊨	name: ⟨name⟩ transformations: ⟨transformationlist⟩
⟨transformationlist⟩	⊨	- ⟨name⟩ | - ⟨name⟩ ⟨transformationlist⟩
⟨activities⟩	⊨	- ⟨activity⟩ | - ⟨activity⟩ ⟨activities⟩
⟨activity⟩	⊨	name: ⟨name⟩ ref: ⟨name⟩ configuration: ⟨configuration⟩ | name: ⟨name⟩ ref: ⟨name⟩ dependsOn: ⟨dependencies⟩ | name: ⟨name⟩ ref: ⟨name⟩ dependsOn: ⟨dependencies⟩ configuration: ⟨configuration⟩
⟨dependencies⟩	⊨	- ⟨dependency⟩ | - ⟨dependency⟩ ⟨dependencies⟩
⟨dependency⟩	⊨	alias: ⟨name⟩ ref: ⟨ref⟩
⟨configuration⟩	⊨	⟨parameter⟩ | ⟨parameter⟩ ⟨configuration⟩
⟨parameter⟩	⊨	⟨name⟩:⟨name⟩
⟨qgates⟩	⊨	- ⟨qgate⟩ | - ⟨qgate⟩ ⟨qgates⟩
⟨qgate⟩	⊨	strategy: ⟨strategy⟩ policy: ⟨rules⟩
⟨rules⟩	⊨	- ⟨qgate⟩ | - ⟨qgate⟩ ⟨qgates⟩
⟨rule⟩	⊨	name: ⟨name⟩ interpretation: ⟨interpretation⟩ assessmentRef: ⟨ref⟩ valueRef: ⟨name⟩ setPoint: ⟨name⟩
⟨strategy⟩	⊨	auto
⟨interpretation⟩	⊨	threshold+ | threshold+ | equality | non-equality
⟨ref⟩	⊨	p://⟨path⟩
⟨path⟩	⊨	⟨name⟩ | ⟨name⟩/⟨name⟩
⟨name⟩	⊨	⟨char⟩ | ⟨num⟩ | ⟨char⟩⟨name⟩
⟨char⟩	⊨	a . . . z | A . . . Z
⟨num⟩	⊨	0 . . . 9

A delivery model begins with the definition of *stages*. At least one stage must be defined. Each stage has a *name* and a list of transformation names. These transformations names reference transformation defined later in the model. We choose not to inline transformations in the stage definition to allow easy inheritance from other models. The stage definition must at least reference one transformation. After the stage definition, transformation are defined indicated with the keyword *transformations*. At least one transformation is required. Each transformation has a *name* to reference it in the model and to visualize it later on. In addition each transformation references a certain activity with the keyword *activityRef*. Optional, dependencies of this transformation can be specified (*dependsOn*). Each dependency has an *alias* to be able to easily reference it. The references activity is specified via the *ref* keyword. Here a valid URI is expected to be able to reference activities defined in other delivery models. Configuration for a transformation is defined via *configuration*. Each configuration entry consists of key-value pairs separated with a colon. After transformations have been defined, assessments can be specified optionally. They are defined identically to transformations. Finally, quality gates can be defined with the keyword *qualityGates*. The pipeline description language originated from an earlier prototype version without process planners, thus quality gates are defined differently than transformations or assessments. A quality gate comprises a *strategy* that defines how the model processing should determine the execution order. Following our quality gate service (cf. section 7.5.5), each quality gate directive has a *policy* comprising of at least one rule. Each rule has a *name*, an *interpretation* (treshold+, treshold-, equality, non-equality) which defines how to compare set-point (specified by *setPoint*) and actual value. The actual value is defined by referencing an assessment via an URI (*assessmentRef*) and by defining the name of the measurement produced by the assessment (*valueRef*).

8.2.1. Model Example

```
stages:
  - name: analyse
    transformations:
      - checkout
transformations:
  - name: checkout
    service: git-service
    activity: checkout
    configuration:
      repositoryUri: https://github.com/...git
assessments:
  - name: java-linecount
    service: utility-service
    activity: file-linecount
    dependsOn:
      - alias: repo
        ref: p://this/transformations/checkout
    configuration:
      files: @repo
      filter: *.java
qualityGates:
  - strategy: auto
    policy:
      - name: AvgJavaFileLength
        interpretation: threshold-
        assessmentRef: p://this/assessments/java-linecount
        valueRef: avgLines
        setPoint: 100
```

Source Code 8.1: Sample Delivery Model Instance

Listing 8.1 shows an example delivery model. It models a single *analyse* stage, which performs a checkout of files from git (transformation), calculates the average line count of java files in this repository (assessments) and promotes the repository artifacts, if their average line count is below 100.

8.3. Process Planner

We developed three planners during the prototype development: Two model-based planners and one project planner. These planners are detailed in section 8.3.1 and section 8.3.2, respectively. Important responsibility of the planners is Req-6: Model Validation (cf. section 7.3). Section 8.3.2 briefly discusses the validation support of our prototype.

8.3.1. Model-based Planning

We developed two model planners during the prototype development. In principle they function the same way, thus we only present the more sophisticated model planner. Listing 8.2 provides an external delivery model excerpt. In the depicted situation, the user provided all dependencies and the configuration mapping, e.g. the user assigned the workspace property of the assemble activity to the workspace output field of the checkout activity (*dependsOn: repo..* and *workspace: @repo/workspace*).

```
transformations:
# - ...
  - name: assemble
    service: maven-service
    activity: assemble
    configuration:
      workspace: @repo/workspace
      classes: @compile/classes
    dependsOn:
      - alias: repo
        ref: p://this/transformations/checkout
      - alias: compile
        ref: p://this/transformations/compile
# - ...
```

Source Code 8.2: Manual Delivery Model

```
transformations:
# - ...
  - name: assemble
    service: maven-service
    activity: assemble
# - ...
```

Source Code 8.3: Self-Organized Delivery Model

Our developed model planner uses the activity specification to automatically determine this configuration, i.e. listing 8.2 and listing 8.3 describe the same delivery process.

The model planner analyses the specification of all activities defined in the external model to determine which activity provides data of a certain datatype that another activity requires. Listing 8.4 provides an excerpt of the *maven assemble* activity input specification. As depicted, it requires a workspace property of type *artifact* with a *workspace* constraint. The model planner analyses all other activities defined in the model. In our case there is a *git checkout* activity. Its activity result specification is provided in listing 8.5. As depicted, git checkout provide a property of type *artifact* with the same *workspace* constraint. Thus, the model planner automatically relate both activities. Overall, our model planner applies basic constraint solving using the specified types and constraint in the activity specification.

```
"paramSchema": {
   "$schema": "...",
   "title": "Maven Assemble In",
   "type": "object",
   "additionalProperties": false,
   "properties": {
     "workspace": {
        "type": "string",
        "ptype": "artifact",
        "constraint": {
          "type": "workspace"
        }
     },
    ...
    }
}
```

Source Code 8.4: Assemble Input Schema

```
"returnSchema": {
   "$schema": "..."
   "title": "Git Checkout Result",
   "type": "object",
   "additionalProperties": false,
   "properties": {
     "workspace": {
        "type": "string",
        "ptype": "artifact",
        "constraint": {
          "type": "workspace"
        }
     }

   }
}
```

Source Code 8.5: Checkout Result Schema

8.3.2. Project-based Planning

For evaluation purposes we also developed a project-based planner. As stated in section 7.3.2 they analyze the referenced project and are therefore typically technology-specific. In our case, we developed a maven planner which analyzes maven project on potential submodules and their dependencies. If submodules are found, the maven project planner splits the project parent activities into multiple activities (one per submodule) and adapts the dependencies following the maven dependency tree. Figure 8.2 provides an example. The depicted delivery model describes a basic build process. It comprises five activities. First, the current project sources need to be checked out, afterwards the project is compiled, tested and packaged. Finally, there is a quality gate to promote or reject the packaged artifact(s). The underlying maven project comprises 10 submodules (common, communicator, cds, core, bridge, gmanagement, web-v1, web-v1-1, web-v2 and ear) where some modules depend on others. The maven project planner analyzes their dependencies and adapts the delivery model correspondingly as depicted in the lower part of figure 8.2. It thereby not only adapts the compile, assemble and test activity but also introduces new quality gates to follow shift-left best practices to provide fast feedback (cf. chapter 2). We further elaborate this characteristic in chapter 9.

Validation

An import goal of the delivery system is to meet Req-6: Model Validation. Thus, we also included validation capabilities in our prototype. At the current stage, validation errors only show up in the log. The user is presented a generic error message in the

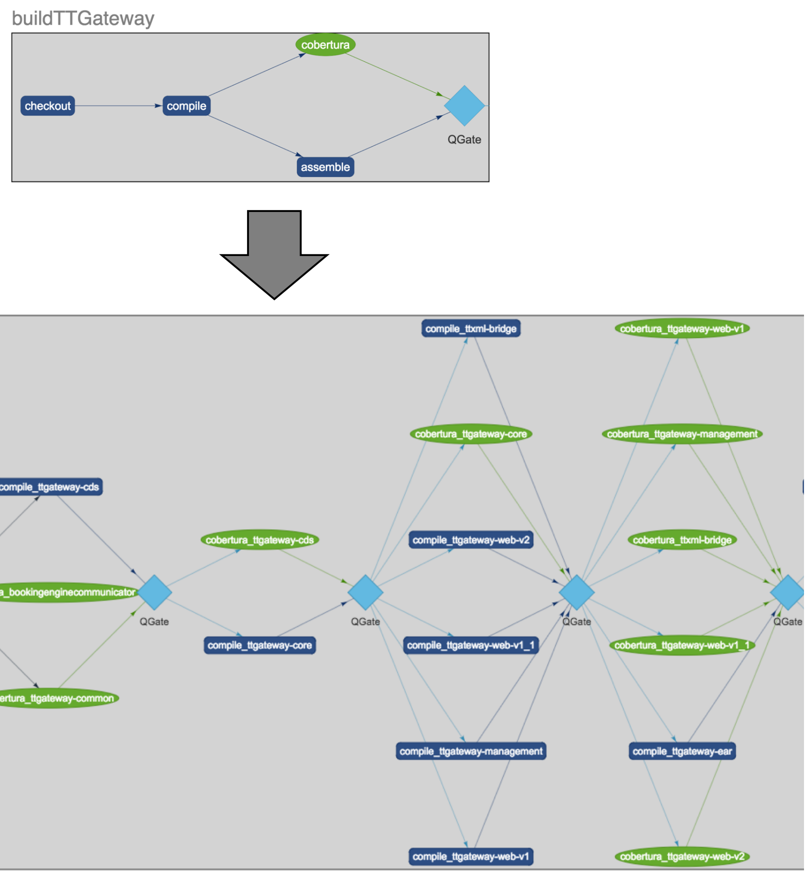
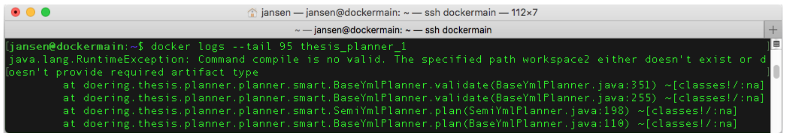


Figure 8.2.: Project-based Planning


jansen — jansen@dockermain: ~ — ssh dockermain — 112×7
~ — jansen@dockermain: ~ — ssh dockermain

```
jansen@dockermain:~$ docker logs --tail 95 thesis_planner_1
 java.lang.RuntimeException: Command compile is no valid. The specified path workspace2 either doesn't exist or d
oesn't provide required artifact type
        at doering.thesis.planner.planner.smart.BaseYmlPlanner.validate(BaseYmlPlanner.java:351) ~[classes!/:na]
        at doering.thesis.planner.planner.smart.BaseYmlPlanner.validate(BaseYmlPlanner.java:255) ~[classes!/:na]
        at doering.thesis.planner.planner.smart.SemiYmlPlanner.plan(SemiYmlPlanner.java:198) ~[classes!/:na]
        at doering.thesis.planner.planner.smart.BaseYmlPlanner.plan(BaseYmlPlanner.java:110) ~[classes!/:na]
```

Figure 8.3.: Process Planner - Validation

delivery system management tool. Figure 8.3 depicts an example validation error. In the modeled case we tried to reference a parameter that does not exist. Future work is of course required to improve the validation experience, but it suffices to evaluate the validation capabilities in chapter 9.

8.4. Activity Service Framework

Activity services play a key role in the delivery system. They not only provide means for Req-3: Abstraction and Req-6: Model Validation, but they are also the key to meet Req-1: Integration of new technology. To integrate a new tool or service provider an activity services needs to be developed. Thus, it is of great importance to make the development as easy as possible.

To this reason we realized a Spring framework to take the heavy lifting. This framework enables developers to concentrate on the business logic instead of implementing delivery system specifics. Recalling section 7.5 an activity service needs to provide the specifications of its activities, register them at the activity specification registry, provide health check support and poll the orchestrator for tasks. If a task is available, the activity service needs to instantiate the corresponding activity, execute it, provide progress and finally must return the result to the orchestrator. At best, the activity service should also be able to execute multiple activities in parallel. Our frameworks tackles all these concerns. Only requirement is to include the frameworks jar for example via maven (see listing 8.6).

```xml
<dependencies>
...
  <dependency>
    <groupId>doering.thesis</groupId>
    <artifactId>pipeline-service-starter</artifactId>
    <version>1.7</version>
  </dependency>
...
</dependencies>
```

Source Code 8.6: Maven - Activity Starter Dependency

If the library dependency is included, the only missing part is to implement the business logic. Listing 8.7 provides a sample activity implementation. Most importantly, the developer need to annotate his class with *@PipelineActivity* and provide the activity name and type (cf. line 1). The framework uses reflection to find all classes annotated with this annotation. The class needs to implements the generic *Activity* interface with a type parameter matching it's execution result class. (cf. line 2). Using this class, the framework can automatically determine the activities return schema. To determine the configuration schema, the classes constructor needs to contain a specialization of *ActivityConfiguration* (cf. line 6). If the activity realization requires other dependencies (e.g. a git client (line 6), they be provided in the constructor too. Our framework uses dependency injection to provide a matching instance during instantiation.

