© Copyright 2018 ACM
This is a draft version of the paper, for private use only!

Designing a Next-Generation Continuous Software Delivery System: Concepts and Architecture

Andreas Steffens
RWTH Aachen University
Aachen, Germany
steffens@swc.rwth-aachen.de

Horst Lichter
RWTH Aachen University
Aachen, Germany
lichter@swc.rwth-aachen.de

Jan Simon Döring
RWTH Aachen University
Aachen, Germany
jan.simon.doering@rwth-aachen.de

ABSTRACT

Continuous Integration and Continuous Delivery are established practices in modern agile software development. The DevOps movement adapted theses practices and places the deployment pipeline at its heart as one of the main requirements to automate the software development process and to deliver and operate software in a more robust way with higher quality.

Over the time a lot of systems and tools has been developed to implement the deployment pipeline and to support continuous delivery. But software development is complex, its process even more and due to the individual organization of software vendors no real all-in-one solution for CD exists. Literature identified a lot of challenges when adopting CD and DevOps in an organization.

This paper presents a conceptual model and fundamental design decisions for a new generation of software delivery systems tackling some of these issues. Our approach focuses on two specific challenges for adopting CD. The first is the lack of flexibility and maintainability of software delivery systems. The second is the insufficient user support to model and manage delivery processes and pipelines. We introduce an automated mechanism to ease the effort for developers and other stakeholders.

Based on these results this paper introduces an architectural proposal for a next-generation continuous software delivery system.

KEYWORDS

Continuous Delivery, Continuous Software Engineering, DevOps, Architecture, Microservices, Framework, Domain Modelling

1 INTRODUCTION

An important activity in the Software Development Life Cycle is the delivery of the developed software [12]. Depending on the software development approach used, it can be performed iteratively. Agile process models like Scrum[25] offer such an iterative approach to gain short customer feedback cycles and to quickly respond to changing requirements. In addition, the agile manifesto states that working software should be delivered frequently[3]. Therefore, with the increasing adoption of agile development practices like Continuous Integration and Continuous Delivery became more and more mainstream [21, 23].

However, adopting Continuous Delivery is challenging. One major challenge is the tooling support for continuous delivery [5, 15] indicated by emerging new delivery systems. Companies like Pivotal, Netflix, Facebook and Google build specific systems and tool chains tailored for their needs.

Delivery systems need to integrate lots of heterogenous tools and technologies and need to cope with the evolution of the software project. The ThoughtWork Technology Radar [26] highlights the immaturity 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.

For Users it is hard to maintain and reuse their delivery processes definitions in these systems, which leads to SnowflakeServers.[17]. The second generation of delivery systems tackle these problems by following the *infrastructure as code* [18] principle and keep all delivery process models and related information in a single, version-controlled file. Thereby calling this approach *pipeline as code.* The stakeholders of the delivery system are forced to incorporate deep technical tool knowledge and to have profound process-related knowledge to be able to execute a correct and complete software delivery process.

Therefore, the presented concept, design and architecture for a new generation of software delivery systems try to tackle the aforementioned issues.

The paper is structured as follows. The next section provides the necessary background followed by a analysis of the existing challenges. Based on this, we introduce a consistent new domain model for continuous software delivery. The following design decisions and the resulting architecture rely heavily on this presented delivery model. Before concluding this paper we present related work how other modern delivery systems tackle these challenges.

2 BACKGROUND: CONTINUOUS DELIVERY

Humble and Farley coined the term *Continuous Delivery* with their book [10] and define it as

> Continuous Delivery is a set of practices that aims to deliver value to customers rapidly, reliably and repeatedly with minimal manual overhead.

The Deployment Pipeline is a central part of Continuous Delivery. Humble and Farley define the term as *an automated manifestation of your process for getting software from version control into the hands of your users.*, i.e. the deployment pipeline is a software system that automates the software delivery process. At other places, Humble and Farley describe the deployment pipeline as a model (*the process*

4th International Workshop on Rapid Continuous Software Engineering (RCoSE '18). ACM, New York, NY, USA, 1-7
DOI: https://doi.org/10.1145/3194760.3194768

© Copyright 2018 ACM
This is a draft version of the paper, for private use only!

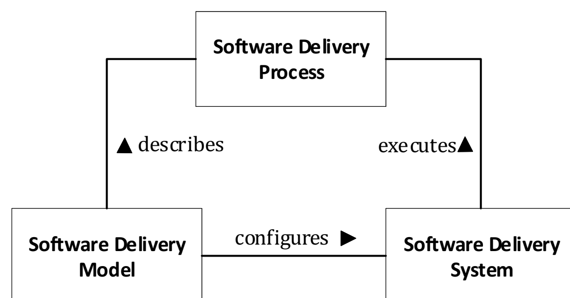

Figure 1: Relations between Software Delivery Process, Model and System

modeled by the deployment pipeline). Bass et al.[2] identify similar dimensions: *First, the DevOps pipeline itself is a piece of software [...]. Second, the DevOps pipeline has characteristics of a process.*.

We think that these different dimensions lead to ambiguity and thus complicate understanding when using the deployment pipeline term as is. Based on the ISO 9001 standard [11] we define a new and consistent terminology.

- A deployment pipeline models a delivery process. Here, we use **Software Delivery Model**, abbreviated as Delivery Model, instead of deployment pipeline.
- A deployment pipeline is an integrated software system. Here, we use **Software Delivery System**, abbreviated Delivery System, instead of deployment pipeline.
- A deployment pipeline itself has characteristics of a process. Here, we use **Software Delivery Process**, abbreviated Delivery Process, instead of deployment pipeline.

Figure 1 shows the relation between the different terms. During this paper we use Software Delivery and Delivery interchangeable. Since literature uses the term pipeline, we sometimes may need to fall-back to this term too.