As the class implements the activity interface, it needs to implement an execute method returning the parametrized type (cf. line 12). The execute method has an *ExecutionMonitor* to allow to provide feedback and meta information during execution. Basically this is everything which is required to implement an activity. The framework automatically registers this activity at the activity specification registry, polls the orchestrator, instantiates the activity with all required dependencies and triggers the execute method and submits the result to the orchestrator. Listing 8.8 details the configuration and result class.

```
1   @PipelineActivity(name = "checkout", type =
        ActivityType.TRANSFORMATION)
2   public class GitCheckoutCmd implements
        Activity<GitCheckoutResult> {
3       private final GitCheckoutConfiguration configuration;
4       private final GitClient git;
5
6       public GitCheckoutCmd(GitCheckoutConfiguration
            configuration, GitClient git) {
7           this.configuration = configuration;
8           this.git = git;
9       }
10
11      @Override
12      public GitCheckoutResult execute(ExecutionMonitor
            monitor) {
13          // implement me
14      }
15  }
```

Source Code 8.7: Example Activity Implementation using our framework

To enable the framework to generate an meaningful activity specification, the developer has to provide some more information on the configuration or result class itself. As stated above, the framework uses reflection to analyze the classes. Thus, it automatically can detect the property data types for the activity specification. But for certain constraints, additional information need to be provided via annotations. Such a constraint could be for example to mark a property as being required. This can be done via *@JsonProperties(required = true)* as depicted in Listing 8.8 (line 2). Beside constraints, the activity specification model supports custom data types (cf. section 5.2.5). These custom data types are heavily used by model planners to determine dependencies between activities (cf. section 7.3.2). Such an information is also provided via annotation. Developer can either rely on predefined annotations or can define their own. In line 9 of Listing 8.8 the predefined workspace annotation is used to define the data type as an artifact of type *workspace*.

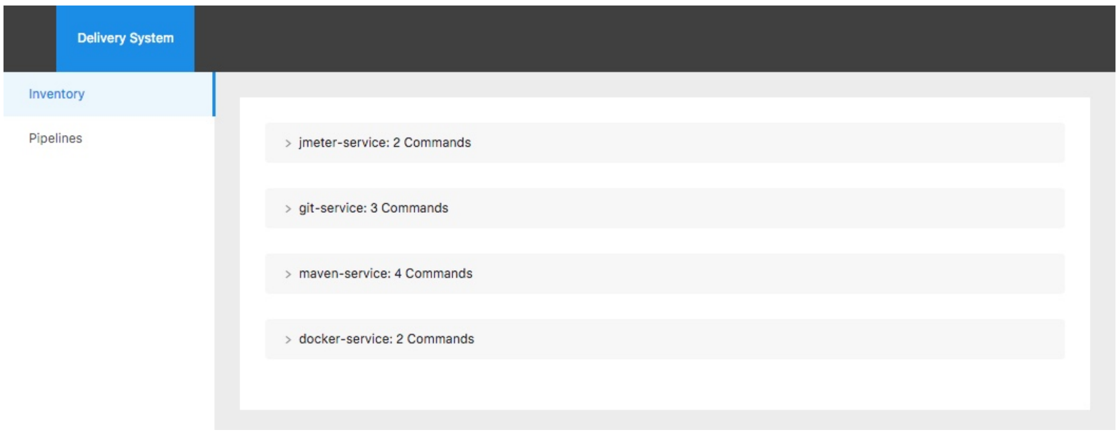
```java
1  public class GitCheckoutConfiguration implements
       ActivityConfiguration {
2      @JsonProperty(required = true)
3      private String repositoryUri;
4      private String branch;
5      //more properties
6  }
7
8  public class GitCheckoutResult implements ExecutionResult {
9      @Workspace
10     private String workspace;
11 }
```

Source Code 8.8: Activity Configuration and Result Class

8.5. Delivery System Management

To get an impression of the functionality of our prototype, we present screenshots of the central functionalities in the following.

8.5.1. Activity Inventory & Activity Specifications



> jmeter-service: 2 Commands

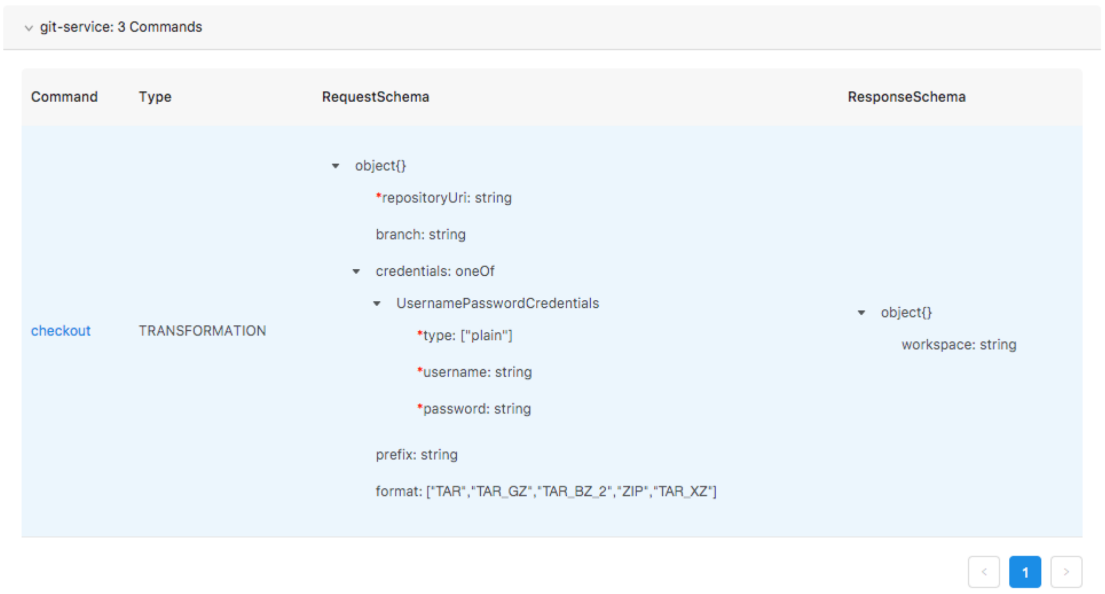
> git-service: 3 Commands

> maven-service: 4 Commands

> docker-service: 2 Commands

Figure 8.4.: Management Tool - Activity Inventory

Beside offering administrative tasks for delivery processes, one important functionality of the delivery system management tool is to provide an overview of available activities and their specifications. Although not further elaborated in this thesis, a delivery


git-service: 3 Commands

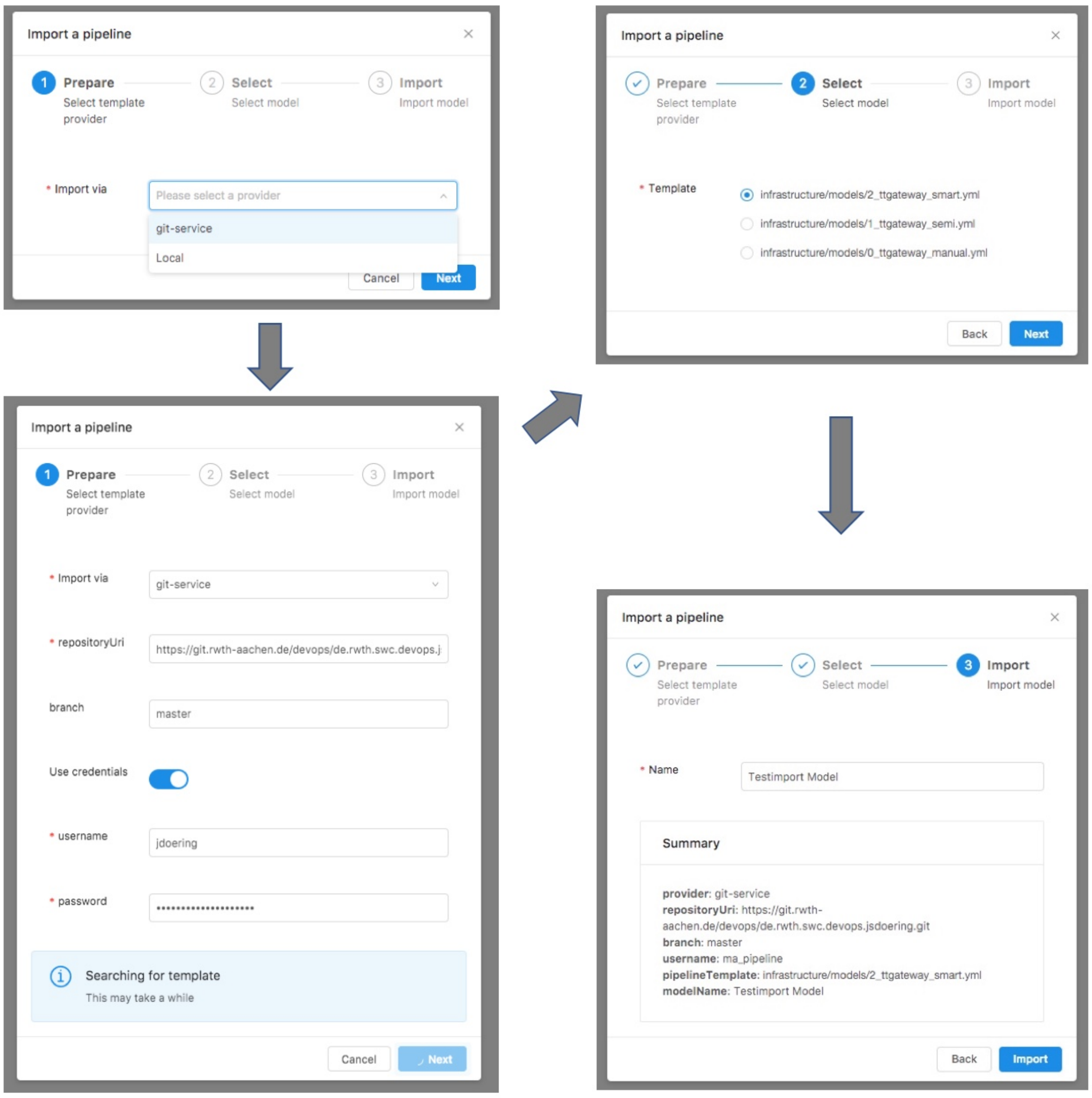
Command	Type	RequestSchema	ResponseSchema
checkout	TRANSFORMATION	object{} *repositoryUri: string branch: string credentials: oneOf UsernamePasswordCredentials *type: ["plain"] *username: string *password: string prefix: string format: ["TAR","TAR_GZ","TAR_BZ_2","ZIP","TAR_XZ"]	object{} workspace: string

Figure 8.5.: Management Tool - Activity Specification

system has different stakeholders. Developers provide activity services and pipeline engineers use the offered functionality to realize delivery process. Thus, it is important for pipeline engineers to have an inventory of available activities in the delivery system which helps in tackling C2 - Modeling Usability. Figure 8.4 depicts the activity inventory of our prototype. The offered activities are grouped under their corresponding activity service. When expanding for example the git service, all it's activities and their activity specification are listed. Figure 8.5 illustrates the specification of the *checkout* activity. As depicted, it's configuration model is relatively comprehensive. The only mandatory parameter is a repository URI though (as indicated by the red star). The branch, (path) prefix or checkout format are optional. In addition credentials can be provided. As depicted, the prototype only supports UsernamePassword credentials in its current stage.

8.5.2. Model Import

Figure 8.6 depicts the subsequent dialogs when a user wants to import an external model into the delivery system. First the user needs to select a model provider. Recalling section 7.2 the model service differentiate between local and remote references. Since the prototype only has a single service to import remote resources - the git service - the user therefore can select git-service or local import. As illustrated in the second dialog, the user has to provide some configuration details. The delivery system management tool thereby dynamically generates the corresponding form based on the activity specification. When the user confirms the configuration, the delivery system searches for compatible models from which the users needs to select one (third dialog). Finally, the delivery system presents an overview of all provided data and requests the user to provide a model



Import a pipeline

* Import via: Please select a provider

- git-service
- Local

Import a pipeline

* Import via: git-service

* repositoryUri: https://git.rwth-aachen.de/devops/de.rwth.swc.devops.j

branch: master

Use credentials: on

* username: jdoering

* password: ••••••••••••••••••••

Searching for template

This may take a while

Import a pipeline

* Template:

- (●) infrastructure/models/2_ttgateway_smart.yml
- ( ) infrastructure/models/1_ttgateway_semi.yml
- ( ) infrastructure/models/0_ttgateway_manual.yml

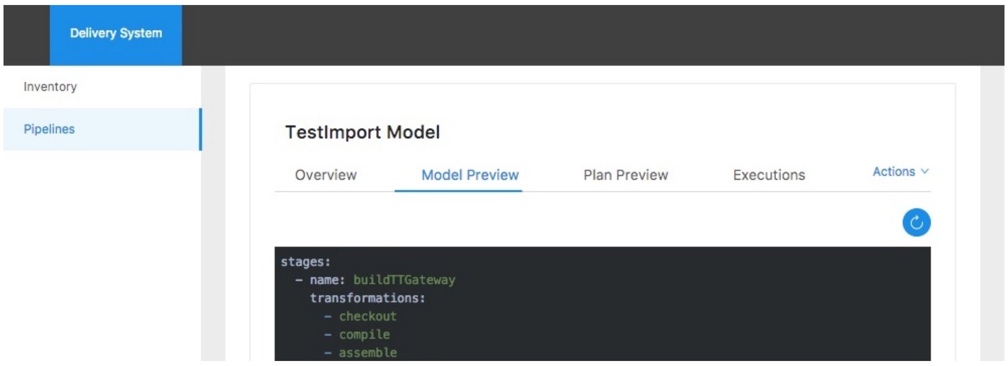
Import a pipeline

* Name: TestImport Model

Summary

provider: git-service
repositoryUri: https://git.rwth-aachen.de/devops/de.rwth.swc.devops.jsdoering.git
branch: master
username: ma_pipeline
pipelineTemplate: infrastructure/models/2_ttgateway_smart.yml
modelName: TestImport Model

Figure 8.6.: Management Tool - Model Import



```
stages:
  - name: buildTTGateway
    transformations:
      - checkout
      - compile
      - assemble
```

Figure 8.7.: Management Tool - Model Preview

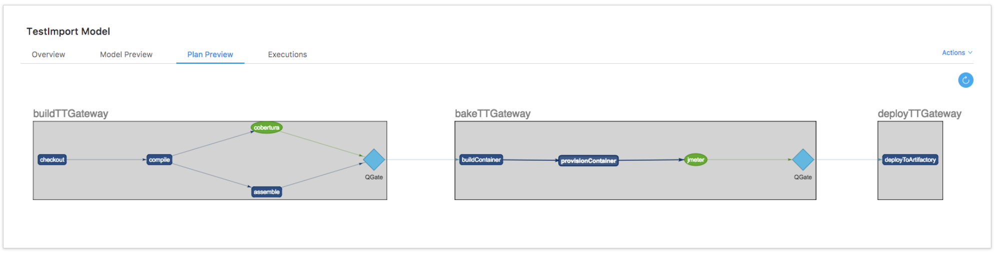


Figure 8.8.: Management Tool - Plan Preview

name to easily identify the reference later on. The user then can trigger the import.

8.5.3. Existing Model

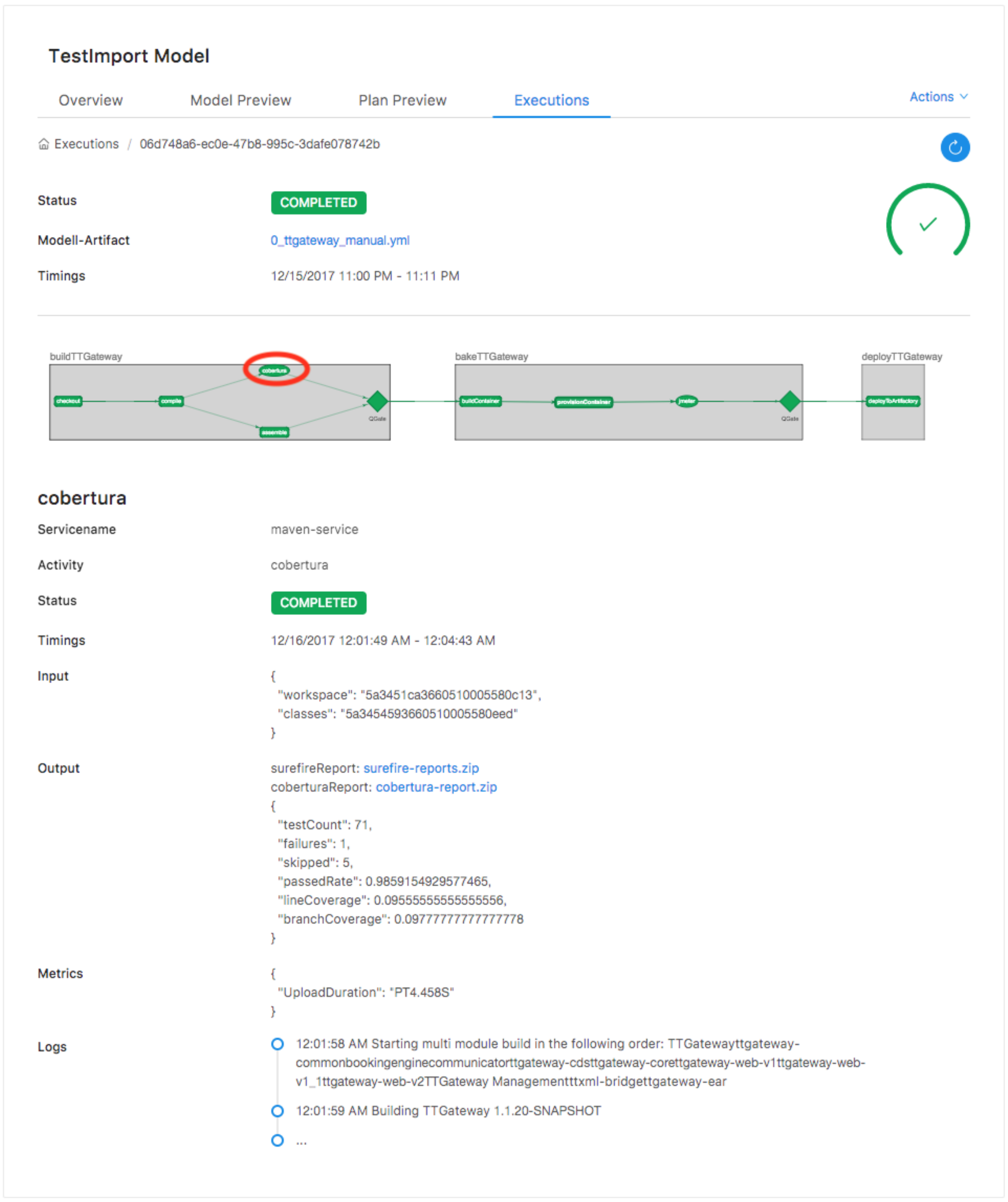
All imported models are listed under *Pipelines* in the delivery system management tool. Each model card provides four tabs. An overview which basically provides the import information, an model preview, a plan preview and an execution tab. Figure 8.7 depicts the model preview tab. Here, the user can initiate an external model fetch to see the external model in its current version.

8.5.4. Plan Preview

Another tab of a model card (cf. section 8.5.3) is the plan preview (see figure 8.8). It presents a preview of the planned delivery process for the current external model. This enables users to easily verify if the delivery process is planned as intended. Each node in the delivery process graph can be selected to get further information about the planned activity.

8.5.5. Execution

The final tab of a model card is the execution tab. It provides a list of all executions for this model reference. Figure 8.9 depicts a successful execution. Beside the delivery



Status COMPLETED

Modell-Artifact 0_ttgateway_manual.yml

Timings 12/15/2017 11:00 PM - 11:11 PM

cobertura

Servicename maven-service

Activity cobertura

Status COMPLETED

Timings 12/16/2017 12:01:49 AM - 12:04:43 AM

Input

```
{
 "workspace": "5a3451ca3660510005580c13",
 "classes": "5a34545936605100055800eed"
}
```

Output

```
surefireReport: surefire-reports.zip
coberturaReport: cobertura-report.zip
{
 "testCount": 71,
 "failures": 1,
 "skipped": 5,
 "passedRate": 0.9859154929577465,
 "lineCoverage": 0.09555555555555556,
 "branchCoverage": 0.09777777777777778
}
```

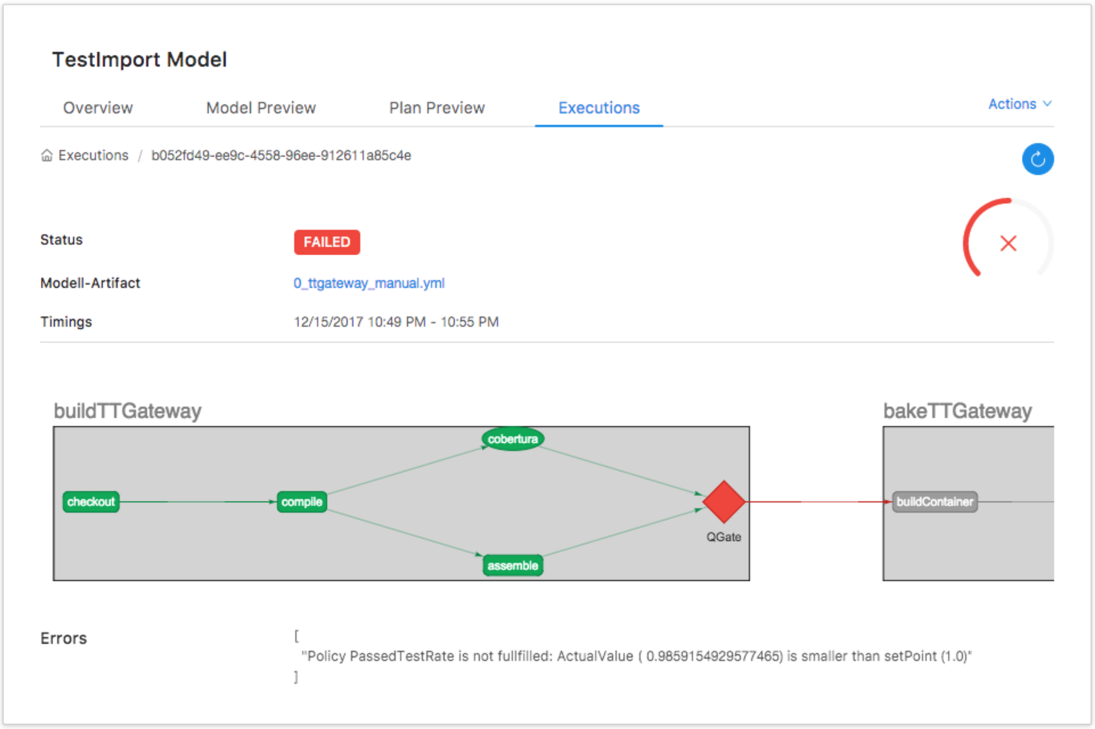
Metrics

```
{
 "UploadDuration": "PT4.458S"
}
```

Logs

```
12:01:58 AM Starting multi module build in the following order: TTGatewayttgateway-
commonbookingenginecommunicatorttgateway-cdsttgateway-corettgateway-web-v1ttgateway-web-
v1_1ttgateway-web-v2TTGateway Managementttxml-bridgettgateway-ear
12:01:59 AM Building TTGateway 1.1.20-SNAPSHOT
...
```

Figure 8.9.: Management Tool - Execution Overview



Executions / b052fd49-ee9c-4558-96ee-912611a85c4e

Status FAILED

Modell-Artifact 0_ttgateway_manual.yml

Timings 12/15/2017 10:49 PM - 10:55 PM

Errors

```
[
 "Policy PassedTestRate is not fullfilled: ActualValue ( 0.9859154929577465) is smaller than setPoint (1.0)"
]
```

Figure 8.10.: Management Tool - Execution Failed

processes graph-based representation, the execution view also contains detail information like the timings for example. To allow for Req-8: Traceability, the user can download the exact external model used. To get more information, the user can select each activity in the delivery process graph. In figure 8.9 the cobertura activity is selected. Its details are depicted below the delivery process graph. Basically, the provided information correspond to the executionResult (cf. section 7.5.2) and the input configuration. As input the cobertura activity got two artifact ids. The output comprises downloadable surefire and cobertura reports. Moreover it details the test count, the number of failure and skipped tests and also the test passed rate as well as coverage values. Below the output are metrics. The cobertura activity published one metric, namely the upload duration of it's artifacts. These metrics could be used for example to detect issues with the delivery system itself but also might serve as an indicator for delivery process issues. At the bottom are log entries the cobertura activity published during execution. This especially helps for root cause anlysis.

Of course, a delivery process also might fail. Figure 8.10 depicts such a situation. In that case, the delivery system management tool highlights the failed activity and provides reason for the failure. In the concrete scenario, a policy was not fulfilled: *"Policy PassedTestRate is not fullfilled: ActualValue (0.9859154929577465) is smaller than setPoint (1.0)"*).

8.6. Summary

In this chapter we provided a small glimpse of our practical work performed during the thesis. Major part was the implementation of a prototype that realizes great parts of the architecture design. During the prototyp development, we additionally implemented an activity service framework to ease the development of new activity services. This framework allows to bootstrap new Spring-based activity service basically by means of an annotation. Thereby, it automatically generates activity specifications by using reflection and registers these specifications at the activity specification registry. In addition it provides the orchestrator integration. Therefore, developers can concentrate their development effort on the activity logic, which eases development of new activity services significantly. To be able to evaluate our architecture and concepts by means of the prototype in the next chapter, we also designed a pipeline description language. This language is closely aligned to our core domain to ease evaluation. As a means to explore Self-Organizing capabilities, we developed multiple model-based and one project-based planner. The model-based planners can automatically derive dependencies between activities and adapt the delivery model accordingly. The project-based planner is a maven planner, that incorporates maven submodules into the delivery model of the project's parent. The planner thereby optimizes the delivery model in terms of fail-fast. Using our prototype, we conducted a case study to evaluate our requirements, which the next chapter presents.

9. Evaluation

Contents

C1 - Project Evolution and C2 - Modeling Usability are two major challenges delivery systems face (cf. chapter 3). Following a domain driven design approach (cf. chapter 5), we proposed an architecture (cf. chapter 6) and detailed important core services (chapter 7) to tackle those challenges. Based on this design, we built a prototype (cf. chapter 8) which this chapter uses to evaluate our concepts and design, i.e. to assess their applicability and quality. The IEEE defines quality as *the degree to which a system, component, or process meets specified requirements* [IEE02]. Therefore, section 9.2 evaluates the architecture in terms of our top level requirements (section 3.2). To gain insights, we conducted a case study, which section 9.1 details. Section 9.3 discusses observations made during the case study which are not directly applicable to the case study objectives.

9.1. Case Study

A case study is an exploratory research methodology, i.e. it's purpose is to find out what is happening, to seek new insights and to generate new input for further research [RH09]. The following presents our case study. Thereby we follow the reporting structure as recommended by P. Runeson and M. Höst [RH09].

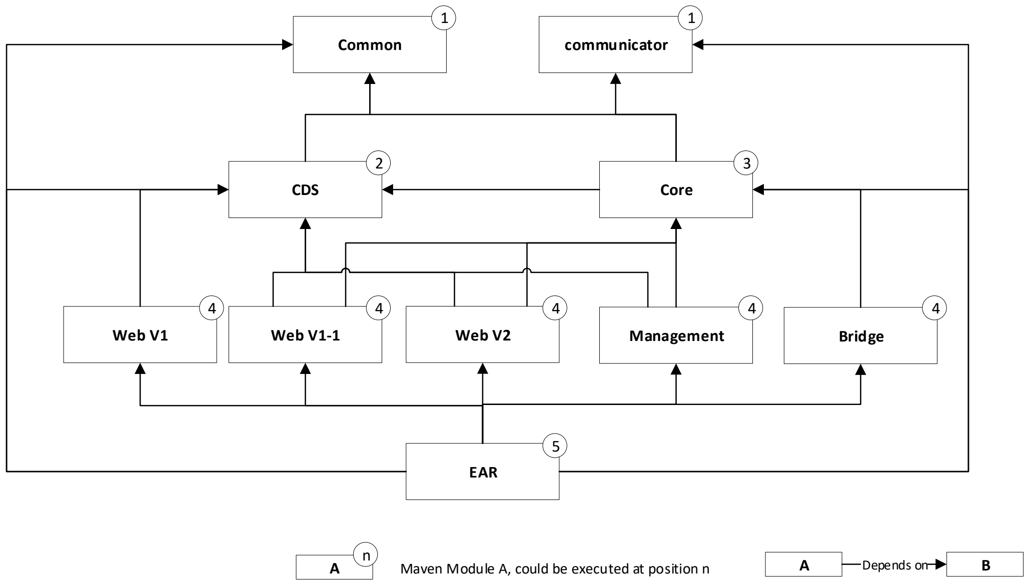


Figure 9.1.: IBE Web Service - Maven Module Dependency Tree

9.1.1. Context

We conducted the case study in an industrial context at TravelTainment [1]. Traveltainment, a company of the Amadeus IT Group SA with about 400 employees, develops software solutions for travel sales. One of their products is the Internet Booking Engine (IBE) which allows customers and travel agencies to search for and book travel offers. The functionality is provided by the *IBE Web Service*. It offers an XML-based API to search for and filter travel offers, to check their availability directly at the tour operator and to book offers. In peak times, the IBE Web Service handles about 20 millions requests per day. Internally, about 18 employees work on the IBE web service.

The IBE Web Service is written in Java EE and has dependencies to three external systems. A database persisting hotel information, a search engine to perform the search requests and keycloak for identity and access management. As of today, the project comprises of about 65 000 lines of code. The entire IBE Web Service project is built with maven by means of 10 maven modules (common, communicator, cds, core, bridge, management, web-v1, web-v1-1, web-v2 and ear). Since this module structure plays an important role in section 9.1.3, figure 9.1 provides an overview of their dependencies. It also indicated the potential execution index when parallelizing the build.

The IBE Web Service is bundled as an enterprise application archive (ear) to be deployed on an application server (JBoss Wildfly). The IBE Web Service project contains multiple unit tests and several junit tests, which are primarily used for functional integration testing. In the beginning of our case study, the jmeter testing was performed manually.

[1]http://www.traveltainment.de

The IBE Web Service team formulated certain requirements to automate this activity (by means of docker). Before evaluating our prototype, we therefore introduced the required changes. Section 9.1.1 details the adapted delivery process.

Delivery Process

The IBE Web Service delivery process consists of two phases. A build phase and a deployment phase. In the build phase, the software is built and tested (by means of junit and jmeter tests) and published in an artifact repository. In the deployment phase the published artifacts are deployed to production. The build phase is performed in a continuous manner. The deployment phase is performed on demand.

For internal reasons, we only could study the build phase. But since the build phase described above can be seen as its own delivery process (compile, test, deploy) and the thesis focuses on the build problem theme (cf. section 3.1), the chosen projects offers great potential for a case study.

Technically, the IBE Web Service team uses Hudson [Oraa], a web-based Continuous Integration (CI) System to carry out their delivery process. Hudson was discontinued and superseded by Jenkins in 2016. The IBE Web Service team defined several Hudson jobs for their delivery process phases. The build phase job which is triggered by commits to version control executes maven to build the project, to unit test them and to publish them to artifactory (Traveltainments global artifact repository). As stated, the junit test were performed manually. Thus, the Hudson job does not execute them. Following traveltainments requirements, we introduced a *docker build* and a *docker publish* activity to the delivery process, which build and run the IBE Web Service and its dependencies (database, keycloak). Using this provision mechanism, the jmeter tests can be automated.

9.1.2. Objective

Having defined the case study context in the previous section, this section describes our objectives. We summarize each objective and detail which data we collect to answer the objective.

Applicability of Core Domain

Descriptions & Rationale The prototype realizes central parts of our core domain (cf. section 5.1.1). Before exploring specific properties, we have to inspect if the overall concepts are applicable.

Required Data To provide evidence for the applicability of the core domain, we measure how much of the the IBE Web Service delivery process we can cover with the concepts of our core domain. Moreover, we count the amount of adoptions required to the core domain (in case we cannot cover the whole process).

Integratability of new technologies

Descriptions & Rationale One major challenge our delivery system tries to tackle is C1 - Project Evolution (cf. section 3.1). Central related requirement to this challenge is Req-1: Integration of new technology. Given the importance of this requirement, we want to explicitly explore if it is met.

Required Data To be able to qualitatively indicate if heterogeneous technologies can be integrated, we have to try their integration. Thereby, we need to count required changes on the delivery system to be able to give evidence about the integratability.

Impacts of Self-Organization

Descriptions & Rationale The second challenge our delivery system tries to tackle is C2 - Modeling Usability (cf. section 3.1). Beside the activity abstraction already assessed with our first objective (section 9.1.2), the key to tackle this challenge is meeting Req-4: Self-Organizing. Self-Organizing promises to ease the delivery process modeling and to automatically stick to best-practices. We want to explore the impacts of using a self-organized approach.

Required Data To explore the impacts of self-organizing, we need to model the IBE Web Service delivery process both with and without planners and compare them. Measures for comparing them are their lines of code, their complexity and also their transparency. Since there are two types of planners (cf. section 7.3.2), we also need to compare their resulting delivery processes.

9.1.3. Collected Data

We collected the following data during the case study:

Coverage of Delivery Process

Approach To determine the coverage of the IBE Web Service Delivery Process, we modeled the process using only concepts from our core domain. Figure 9.2 depicts the resulting delivery process. We thereby used the same notation as for our explanatory model in section 5.1.2. Overall, we were able to classify each activity either as a transformation or an assessment, thus we can cover the delivery process to 100% with our core domain.

Value 100% coverage

Related Objective Applicability of Core Domain

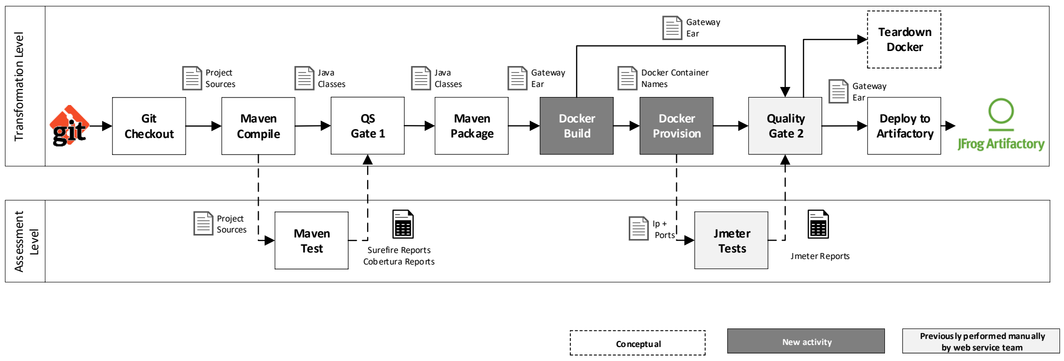


Figure 9.2.: Case Study - Delivery Process

Model-Induced domain changes

Approach Having defined the conceptual delivery process for the IBE Web Service Delivery Process, we used our pipeline description language to express this process in order to collect evidence for the applicability of our domain. Since this language is closely aligned to our domain, shortcomings in the language might signal domain problems. The appendix (Appendix A.1) provides the delivery model definition in our PDL. We thereby modeled each aspect manually without relying on self-organizing capabilities in order to explore the language expressiveness. Overall, we were able to define each aspect of the IBE Web Service delivery process without adapting the language or our core domain.

Value No changes required

Related Objective Applicability of Core Domain

Delivery System adoptions

Approach To gain evidence about the integratability of new technologies into the delivery system, we realized two activity services. A docker service providing functionality to build and provision docker containers and a jmeter service allowing to execute jmeter tests. Previous to the case study we also implemented a maven and a git service.

Value Developing and integrating a new activity service into the delivery system didn't require to adapt the delivery system or other activity service. Each integration was isolated thus offering high locality.

Related Objective Integratability of new technologies

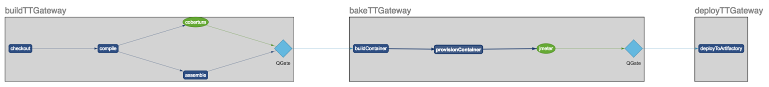


Figure 9.3.: Model-planned Delivery Process

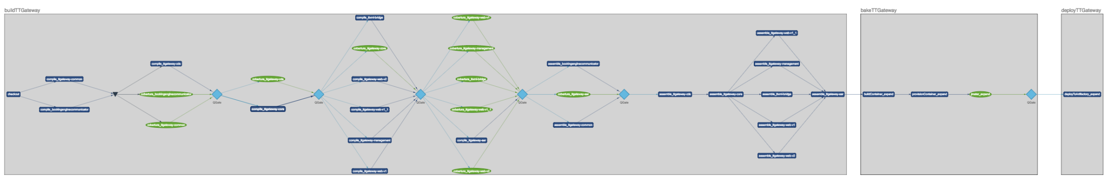


Figure 9.4.: Project-planned Delivery Process

Technological-Induced domain changes

Approach During the integration of new activity services (see above) we count if any changes are necessary to core domain to being able to accommodate the activity service.

Value No changes to the core domain were required to integrate new technologies

Related Objective Applicability of Core Domain

Equivalence of Delivery Processes

Approach To provide evidence for the impacts of self-organizing, we modeled the delivery process in three different ways: *manually*, i.e. without relying on self-organizing capabilities, *model-planned*, i.e. with relying on model planners to determine the dependencies between activities section 8.3.1 and *project-planned*, i.e. with both model-planners and project-planners enabled. Appendix A.1 provides the manual model definition, appendix A.2 the model-planned definition and appendix A.3 the project-planned definition. We then compared the resulting delivery processes of these models. Figure 9.3 and figure 9.4, respectively depict them to get an impression.

Value The delivery processes defined by the manual model and the model-planned model are equivalent, i.e. they comprises identical activities including same dependencies and configuration. The delivery processes of the model-planned and project-planned model are semantical equivalent, i.e. their resulting artifacts are identical, but the differ in the way or reaching their result. Since we realized a maven project-planner (cf. section 8.3.2) which knows about maven modules, the project-planned delivery process has more intermediate steps and quality gates. Altogether, the project-planned delivery process comprises 24 transformations, 11 assessments and 6 quality gates, while the model-planned delivery process has 6 transformations, 2 assessments and 2 quality gates.

Related Objective Impacts of Self-Organization

LOC of Delivery Models

Approach We compared the different delivery models (manual, model-planned and project-planned, see above) in terms of their lines of code to collect evidence on the impact of self-organizing on the delivery process modeling.

Value The manual model comprises 124 lines of code. The model-planned and project-planned models are semantically almost identical. The only difference is a flag enabling or disabling project-planning. They both have 66 lines of code. The overall savings correspond to 46,77 %. Listing 9.1 exemplify the reason for this enormous saving potential: No dependsOn nor related configuration properties must be provided.

```
name: compile
service: maven-service
activity: compile
# below is only mandatory for manual model #
dependsOn:
  - alias: repo
    ref: p://this/.../checkout/workspace
configuration:
  workspace: "@repo"
```

Source Code 9.1: Exemplary difference between manual and self-organizing

Related Objective Impacts of Self-Organization

Complexity of Delivery Models

Approach We also compared the different delivery models (manual, model-planned and project-planned, see above) in terms of their complexity. As a measure for complexity we use a similar approach as the McCabe Cyclomatic Complexity metric [LL10] defined for software components: We count all dependencies and cross-references (e.g. in policies) on other activities in the delivery model. Thus, we define the complexity of a model as:

$$c(x) = 1 + d(x) + r_{policies}(x)$$

where:

$$\begin{aligned} d(x) &= \text{amount of dependsOn declarations in x} \\ r(x) &= \text{amount of references in policy definitions in x} \end{aligned}$$

Execution of Delivery	Manual-defined / Model-Planned Delivery Process	Project-Planned Delivery Process
Run 1	298,466s	310,933s
Run 2	251,732s	295,606s
Run 3	306,027s	329,857s
Average	285,408s	312,132s

Table 9.1.: Manual vs Project-planned Delivery Process execution duration

Value Using our complexity measure, the manual delivery model has a complexity of 18. Both the model-planned and the project-planned model have a complexity of 1.

Related Objective Impacts of Self-Organization

Execution overhead of Self-Organization

Approach To further investigate the effects of self-organizing, we compared the execution behavior of the resulting delivery processes by measuring their execution durations. We couldn't compare the execution durations of our delivery system with the hudson setup, as the provided hardware differs from the hudson system. To indicate the execution overhead, we performed several runs of the delivery processes defined by the manual model and the delivery process defined by the project-planned model. Although our sample size isn't big enough to provide statistically relevant insights, at least they provide an indication. Since our prototype uses caching, the execution durations do not include the checkout duration required for project-planning. This duration nevertheless heavily depends on the project size and the bandwidth.

Value As already indicated (cf. section 9.1.3), the resulting delivery processes of the manual and the model-planned are equivalent. The model-planning itself didn't introduce a noticeable overhead (around 2ms). The execution durations of the delivery process expressed by the manual delivery model and the project-planned delivery model respectively are provided in table 9.1. The management overhead (copying of artifacts, tracking, etc.) of 31 additional activities outbalanced the improved parallelization degree. On average, the project-planned delivery process execution duration exceeded the manual defined delivery process by 9,14 %.

Related Objective Impacts of Self-Organization

Defect Seeding		Project-Planned			Normal			Speedup
		Run 1	Run 2	Avg	Run 1	Run 2	Avg	
Level 1	Common	36,6s	32,8s	35,6s	128,1s	110s	117,7s	**3,31**
	Communic.	37s	36,1s		121,7s	111s		
Level 3	Core	67,9s	69,4s	68,65s	113,7	111,2s	112,45s	**1,64**
Level 4	web-v1	97,5s	92,8s	95,7s	109s	112s	111,3s	**1,16**
	web-v1.1	96s	96,7s		110,9s	111,2s		
	web-v2	95,5s	95,5s		112s	112,5s		

Table 9.2.: Test Failure Seeding - Execution effects

Project-Planning Impacts

Approach Self-Organization promises to help to stick to best practices (cf. section 3.2). Having implemented a maven project planner for our prototype, we explored this feature for the *fail fast* best practice (cf. section 2.2.5). The maven planner adapts the delivery model based on the maven module structure (see figure 9.4 (page 126)). Thus, in principle defects should be detected earlier than in the manual delivery process. Similar to *Defect Seeding* [LL10] we applied Test Failure Seeding to be able to selectively control when the delivery process should fail. Thereby, we added failing tests (e.g. assertTrue(false)) in modules of the IBE Web Service that contain tests (common, communicator, core, web-v1, web-v1.1, web-v2) to explore an realistic effect. With the defects in place, we executed both the project-planned and the manually defined delivery process and measures their execution times. Both delivery processes use the exact same maven command as the Hudson build to demonstrate concrete saving potentials. Thus, there is no maven optimization in place (e.g. *skipAfterFailureCount* [Apab]).

Value Table 9.2 provides the execution durations until failure of the project-planned and the normal (manually defined) delivery process. Each level thereby corresponds to the execution index provided in figure 9.1 (page 122). If we add a defect in a module of level 1, the project-planned delivery process fails on average 3,31 times faster than the normal delivery process. For level 3 the speedup constitutes 1,64 and for level 4 we have at least 1,16. Of course these values only provide an indication, as a sample size of 2 does not provide any statistical substantiated evidence. Nevertheless the measurements indicate an enormous potential of project-based planning.

Related Objective Impacts of Self-Organization

9.1.4. Conclusion

We conducted a case study at traveltainment to explore if our proposed delivery system tackles related requirements of both our challenges C1 - Project Evolution and C2 - Modeling Usability. More concretely, our case study was driven by three objectives detailed in section 9.1.2: The applicability of the core domain to verify our concepts and understanding, the integratability of new technologies and to explore the impacts of Self-Organizing. These objectives are discussed in the the following:

Applicability of Core Domain The collected data indicates that the core domain is applicable. We were able to cover the complete IBE Web Service delivery process without making changes to the core domain. The core domain also remained stable when integrating different technologies. Thus, is not only is applicable but also supports evolution.

Integratability of new technologies Because of the microservice architecture, new technologies could easily be integrated by means of developing new activity services without affecting other services or the delivery system itself. No adaption were required to integrate a new activity service. This emphasizes the high degree of isolation and locality the architecture provides. Using the activity service framework (cf. section 8.4), the activity service realization can only focus on business logic.

Impacts of Self-Organization Major focus of the case study were the impacts of self-organization. The results indicate great potential: The model-based planning allowed to reduce the lines of code of the external delivery models by almost 47 %. These reduced models were also less complicated [2]. Basically following a *Convention over configuration* approach, a reduced model has the risk of reduced comprehensiveness as certain configuration and dependencies are applied automatically (planned). But the fact of being able to define the delivery model in different level of details (manually, or self-organized or anything in between) highlights the adaptability of our delivery system. The user can freely combine what fit his needs and skill level best.

The results also indicate great potential of project-based planning. Although the project-planned delivery process could not benefit from its increased parallelization degree as the execution in the positive case needed 9,4 % longer, the potential in terms of fail fast is promising: The project-planned delivery process provided up to 3,31 times faster feedback of occurred failures. Considering the great importance of providing fast feedback for delivery systems (cf. section 2.2.4), this factor indicates the great potential of project-planning. In addition, the project planning provided detailed insights into the delivery process.

[2]in terms of our complexity measure

9.2. Architectural Requirements

The following briefly discusses if the proposed architecture meets its requirements.

Req-1: Integration of new technology

Summary Key requirement for C1 - Project Evolution is to support easy integration of new technologies. The integration is easy when heterogeneous technologies can be added without requiring to introduce major changes to the delivery system.

Evaluation New technologies are integrated by means of activity services. Each activity service encapsulates functionality related to a certain technology. Following the service principles, each activity service has high autonomy. Each activity service is realized as a microservice, thus supporting arbitrary technologies and enforcing loose coupling. The case study supports this theory. Overall, the integration of a git service, maven service, jmeter service and a docker service could be easily realized. The only requirement for an activity service is to register its activity specifications at the activity specification registration and to poll the orchestrator for new tasks. The service registry thereby decouples activity services from a concrete registry and orchestrator instance. When introducing a new activity service, no existing components of the architecture needed to be adopted, which highlights the locality property of the architecture. In addition, the delivery system can even be extended during runtime. Summing up, the proposed architecture supports the easy integration of new technologies by means of providing an unified activity interface.

Req-2: Modularity

Summary Following the law of continuing change [Leh80], a software project will change and evolve. As the software evolves, its delivery process evolves too. The delivery system architecture should therefore be flexible. Important strategy to improve flexibility of a design is modularity [BCK12]. The architecture therefore should be modular

Evaluation The architecture is designed by means of the microservice architectural style, which by definition is modular. Since each microservice was scoped according to identified bounded contexts in Domain Driven Design, we consider them to be right-sized.

Req-3: Abstraction

Summary Related to C2 - Modeling Usability is the requirement to ease the delivery process by introducing abstractions. The delivery model should not contain every (technical) detail. The architecture should enforce information hiding and encapsulation by means of abstractions.

Evaluation The proposed architecture encapsulates the delivery process steps in activities. Each activity corresponds to a unit of work. These activities thereby have a defined interface hiding the technical details. Users only need to reference these activities (or more precisely their interface) in the delivery model without the need to know each and every aspect, which meets the requirement. Overall, the activities realize information hiding and assist reuseability and composition.

Req-4: Self-Organizing

Summary Central requirement in assisting the modeling (cf. C2 - Modeling Usability) is Self-Organization. The architecture should not require a fully-fledged delivery model as input. Instead, the architecture should infer information to handle incomplete models. This eases the modeling as the user does not have to provide each aspect. In addition to architecture should also be able to reorganize a delivery model.

Evaluation The proposed architecture enables planners to transform the given delivery model. Model-Planners rely on meta-information to gradually plan and complete the delivery model. Project-Planners use project specific information (source code) to adapt the defined delivery process. Overall and as indicated by our case study, this planning process allows to meet the Self-Organizing requirement. Our case study also provided evidence on the great potential of self-organization (see section 9.1.4). Section 9.3.2 discusses potential limitations of Self-Organization.

Req-5: Custom PDLs

Summary The architecture should support different pipeline description languages. This allows users to introduce models according to their needs and context improving the accessibility of the delivery model.

Evaluation The proposed architecture separates the concerns of modeling and execution by means of two different models (external and internal model). This property allows to introduce any type of external model as long as it can be converted to the internal model. Since this model to model transformation might loose information, a project-planner could always use the original external model to incorporate these information in his planning process, which mitigates the risks of primarily relying on the internal model. The event driven nature of the proposed architecture enables the view service to provide different visualizations for different model types. The view service thereby projects the execution domain events into the different external models. Overall, the architecture supports multiple model types both as input and as visualization target.

Req-6: Model Validation

Summary The second, major challenge is C2 - Modeling Usability. The architecture needs to validate the delivery model which both assists the user in modeling and improves the robustness of the architecture. The validation should comprise both of syntactic validation of the model itself, but also of semantic validation if the modeled activities exist and if their configuration matches their activity specification.

Evaluation The architecture employs a two phase validation process. First, the delivery model service validates the external delivery model syntactically before converting it into the internal delivery model. The process planner service uses its planners to validate the internal delivery model semantically. Model-based Planner validate modeled activities against their activity specification detecting compatibility and configuration issues. Project-based planner extends the semantic validation by project-specific concerns. They can for example validate, if an assessment activity can be performed, or if the project does not contain any tests at all. Overall, this provides a really powerful validation mechanism.

Req-7: Best practices

Summary The architecture should enforce Continuous Delivery best practices (cf. 2.2.5).

Evaluation The Self-Organization property of the architecture allows to adapt a delivery model. Assuming there is a planner for a concrete Continuous Delivery best practice, the self-organization process would ensure that this best practice is enforced. In our case study we realized a maven project planner that obeys the fail fast principle. Many best practices therefore can be reached by providing the corresponding planner. But similar to the discussion in section 9.3.1 the architecture information hiding imposes certain difficulties. When considering the build once principle (cf. section 2.2.5) for example, a planner could only assure that a single build activity is present in the delivery model. But he has no control over the internals of other activities. They could internally rebuild the artifact as they like. Since these problems hardly can be tackled on architecture level and require organizational measures, we still consider Req-7: Best practices to be met. In addition, the best practice requirement targets C2 - Modeling Usability, which is not about intentionally violating best practices.

Req-8: Traceability

Summary An important delivery system requirement is to provide visibility for every step in the delivery process as this improves the stakeholder collaboration (cf. section 2.2.3). Central for visibility in the delivery process context is the ability to backtrack, i.e. to trace which action produced which result. Given the great responsibility of a delivery system, traceability is also required to assist debugging, especially for issues regarding the deployment to production or issues that might occur when evolving the delivery model

Evaluation The proposed architecture relies on Event Sourcing to tackle the traceability requirement. Thereby, each business relevant incident is expressed as a domain event. The core services (more precisely their aggregates) use these events to determine state. This guarantees that every state change is published as an event. Ordering all events then allows to determine what happened. Another important aspect of the traceability requirement in the CD context is to have all binary results, i.e. the artifacts available. The architecture solves this by persisting all artifacts in the artifact store. Overall, the traceability requirement is met as long as all activity services stick to their classification. As discussed in section 9.3.1, activities that violate their classification (e.g. assessments that also perform a transformation) could produce results and artifact not recorded by the delivery system, which would violate the traceability requirement.

9.3. Discussion

Given the narrow focus of our case study presented in section 9.1, we made some observations not directly applicable to the case study objectives. As these observations still contribute to the evaluation, this section discusses them. In addition it discusses rationales to important design decisions.

9.3.1. Activity Classification Compliance

The implementation of activity services in the case study context showed us that the prototype doesn't enforce the activity classification. It's up to developers to strictly differentiate between transformations and assessments. In principle, the architecture allows to violate this classification by for example realizing an assessment that also performs an transformation (e.g. running docker containers). This introduces serious problems in terms of Req-8: Traceability. In addition, it violates the single responsibility principle, thus increasing maintenance effort and reducing reuseability. To tackle such risks, one differentiates between technical and organizational measures. The information hiding enforced by the architecture makes is difficult to realize technical measures against such smells as the activity realization is transparent to the delivery system. Instead, organizational measures could be easily applied to govern the activity classification compliances. Overall, this demonstrates both the flexibility of the architecture, as almost arbitrary activities can be realized, but also the potential trade-offs.

9.3.2. Limitations of Self-Organization

Although not investigated explicitly, the prototype and the case study indicate some limitations of Self-Organization. From a conceptual point of view, the planner requires knowledge to plan a delivery process. Either this knowledge is directly part of the external model, it can be derived from the context (activity specification) or it is part of a specialized planner. To deduce enough information from the context, the modeled delivery process activities must be very specific. While these activities are easy to validate, specialized activities are hardly reuseable. More generic activities make the validation harder and require a detailed external model or specialized planners. A detailed external model might conflict with the requirement to ease modeling. The last possibility of using specialized planners in combination with reuseable activities would restrict the delivery system to handling a concrete project as otherwise the amount of planner would explode. Overall, there is no concluding answer, instead every approach has a tradeoff and it heavily depends on the context what approach to choose.

Beside these conceptual tradeoffs, anectodal evidence collected during the case study suggests that the external model shouldn't be too minimalistic and abstract. Especially for production use, the external model should be as explicit as possible. So while theoretically self-organizing is only limited by *decideability*, practically the context decides on the self-organizing scope.

9.3.3. Delivery Model Transformations

Rumpe discusses that model transformation do not necessarily preserve semantics in all details [Rum17]. This might introduce problems as discussed in the following:

In the delivery system architecture, there are two types of model transformations: During the import of an external delivery model, there is an exogenous [MV06] (external-to-internal) delivery model transformation. During the planning, there is an endogenous [MV06] (internal-to-internal) delivery model transformation.

The import transformation typically will loose information considering the distilled character of our internal delivery model. From a planning perspective the information loss is without problems, as a specialized project planner still could use the external delivery model to incorporate missing aspects. From a visualization perspective though, it might introduce problems. Recalling chapter 6, the view service provides multiple views on a delivery process, typically one view per external model type. Since the execution domain events processed by the view service only occur for elements of the internal delivery model (delivery process and activities), the view service might be unable to provide a visualization matching the external delivery model.

For the planning transformation, different semantics might lead to nonequivalent planning results. To motivate planning equivalence related problems, we revisit our case study. Thereby, policies were defined globally (e.g. code coverage). The project-planning introduced additional quality gates and at the same time separated the project code modules. As a result, the policy was evaluated for each individual module, while in the model-planned process the policy was evaluated against the complete set of modules. Regarding code coverage this lead to a different behavior of the delivery system: For the individual policy evaluation the delivery process failed, while for the global evaluation the delivery process succeeded.

Considering the architectural scope of this thesis, we do not cover this aspect in further details. However, planning plays such an important role for tackling C2 - Modeling

Usability that future work should define a formal model to be able to reason about planning and related transformation.

9.3.4. CQRS and Event Sourcing

The core services rely on both CQRS and Event Sourcing. Since these decisions have a major impact on the architecture, the following discusses the rationales for using them.

The main rationale behind applying CQRS was to meet Req-5: Custom PDLs. Another possibility would be that the planner provides the visualization for different models. But since the visualization also includes execution aspects (e.g. execution status), this would violate the Single Responsible Principle [Mar02]. An obvious disadvantage of CQRS is the replication lag, i.e. because of the asynchronous nature it may take some time until changes are reflected in the query model. But since we do not require real-time information in the delivery process context the replication lag is acceptable. Another disadvantage of CQRS is the potentially increased complexity [Mar11] if there is some overlap between the command and query sides. In such cases, sharing a model might be easier. Since we designed the architecture from scratch by applying domain driven design, there only is a minimal overlap between the query and command sides. The core services separate the concerns of importing, planning and executing, while the view service integrates all events providing a query model for visualization. Overall, this separation leads to a minimal model overlapping such that the additional complexity concern of CQRS does not apply in our context. We even argue, that CQRS reduces complexity in our scenario as the core services do not need to deal with the visualization.

Central rationale behind applying event sourcing was to meet Req-8: Traceability. The traceability requirement demands to backtrack which activity produces which result. This can also be realized by using a correlation identifier (cf. [HW03]) for example. But having the self-organizing aspect in mind, it is foreseeable that the traceability requirement will be tightened. Moreover, and as section 10.2 hints, the event log also offers many possibilities for further improving the delivery system. Thus, we decided to use event sourcing for our core services. Central disadvantage of event sourcing is that all events need to be replayed to determine the current application state. Depending on the number of events this might introduce performance issues. To cope with this problem, we require our core services and the event store to support snapshotting mechanism such that only events newer than the snapshot need to be applied.

9.3.5. Orchestrator Design

The design decision to use an orchestrator for the execution management has a big impact on the overall architecture as it enforces a *hub and spoke* architecture (cf. [HW03]) for the delivery process execution. Thus, this section briefly discusses the rationales for this decision.

The potential disadvantages of a hub and spoke architecture are driven by the danger of a central processing unit, namely single point of failure and a performance bottleneck. To overcome these disadvantages a decentralized architecture is required which leads to a command *choreography* (cf. [New15]). An architecture pattern employing this choreography are distributed pipes and filters (see. [HW03]). Since this architecture has no central management unit, the process must be embedded into the application. This makes it difficult to change the process. In addition, it is a great challenge to answer what the progress of an execution is, as each underlying process might be different. A solution to overcome the embedded process problem is to use a *Routing Slip* ([HW03]), which basically means that the process information are attached to the message triggering the execution. Using this information each receiver can determine the next receiver without the need to hard-code the routing behavior. But because the process information is attached to the message which is passed around, only linear processes are supported. This conflicts with Req-7: Best practices: Parallel Workflow. Overall, an orchestrated approach has the disadvantages of a single point of failure and the choreographed approach embeds the process logic in the application making it difficult to change and to track progress. Given that visibility is an important goal of continuous delivery (cf. section 2.2.4) and our architecture should focus on flexibility and maintainability (cf. section 3.1.1), we decided to follow the orchestrated approach. For similar reasons, the netflix team decided the same (see [Net17]). Moreover, the internal design decisions (see above) allow to scale the orchestrator horizontally and thereby tackle the disadvantages of a centralized unit.

10. Conclusion & Future Work

Contents

10.1. Conclusion

This thesis presented an architecture, more precisely an application framework, for Self-Organizing Delivery Systems. Central goal of this architecture is to provide a foundation for Software Delivery Systems, that are flexible and maintainable both technically, but also from a delivery process modeling perspective. Based on related work in the literature and our experience, we identified these aspects as major challenges existing delivery systems face. More precisely, our first identified challenge is the evolveability of the delivery system itself to cope with changing requirements introduced by evolving software projects. The second challenge is related to the delivery process itself. Existing delivery systems do not separate the delivery model from the execution model, they directly execute the delivery process modeled by the user. This requires users to have deep technical as well as process-related knowledge.

From these challenges we deduced important requirements that need to be met in order to tackle those challenges. Our subsequent analysis of existing delivery systems indicated that there is an awareness for these kind of problems, but no one sufficiently tackles the challenges. Moreover, we believe that the struggle of existing solutions results from conceptual problems and the complexity of the problem domain which is inherent since delivery systems deal with large parts of the Software Development Life Cycle. The problems are even more intensified by ambiguous definitions. Therefore, we first defined consistent terminology and explored and defined important domain concepts which we distilled in our core domain. Continuing this Domain Driven Design approach, we define contexts and responsibilities and motivated microservices and messaging as our major design decisions. After that strategic design, we transitioned into tactical design refining the concepts and further motivating fine-grained design decisions like event sourcing and the activity encapsulation to tackle individual requirements.

Guided by our requirements, our central design decisions and with a clear domain understanding, we transferred the concepts into an architecture, i.e. the solution space.

Based on a top down approach, we gave an overview of the overall system, detailed dynamic aspects and then discussed important components. We presented a framework architecture, that isolates delivery process activities in dedicated activity microservices. Each activity encapsulates its related business logic and provides access via a simple to use unified interface. The microservice architectural thereby allows to integrate new technologies in an easy and robust manner.

Central aspect of the framework architecture is a three-staged process that controls the execution of a delivery process: First, the model service imports the delivery model, for example from version control. It translates the model into our internal delivery model, which we distilled as part of our core domain. Then different planners hierarchically validate, complement and optimize the delivery model. This not only reduces the amount of knowledge users need to have, it also allows to optimize a delivery model for a dedicated target. Additionally, it enables the delivery system to automatically stick to best practices and to realize organizational policies. After planning, the delivery system executes the planned delivery process by means of an orchestrator, controlling and monitoring the execution of corresponding activity services.

To validate our concepts and architecture, we developed a prototype based on the proposed design. We conducted a case study in an industrial context. Beside exploring the applicability of our concept, an important goal of the case study was to explore the impacts of Self-Organization, i.e. the ability of the delivery system to adapt and optimize a given delivery model. The case study indicated great potential of our approach. We could easily implement multiple, heterogeneous activity services that realize a complex delivery process, consisting of a build and unit test stage, a docker bake stage, a functional testing stage which temporarily deployed the built containers and performed junit tests and finally a deploy stage, that deployed the artifacts to artifactory.

Our prototypical planners thereby were able to optimize the delivery model in terms of fail fast, providing up to 3,31 times faster feedback than in the manual defined delivery process. Additionally, the planning capabilities nearly halved (47 %) the lines of code required to describe the delivery process in the delivery model. As anecdotal evidence, collected during the case study suggests, one needs to balance the model explicitness and its convenience.

Overall, we conclude that our architecture with its focus on isolation and explicitness of activities in combination with planning capabilities, provides both great flexibility and tremendously eases delivery process modeling, thereby reducing the modeling primarily on functional aspects and providing great potential for further process optimizations. On the downside, however, are the increased development effort for making activities explicit. But with dedicated activity service frameworks, like the one we developed for our prototype, the effort is manageable and provides long-term saving potentials. The inherent separation between activity development and delivery process modeling thereby allows to treat the delivery system as what it is, namely a first class citizen of the software development process.

10.2. Future Work

Considering the application framework nature of our architecture and its focus on flexibility, there are many great possibilities for future work:

Modeling Tools The activity specification registry and the amount of delivery process related information stored in the event store enable great possibilities for modeling tools:

- **Graphical Modeling Tool** Using the activity specification registry, a graphical modeling tool could be developed to further improve the modeling usability.
- **Editor with Autocompletion** The activity specification registry could also assist in developing DSL-based model editors with autocompletion support and live validation
- **Smart Tooling** The stored events could be analyzed to develop smart tooling, like recommenders

Smells and anti-pattern detection, policy-based validation So far, the delivery system architecture provides validation support based on the activity specifications. This allows for technical validation (e.g. parameter names correct) and functional validation (e.g. parameters in valid range, compatibility of activities). Thereby, the validation is performed only on activity-level. Future work could extend the validation to consider the overall delivery process. Delivery model smells and anti-pattern could be defined and detected. Policies could be used to validate if, for example, required assessments are present in the model.

Learning from Event Store During operation the delivery system produces many domain events. Resulting from the event sourced approach, each state change is captured in an event. This amount of data employs a whole range of possibilities for learning and making data-driven decisions. For example, planners can incorporate historical data for test selection or to prioritize their execution. Or usage-related data can be used to scale services pro-actively.

Cloud-Native The delivery system architecture consists of multiple microservices. Following our event-driven approach, they can be easily scaled since every service is stateless (beside the event store). Nevertheless, the architecture only is *cloud-compatible*. Future work could extend the architecture to support *cloud-native* principles like elasticity, i.e. the capability to dynamically start and stop services. Currently, this would not be possible since the delivery system capabilities (activity services) register themselves dynamically. If they are stopped, the delivery system does not know about them. In a similar veign, future work could explore the possibilities of serverless architecture.

Dynamic (Re-) Planning Central feature of the architecture is the Self-Organizing capability which manifests in planners adapting and optimizing the delivery model. Right now, planning and executing a delivery process is performed sequentially: The delivery model is first planned and the resulting delivery process then executed. This limits planning to static aspects as the delivery process cannot be dynamically adapted based on runtime events. For example, dynamic planning would allow to add additional assessments if the deployment takes longer than usual. Of course, one still has to keep the traceability requirement in mind. Taking Maven for example, they re-factored their life cycle planning in version 2.1 to be mapped out completely, before it is executed (cf. [Joh08]).

Supporting Continuous Experimentation A recent evolutionary step of Agile Software Development is called Continuous Software Engineering [Bos14]. It extends Continuous Delivery with Continuous Experimentation, which basically is the notion of continuously performing controlled experiments. The delivered product is instrumented and techniques like A/B testing then help to directly get feedback and learn. Future work could explore how to integrate Continuous Experimentation into the delivery system architecture. The first step could be the development of an experimentation planner, but central concern of continuous experimentation is to collect data, thus further mechanisms are required.

Activity & Delivery Process Templates During the case study we experienced the need to use specialized activities in order to have a high degree of Self-Organization. This required to implement activities that realize a facade to other activities. In order to reduce implementation effort, future work could extend the architecture to define such facade activities as a template. Process planners then map this template to a general purpose activity using the mapping defined in the template. Similarly, future work could also add support of delivery process templates in general or support for combining delivery models to further improve the usability.

Delivery Ecosystem Our delivery system is not agnostic to the projects it operates on. To be able to build and handle them, it requires activity services that provide functionality related to the project's technologies. To be able to handle many different heterogeneous projects, lots of different activity services are required. This introduces great effort as each project needs to implement customized activity services for general purpose functionality. To improve the acceptance and ease of use of the delivery system, future work could explore possibilities to provide a delivery ecosystem, where everybody can contribute general purpose activity services, which then can be tailored for specific use cases (e.g. via activity templates, see above).

Additional Delivery Models Our prototype so far supports models defined in our pipeline description language. A delivery process can be considered a business process. Thus, one could, for example, make use of the extensive support around the Business Process Modeling Notation (BPMN) to define delivery processes.

Multi-Target Planning The delivery system architecture supports multiple planners. Currently, they plan the delivery model sequentially, which makes it difficult to plan for different optimization targets. Basically, the last planner determines what the optimization target is, as it can overwrite the model adoptions of other planners. Future work could integrate other planner, e.g. for optimizing energy efficiency and explore how to reach consensus on different (maybe even conflicting) targets.

External Activities & Manual approval The delivery system architecture assumes that activities, respectively activity services are in control of performing their actions. Considering other domains, like IoT (Internet of Things), activity services might need to wait on external agents that might only be partially available. Such scenarios could be realized by specialized activity services, but might impose additional requirements like statefulness of activity services. Manual approval activities have similar challenges. Future work could explore possibilities to integrate such activities while still having stateless activity services that can be dynamically started & stopped (see cloud native above).

Formal Planning Framework The thesis focused on architectural concerns. Therefore, we did not provide formal foundations of the planning process. To be able to ensure certain properties like model equivalence or to analyze and compare different planners, future work could explore and define the formal foundations of the planning process.

Development Process Integration Important for the usage of our delivery system is its integration into the Software Development process. As stated, we designed a framework architecture. This framework must be tailored to the projects it delivers (e.g. regarding activity services). Since the evolution of a software project influences the delivery system, future work could analyze how to seamlessly integrate the delivery system tailoring into the software development process.

Delivery System SDKs Making activities explicit, has the downside of increased initial development effort, which amortizes itself quickly, considering the maintainability effort in the non-explicit case. Nevertheless, similar to our java-based activity service framework, future work could develop SDKs to allow for seamless development of activity services in other technologies.

All in all, this extensive list shows the great potential of our delivery system architecture. It provides lots of possibilities to be extended and improved to realize software delivery as best as possible.

A. Appendix

A.1. TT Web Service - Manual Model

```
stages:
  - name: buildTTGateway
    transformations:
      - checkout
      - compile
      - assemble
  - name: bakeTTGateway
    transformations:
      - buildContainer
      - provisionContainer
  - name: deployTTGateway
    transformations:
      - deployToArtifactory
transformations:
  - name: checkout
    service: git-service
    activity: checkout
    configuration:
      repositoryUri: https://git.../ttgateway.git
      branch: pipeline
  - name: compile
    service: maven-service
    activity: compile
    dependsOn:
      - alias: repo
        ref: p://this/transformations/checkout/workspace
    configuration:
      workspace: "@repo"
  - name: assemble
    service: maven-service
    activity: assemble
    configuration:
      workspace: "@repo"
      classes: "@compile"
    dependsOn:
      - alias: repo
        ref: p://this/transformations/checkout/workspace
      - alias: compile
        ref: p://this/transformations/compile/classes
  - name: buildContainer
    service: docker-service
    activity: buildTTGateway
    configuration:
      workspace: "@repo"
      javaPackage: "@assemble"
    dependsOn:
      - alias: repo
        ref: p://this/transformations/checkout/workspace
      - alias: assemble
        ref: p://this/transformations/assemble/assembly
  - name: provisionContainer
    service: docker-service
    activity: provisionTTGateway
    configuration:
      wildflyImageName: "@buildContainer/wildflyImage"
      databaseImageName: "@buildContainer/databaseImage"
      keycloakImageName: "@buildContainer/keycloakImage"
    dependsOn:
      - alias: buildContainer
        ref: p://this/transformations/buildContainer
  - name: deployToArtifactory
    service: maven-service
    activity: deploy
    configuration:
      workspace: "@repo"
      assemblies: "@assemble"
      classes: "@compile"
    dependsOn:
      - alias: repo
        ref: p://this/transformations/checkout/workspace
      - alias: assemble
        ref: p://this/transformations/assemble/assembly
      - ref: p://this/transformations/compile/classes
        alias: compile
      - ref: p://this/assessments/jmeter
assessments:
  - name: cobertura
    service: maven-service
    activity: cobertura
    configuration:
      workspace: "@repo"
      classes: "@compile"
    dependsOn:
      - alias: repo
        ref: p://this/transformations/checkout/workspace
      - alias: compile
        ref: p://this/transformations/compile/classes
  - name: jmeter
```

```
    service: jmeter-service
    activity: jmeterTT
    configuration:
      wildflyIp: "@provision/wildflyIp"
      wildflyPort: "@provision/wildflyPort"
      keycloakIp: "@provision/keycloakIp"
      keycloakPort: "@provision/keycloakPort"
      jmeterTestsRef: "@repo"
    dependsOn:
      - alias: provision
        ref: p://this/transformations/provisionContainer
      - alias: repo
        ref: p://this/transformations/checkout
qualityGates:
  - strategy: auto
    policies:
      - name: PassedTestRate
        interpretation: threshold-
        actualValue: passedRate
        setPoint: 1
        assessmentRef: p://this/assessments/cobertura
      - name: LineCoverage
        interpretation: threshold-
        actualValue: lineCoverage
        setPoint: 0
        assessmentRef: p://this/assessments/cobertura
      - name: AvgResponseTime
        interpretation: threshold+
        actualValue: avgResponseTimeMs
        setPoint: 400
        assessmentRef: p://this/assessments/jmeter
      - name: ResponseSuccessRate
        interpretation: threshold-
        actualValue: successRate
        setPoint: 0.26
        assessmentRef: p://this/assessments/jmeter
```

Source Code A.1: ManualModel.yml

A.2. TT Web Service - Model-Planner Model

```
planner:
  projectPlanning: false
stages:
  - name: buildTTGateway
    transformations:
      - checkout
      - compile
      - assemble
  - name: bakeTTGateway
    transformations:
      - buildContainer
      - provisionContainer
  - name: deployTTGateway
    transformations:
      - deployToArtifactory
transformations:
  - name: checkout
    service: git-service
    activity: checkout
    configuration:
      repositoryUri: https://git.../ttgateway.git
      branch: pipeline
  - name: compile
    service: maven-service
    activity: compile
  - name: assemble
    service: maven-service
    activity: assemble
  - name: buildContainer
    service: docker-service
    activity: buildTTGateway
  - name: provisionContainer
    service: docker-service
    activity: provisionTTGateway
  - name: deployToArtifactory
    service: maven-service
    activity: deploy
assessments:
  - name: cobertura
    service: maven-service
    activity: cobertura
  - name: jmeter
    service: jmeter-service
    activity: jmeterTT
qualityGates:
  - strategy: auto
    policies:
      - name: PassedTestRate
        interpretation: threshold-
        actualValue: passedRate
        setPoint: 1
      - name: LineCoverage
        interpretation: threshold-
        actualValue: lineCoverage
        setPoint: 0
      - name: AvgResponseTime
```

```
        interpretation: threshold+
        actualValue: avgResponseTimeMs
        setPoint: 400
      - name: ResponseSuccessRate
        interpretation: threshold-
        actualValue: successRate
        setPoint: 0.26
```

Source Code A.2: ModelPlannerModel.yml

A.3. TT Web Service - Project-Planner Model

```
planner:
  projectPlanning: true
stages:
  - name: buildTTGateway
    transformations:
      - checkout
      - compile
      - assemble
  - name: bakeTTGateway
    transformations:
      - buildContainer
      - provisionContainer
  - name: deployTTGateway
    transformations:
      - deployToArtifactory
transformations:
  - name: checkout
    service: git-service
    activity: checkout
    configuration:
      repositoryUri: https://git.../ttgateway.git
      branch: pipeline
  - name: compile
    service: maven-service
    activity: compile
  - name: assemble
    service: maven-service
    activity: assemble
  - name: buildContainer
    service: docker-service
    activity: buildTTGateway
  - name: provisionContainer
    service: docker-service
    activity: provisionTTGateway
  - name: deployToArtifactory
    service: maven-service
```

```yaml
    activity: deploy
assessments:
  - name: cobertura
    service: maven-service
    activity: cobertura
  - name: jmeter
    service: jmeter-service
    activity: jmeterTT
qualityGates:
  - strategy: auto
    policies:
      - name: PassedTestRate
        interpretation: threshold-
        actualValue: passedRate
        setPoint: 1
      - name: LineCoverage
        interpretation: threshold-
        actualValue: lineCoverage
        setPoint: 0
      - name: AvgResponseTime
        interpretation: threshold+
        actualValue: avgResponseTimeMs
        setPoint: 400
      - name: ResponseSuccessRate
        interpretation: threshold-
        actualValue: successRate
        setPoint: 0.26
```

Source Code A.3: ProjectPlannerModel.yml

Bibliography

[AM16] B. Adams and S. McIntosh. “Modern Release Engineering in a Nutshell: Why Researchers should Care”. In: *Proc. of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 2016, pp. 78–90. ISBN: 978-1-5090-1855-0. DOI: `10.1109/SANER.2016.108` (cited on page 12).

[And14] Andreas Grabner. *Software Quality Metrics for your Continuous Delivery Pipeline – Part I | Dynatrace blog.* 2014. URL: `https://www.dynatrace.com/blog/software-quality-metrics-for-your-continuous-delivery-pipeline-part-i/` (visited on 10/03/2017) (cited on page 11).

[Apaa] Apache Software Foundation. *Apache Kafka.* URL: `https://kafka.apache.org/contact%7B%5C#%7D` (visited on 12/17/2017) (cited on page 104).

[Apab] Apache Software Foundation. *Maven Surefire Plugin – Skipping Tests After Failure.* URL: `http://maven.apache.org/surefire/maven-surefire-plugin/examples/skip-after-failure.html` (visited on 12/23/2017) (cited on page 129).

[Bai09] D. Bailey. “S.O.L.I.D. Software Development, One Step at a Time”. In: *CODE Magazine, 2010 Jan/Feb* (2009). URL: `http://www.codemag.com/article/1001061` (cited on page 47).

[Bar+10] M. J. A. Barturen et al. *Integrated system and method for the management of a complete end-to-end software delivery process.* 2010. URL: `https://www.google.com/patents/US7735080` (cited on page 6).

[BC89] D. C. Brown and B. Chandrasekaran. *Design problem solving : knowledge structures and control strategies.* Pitman, 1989, p. 199. ISBN: 0934613079 (cited on page 84).

[BCK12] L. Bass, P. Clements, and R. Kazman. “Software Architecture in Practice”. In: *Vasa* 2nd (2012), pp. 1–426. ISSN: 03008495. DOI: `10.1024/0301-1526.32.1.54`. arXiv: `arXiv:1011.1669v3` (cited on pages 19, 131).

[Bec+01] K. Beck et al. *Agile Manifesto.* 2001. DOI: `10.1177/004057368303900411`. URL: `http://agilemanifesto.org/` (cited on page 1).

[Ber15] *ICSE '15: Proceedings of the 37th International Conference on Software Engineering - Volume 1.* Piscataway, NJ, USA: IEEE Press, 2015. ISBN: 978-1-4799-1934-5 (cited on page 1).

[Boe79] B. W. Boehm. “Guidelines for Verifying and Validating Software Requirements and Design Specifications”. In: *Euro IFIP 79* 1 (1979), pp. 711–719. ISSN: 0740-7459. DOI: 10.1109/MS.1984.233702 (cited on page 14).

[Bos14] J. Bosch. *Continuous software engineering*. Vol. 9783319112. 2014, pp. 1–226. ISBN: 9783319112831. DOI: 10.1007/978-3-319-11283-1 (cited on pages 1, 142).

[BWZ15] L. Bass, I. Weber, and L. Zhu. *DevOps: A Software Architect’s Perspective*. 1st. Addison-Wesley Professional, 2015. ISBN: 0134049845, 9780134049847 (cited on pages 6, 8–10, 12, 20).

[Cam] Cambridge Dictonary. *stage-gate Meaning in the Cambridge English Dictionary*. URL: http://dictionary.cambridge.org/dictionary/english/stage-gate (visited on 10/03/2017) (cited on page 11).

[CBA15] G. G. Claps, R. Berntsson Svensson, and A. Aurum. “On the journey to continuous deployment: Technical and social challenges along the way”. In: *Information and Software Technology*. Vol. 57. 1. 2015, pp. 21–31. ISBN: 09505849. DOI: 10.1016/j.infsof.2014.07.009 (cited on pages 1, 17).

[Che17] L. Chen. “Continuous Delivery: Overcoming adoption challenges”. In: *Journal of Systems and Software* 128 (June 2017), pp. 72–86. ISSN: 01641212. DOI: 10.1016/j.jss.2017.02.013. URL: http://linkinghub.elsevier.com/retrieve/pii/S0164121217300353 (cited on page 17).

[Clo] Cloudbees Inc. *Jenkins*. URL: https://jenkins-ci.org/ (visited on 12/27/2017) (cited on page 2).

[Doc] Docker Inc. *Docker - Build, Ship, and Run Any App, Anywhere*. URL: https://www.docker.com/ (visited on 12/17/2017) (cited on page 103).

[EBM16] Ed Bukoski, Brian Moyles, and Mike McGarr. *How We Build Code at Netflix – Netflix TechBlog – Medium*. 2016. URL: https://medium.com/netflix-techblog/how-we-build-code-at-netflix-c5d9bd727f15 (visited on 10/02/2017) (cited on page 11).

[Erl05] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. 2005, p. 760. ISBN: 0131858580. DOI: 10.1109/SAINTW.2003.1210138 (cited on pages 55, 73, 74).

[Eva03] E. Evans. “Domain-Driven Design: Tackling Complexity in the Heart of Software”. In: *Folia primatologica international journal of primatology* 70.5 (2003), p. 560. ISSN: 00155713. DOI: 10.1159/000067454 (cited on pages 35, 36, 43, 44, 46, 49, 51).

[Eve] Eventuate Inc. *Using Eventuate™*. URL: http://eventuate.io (visited on 12/17/2017) (cited on page 104).

[Faca] Facebook Inc. *Buck: A fast build tool*. URL: https://buckbuild.com/ (visited on 01/03/2018) (cited on page 1).

[Facb] Facebook Inc. *React - A JavaScript library for building user interfaces.* URL: `https://reactjs.org/` (visited on 12/17/2017) (cited on page 103).

[FFB13] D. G. Feitelson, E. Frachtenberg, and K. L. Beck. “Development and Deployment at Facebook”. In: *IEEE Internet Computing* 17.4 (July 2013), pp. 8–17. ISSN: 1089-7801. DOI: `10.1109/MIC.2013.25`. URL: `http://ieeexplore.ieee.org/document/6449236/` (cited on page 11).

[Fow02] M. Fowler. *Patterns of Enterprise Application Architecture.* Vol. 48. 2. 2002, p. 560. ISBN: 0321127420. DOI: `10.1119/1.1969597`. arXiv: `arXiv:1011.1669v3` (cited on pages 53, 54, 61, 62, 88).

[Fow05] M. Fowler. *Inversion of Control.* 2005. URL: `http://martinfowler.com/bliki/InversionOfControl.html` (cited on page 64).

[FS14] B. Fitzgerald and K.-J. Stol. “Continuous software engineering and beyond: trends and challenges”. In: *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering - RCoSE 2014.* New York, New York, USA: ACM Press, 2014, pp. 1–9. ISBN: 9781450328562. DOI: `10.1145/2593812.2593813`. URL: `http://dl.acm.org/citation.cfm?doid=2593812.2593813` (cited on page 7).

[Gam+02] E. Gamma et al. *Design Patterns – Elements of Reusable Object-Oriented Software.* 2002, p. 334. ISBN: 9780201715941. DOI: `10.1093/carcin/bgs084`. arXiv: `dd` (cited on pages 52, 65, 76, 88, 90, 92, 93).

[Get+04] V. Getov et al. *Performance Analysis and Grid Computing : Selected Articles from the Workshop on Performance Analysis and Distributed Computing August 19-23, 2002, Dagstuhl, Germany.* Springer US, 2004, p. 306. ISBN: 9781461503613 (cited on page 98).

[GIt] GItlab Inc. *Auto DevOps - GitLab Documentation.* URL: `https://docs.gitlab.com/ee/topics/autodevops/%7B%5C#%7Doverview` (visited on 12/26/2017) (cited on page 29).

[Gita] GitLab Inc. *GitLab Continuous Integration & Deployment | GitLab.* URL: `https://about.gitlab.com/features/gitlab-ci-cd/` (visited on 09/20/2017) (cited on page 28).

[Gitb] Gitlab Inc. *GitLab Architecture Overview - GitLab Documentation.* URL: `https://docs.gitlab.com/ce/development/architecture.html` (visited on 12/31/2017) (cited on page 30).

[Git17] Gitlab Inc. *GitLab Continuous Integration named a Leader in the Forrester Wave™ | GitLab.* 2017. URL: `https://about.gitlab.com/2017/09/27/gitlab-leader-continuous-integration-forrester-wave/` (visited on 12/26/2017) (cited on page 28).

[Goo] Google Inc. *Bazel - a fast, scalable, multi-language and extensible build system" - Bazel.* URL: `https://bazel.build/` (visited on 01/03/2018) (cited on page 1).

[Gor06] I. Gorton. *Essential software architecture.* 2006, pp. 1–283. ISBN: 3540287132. DOI: 10.1007/3-540-28714-0. arXiv: arXiv:1011.1669v3 (cited on page 63).

[Her15] J. Hermanns. "Artifact Promotion with Deployment Pipelines in the Context of Continuous Delivery". Bachelor Thesis. RWTH Aachen University, 2015 (cited on pages 10, 14, 15).

[HF10] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* 2010, p. 497. ISBN: 978-0-321-60191-9. DOI: 10.1007/s13398-014-0173-7.2. arXiv: arXiv:1011.1669v3 (cited on pages 6–10, 12, 14, 37).

[HG03] F. Heylighen and C. Gershenson. *The Meaning of Self-organization in Computing.* 2003. DOI: 10.1109/MIS.2003.1217631 (cited on page 81).

[HMM88] P. Harmon, R. Maus, and W. Morrissey. *Expert systems : tools and applications.* Wiley, 1988, p. 289. ISBN: 0471839507. URL: https://dl.acm.org/citation.cfm?id=576204 (cited on page 84).

[HP17] Hendrik Brinkmann and Philip Stroh. *Jenkins 2 in der Praxis: So entwickelt man Pipelines.* 2017. URL: https://jaxenter.de/pipeline-jenkins-2-61568 (visited on 12/31/2017) (cited on pages vii, 2).

[HW03] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* 2003, p. 736. ISBN: 0321200683. DOI: 10.1525/vs.2009.4.3.toc. arXiv: 1011.1669v3 (cited on pages 45, 48, 62, 65, 66, 85, 87, 92, 94, 136, 137).

[IEE02] IEEE. *IEEE Standard 610.12-1990 Glossary of Software Engineering Terminology (Reaffirmed 2002).* Vol. 121990. 1. 2002, p. 1. ISBN: 155937067X. DOI: 10.1109/IEEESTD.1990.101064. URL: http://ieeexplore.ieee.org/xpls/abs%7B%5C_%7Dall.jsp?arnumber=159342 (cited on pages 1, 37, 121).

[ISO15] ISO 25000. *ISO 25010.* 2015. URL: http://iso25000.com/index.php/normas-iso-25000/iso-25010 (cited on pages 18, 19).

[Jam13] James Hugh. *Micro Service Architecture.* 2013. URL: https://yobriefca.se/blog/2013/04/29/micro-service-architecture/ (visited on 11/29/2017) (cited on page 48).

[Jin11] Jinesh Varia. "Architecting for the Cloud: Best Practices". In: (2011). URL: https://media.amazonwebservices.com/AWS%7B%5C_%7DCloud%7B%5C_%7DBest%7B%5C_%7DPractices.pdf (cited on page 71).

[Joh08] John Casey. *Deterministic Lifecycle Planning - Maven - Apache Software Foundation.* 2008. URL: https://cwiki.apache.org/confluence/display/MAVENOLD/Deterministic+Lifecycle+Planning (visited on 12/27/2017) (cited on page 142).

[Ken15] Ken Mugrage. *5 Key Deployment Pipeline Patterns | GoCD Blog.* 2015. URL: `https://www.gocd.org/2015/08/28/pipeline-patterns/` (visited on 08/10/2017) (cited on page 10).

[Kim17] S. Kim. *Spinnaker: continuous delivery from first principles to production (Google Cloud Next '17).* 2017 (cited on page 10).

[Kru95] P. Kruchten. "The 4+ 1 view model of architecture". In: *Software, IEEE* November 1.November (1995), p. 9. ISSN: 0740-7459. DOI: `10.1109/52.469759`. URL: `http://ieeexplore.ieee.org/xpls/abs%7B%5C_%7Dall.jsp?arnumber=469759` (cited on page 59).

[Leh+15] T. Lehtonen et al. "Defining metrics for continuous delivery and deployment pipeline". In: *CEUR Workshop Proceedings.* Vol. 1525. 2015, pp. 16–30 (cited on page 12).

[Leh80] M. M. Lehman. "Programs, Life Cycles, and Laws of Software Evolution". In: *Proceedings of the IEEE* 68.9 (1980), pp. 1060–1076. ISSN: 15582256. DOI: `10.1109/PROC.1980.11805` (cited on pages 17, 131).

[Lev14] Lev Gorodinski. *Sub-domains and bounded contexts in Domain-Driven Design (DDD) - Lev Gorodinski.* 2014. URL: `http://gorodinski.com/blog/2013/04/29/sub-domains-and-bounded-contexts-in-domain-driven-design-ddd/` (visited on 10/10/2017) (cited on page 45).

[LIL17] E. Laukkanen, J. Itkonen, and C. Lassenius. "Problems, causes and solutions when adopting continuous delivery—A systematic literature review". In: *Information and Software Technology* 82 (Feb. 2017), pp. 55–79. ISSN: 09505849. DOI: `10.1016/j.infsof.2016.10.001`. URL: `http://linkinghub.elsevier.com/retrieve/pii/S0950584916302324` (cited on pages 1, 17).

[LL10] J. Ludewig and H. Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken.* dpunkt-Verlag, 2010. ISBN: 9783898646628. URL: `http://books.google.de/books?id=Wa3qQgAACAAJ` (cited on pages 6, 37, 38, 55, 76, 91, 127, 129).

[MA89] B. Meyer and P. America. "Object-oriented software construction 2nd Ed." In: *Science of Computer Programming* 12.1 (1989), pp. 88–90. ISSN: 01676423. DOI: `10.1016/0167-6423(89)90034-8`. arXiv: `arXiv:1011.1669v3` (cited on pages 54, 88).

[Mar] Martin Fowler. *DDD_Aggregate.* URL: `https://martinfowler.com/bliki/DDD%7B%5C_%7DAggregate.html` (visited on 12/06/2017) (cited on page 49).

[Mar02] R. C. Martin. *Agile Software Development, Principles, Patterns, and Practices.* 2002. DOI: `10.1007/BF03250842` (cited on page 136).

[Mar10] Martin Fowler. *BlueGreenDeployment*. 2010. URL: https://martinfowler.com/bliki/BlueGreenDeployment.html (visited on 09/26/2017) (cited on page 10).

[Mar11] Martin Fowler. *CQRS*. 2011. URL: https://martinfowler.com/bliki/CQRS.html (visited on 11/20/2017) (cited on page 136).

[Mar12] Martin Fowler. *SnowflakeServer*. 2012. URL: https://martinfowler.com/bliki/SnowflakeServer.html (visited on 10/04/2017) (cited on pages 1, 9).

[Mar13a] Mark Chang. *Model everything to fail fast | ThoughtWorks*. 2013. URL: https://www.thoughtworks.com/de/insights/blog/model-everything-fail-fast (visited on 12/23/2017) (cited on page 10).

[Mar13b] Martin Fowler. *ContinuousDelivery*. 2013. URL: https://martinfowler.com/bliki/ContinuousDelivery.html (visited on 08/08/2017) (cited on page 1).

[Mar13c] Martin Fowler. *DeploymentPipeline*. 2013. URL: https://martinfowler.com/bliki/DeploymentPipeline.html (visited on 10/03/2017) (cited on page 8).

[Mar16] Martin Fowler. *InfrastructureAsCode*. 2016. URL: https://martinfowler.com/bliki/InfrastructureAsCode.html (visited on 01/03/2018) (cited on page 2).

[McI+11] S. McIntosh et al. "An empirical study of build maintenance effort". In: *Proceeding of the 33rd international conference on Software engineering - ICSE '11*. 2011, p. 141. ISBN: 9781450304450. DOI: 10.1145/1985793.1985813. URL: http://portal.acm.org/citation.cfm?doid=1985793.1985813 (cited on page 18).

[MJ14] Martin Fowler and James Lewis. *Microservices*. 2014. URL: https://martinfowler.com/articles/microservices.html (visited on 10/09/2017) (cited on page 47).

[MV06] T. Mens and P. Van Gorp. "A Taxonomy of Model Transformation". In: *Electronic Notes in Theoretical Computer Science* 152 (Mar. 2006), pp. 125–142. ISSN: 1571-0661. DOI: 10.1016/J.ENTCS.2005.10.021. URL: https://www.sciencedirect.com/science/article/pii/S1571066106001435 (cited on page 135).

[Neta] Netflix Inc. *Architecture - Spinnaker*. URL: https://www.spinnaker.io/reference/architecture/ (visited on 12/25/2017) (cited on pages 1, 21).

[Netb] Netflix Inc. *Introduction - Conductor*. URL: https://netflix.github.io/conductor/ (visited on 12/17/2017) (cited on page 103).

[Netc] Netflix Inc. *Netflix Open Source Software Center*. URL: https://netflix.github.io/ (visited on 12/17/2017) (cited on page 103).

[Netd] Netflix Inc. *Spinnaker.* URL: `https://www.spinnaker.io/` (visited on 12/25/2017) (cited on page 21).

[Net12] Netflix Inc. *Asgard: Web-based Cloud Management and Deployment – Netflix TechBlog – Medium.* 2012. URL: `https://medium.com/netflix-techblog/asgard-web-based-cloud-management-and-deployment-2c9fc4e4d3a1` (visited on 12/25/2017) (cited on page 21).

[Net15] Netflix Inc. *Global Continuous Delivery with Spinnaker – Netflix TechBlog – Medium.* 2015. URL: `https://medium.com/netflix-techblog/global-continuous-delivery-with-spinnaker-2a6896c23ba7` (visited on 12/25/2017) (cited on pages 21, 22).

[Net17] Netflix Inc. *Introduction - Conductor.* 2017. URL: `https://netflix.github.io/conductor/` (visited on 11/26/2017) (cited on page 137).

[New15] S. Newman. *Building Microservices.* 2015, p. 280. ISBN: 978-1-491-95035-7. DOI: `10.1109/MS.2016.64`. arXiv: `1606.04036` (cited on pages 62, 64, 71, 87, 95, 137).

[OAB12] H. H. Olsson, H. Alahyari, and J. Bosch. "Climbing the "Stairway to Heaven" – A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software". In: *2012 38th Euromicro Conference on Software Engineering and Advanced Applications.* IEEE, Sept. 2012, pp. 392–399. ISBN: 978-0-7695-4790-9. DOI: `10.1109/SEAA.2012.54`. URL: `http://ieeexplore.ieee.org/document/6328180/` (cited on page 42).

[Obj17] Object Management Group. *OMG IDL.* 2017. URL: `http://www.omg.org/gettingstarted/omg%7B%5C_%7Didl.htm` (visited on 11/24/2017) (cited on pages 76, 94).

[Oraa] Oracle Inc. *Hudson Continuous Integration.* URL: `http://hudson-ci.org/` (visited on 12/20/2017) (cited on page 123).

[Orab] Oracle Inc. *MySQL.* URL: `https://www.mysql.com/` (visited on 12/17/2017) (cited on page 104).

[Per15] Perforce Software Inc. "Continuous Delivery: The New Normal for Software Development". In: (2015). URL: `http://www.perforce.com/continuous-delivery-report` (cited on page 1).

[Piva] Pivotal Inc. *Concourse Documentation.* URL: `https://concourse.ci/single-page.html` (visited on 12/26/2017) (cited on pages 24–26).

[Pivb] Pivotal Inc. *Spring Boot.* URL: `https://projects.spring.io/spring-boot/` (visited on 12/17/2017) (cited on page 102).

[Pivc] Pivotal Software Inc. *Concepts.* URL: `https://concourse.ci/concepts.html` (visited on 10/03/2017) (cited on page 11).

[Pivd] Pivotal Software Inc. *Concourse: CI that scales with your project.* URL: `https://concourse.ci/` (visited on 12/26/2017) (cited on pages 1, 24).

[RH09] P. Runeson and M. Höst. "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical Software Engineering* 14.2 (2009), pp. 131–164. ISSN: 13823256. DOI: 10.1007/s10664-008-9102-8. arXiv: 9809069v1 [arXiv:gr-qc] (cited on page 121).

[Rob17] Rob Zienert. *Codifying your Spinnaker Pipelines – The Spinnaker Community Blog.* 2017. URL: https://blog.spinnaker.io/codifying-your-spinnaker-pipelines-ea8e9164998f (visited on 12/25/2017) (cited on page 22).

[Rod+12] P. Rodríguez et al. "Survey on agile and lean usage in finnish software industry". In: *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM '12*. New York, New York, USA: ACM Press, 2012, p. 139. ISBN: 9781450310567. DOI: 10.1145/2372251.2372275. URL: http://dl.acm.org/citation.cfm?doid=2372251.2372275 (cited on page 1).

[Rod+17] P. Rodriguez et al. "Continuous deployment of software intensive products and services: A systematic mapping study". In: *Journal of Systems and Software* 123 (2017), pp. 263–291. ISSN: 01641212. DOI: 10.1016/j.jss.2015.12.015 (cited on pages 7, 12, 17).

[Rum17] B. Rumpe. *Agile modeling with UML: Code generation, testing, refactoring.* 2017, pp. 1–388. ISBN: 9783319588629. DOI: 10.1007/978-3-319-58862-9 (cited on page 135).

[Sho04] J. Shore. "Fail fast". In: *IEEE Software* 21.5 (2004), pp. 21–25. ISSN: 07407459. DOI: 10.1109/MS.2004.1331296 (cited on page 10).

[Som11] I. Sommerville. *Software Engineering.* 2011, p. 773. ISBN: 9780137035151. DOI: 10.1111/j.1365-2362.2005.01463.x. arXiv: 0321313798 (cited on page 5).

[Spi] Spinnaker. *Concepts - Spinnaker.* URL: https://www.spinnaker.io/concepts/ (visited on 10/03/2017) (cited on page 11).

[Tho] Thoughworks Inc. *Introduction · GoCD User Documentation.* URL: https://docs.gocd.org/current/ (visited on 09/20/2017) (cited on page 11).

[Tho17] Thoughworks Inc. *A single CI instance for all teams | Technology Radar | ThoughtWorks.* 2017. URL: https://www.thoughtworks.com/de/radar/techniques/a-single-ci-instance-for-all-teams (visited on 10/02/2017) (cited on page 1).

[Vak+15] M. Vakilian et al. "Automated Decomposition of Build Targets". In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 123–133. ISBN: 978-1-4799-1934-5. URL: http://dl.acm.org/citation.cfm?id=2818754.2818772 (cited on page 82).

[Ver12] V. Vernon. *Implementing domain-driven design.* Addison-Wesley Professional, 2012. ISBN: 0321834577 (cited on pages 41, 45, 54).

[Ver16] V. Vernon. *Domain-driven design distilled.* Addison-Wesley, 2016. ISBN: 0134434420 (cited on pages 35, 45, 48, 49, 52, 53, 60, 71).

[Via16] M. Vianden. "Systematic Metric Systems Engineering: Reference Architecture and Process Model". PhD thesis. 2016 (cited on pages 37, 74).