3 CHALLENGES

Continuous Delivery (CD) offers a lot of benefits. But adopting these practices is difficult [5]. Laukkanen et al. identified six recurring problem themes in their systematic literature review [15]. 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 [4]. In addition, a systematic mapping study by Rodriguez et al. found, that only 7% of their analyzed publications contribute to this Delivery System area [24].

From these insights we can derive two major challenges for designing delivery systems.

A delivery system has to cope with evolution. Following the Law of Continuing Change [16], a software will change and evolve. As the delivery system realizes the project's delivery process it needs to reflect and adapt to these changes. The project evolution comprises multiple dimensions.

The first one is related to the project's software architecture. Each architectural decision imposes new functional requirements on the software delivery system. New components and technologies might need be integrated into the software delivery system.

Second, the delivery process might evolve, e.g. a new test stage has to be integrated. At last, from an organizational perspective, new non-functional requirements or policies might emerge that affect the delivery process but also the delivery system itself, i.e., the project is obligated to comply to new laws and regulations Overall, the evolution challenge requires delivery systems to be flexible and maintainable.

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* [13]. The identified usability challenge relates to the modeling context. Generalizing Laukkanen's finding, the delivery model is complex and difficult. Currently, users need to possess lots of technical details which complicates the construction of a good delivery model. The delivery system should minimize the amount of technical knowledge required to define a delivery model and use a delivery system.

In addition, the delivery system should also minimize the amount of process knowledge required to define a delivery model. Users should not be concerned with parallelization respectively defining an explicit order. Delivery systems should assist users in defining the delivery model as much as possible. They are required to offer validation support and mechanisms like auto-completion or recommendation for delivery models.

McIntosh et al. [20] suggests that build maintenance account for 27% of code development and 44% of test development effort. Therefore, increasing the usability of delivery systems is essential. Our paper provides two contributions to tackle these two challenges. First we offer a conceptual domain model for software delivery processes and systems, which defines key abstractions and concepts regarding the modeling of delivery processes and the design of a delivery system. Due to the separation of the delivery model and the delivery system we achieve the ability to tackle the aforementioned problems on separate levels. This conceptual model can be used as a meta-model to define and describe delivery processes. Through model to model transformations existing languages can be integrated into our approach. Furthermore the model includes concepts for the automatic processing and generation of delivery models without user interaction. The second contribution is a highly flexible microservice-based software architecture, which, based on our conceptual domain model and context map, separates the different concerns and responsibilities of the software delivery domain. It provides a clear focus on the most important aspects and components to tackle the aforementioned challenge of flexibility and maintainability.

4 SOFTWARE DELIVERY DOMAIN MODEL

Following the principles of Domain Driven Design [7] this section introduces the central concepts of our approach for a next-generation software delivery system in form of a domain model.

4th International Workshop on Rapid Continuous Software Engineering (RCoSE '18). ACM, New York, NY, USA, 1-7
DOI: https://doi.org/10.1145/3194760.3194768

© Copyright 2018 ACM
This is a draft version of the paper, for private use only!

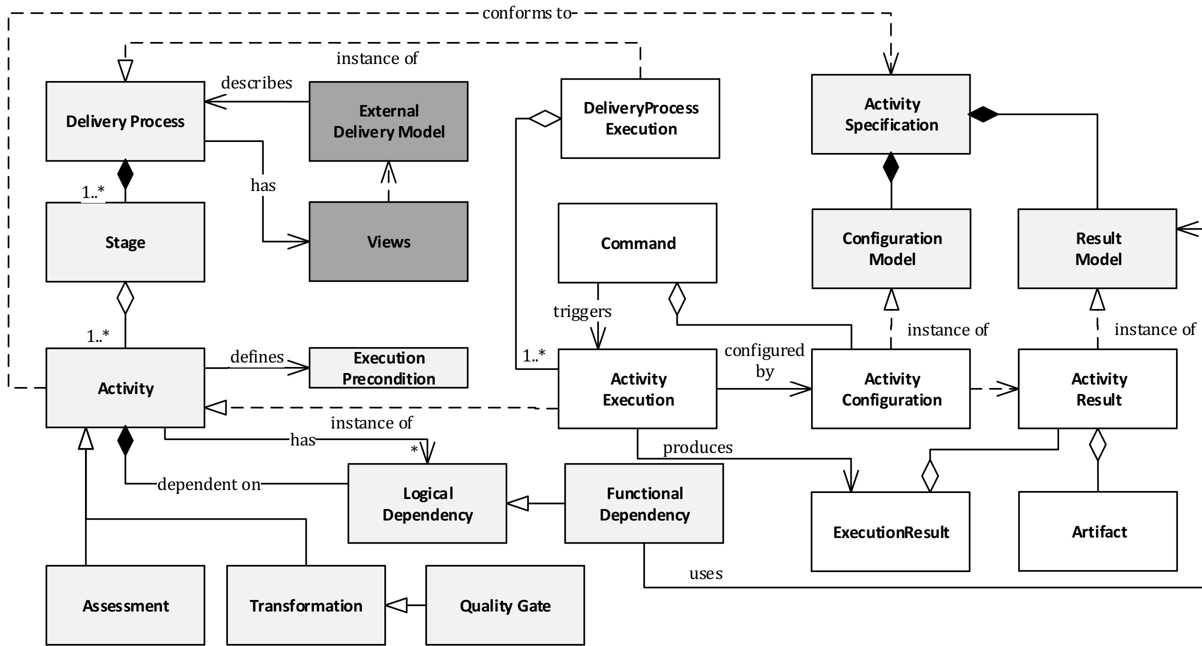

Figure 2: Delivery System Core Domain

Figure 2 depicts the complete domain model. We modeled only a few cardinalities as we focus on the general concept relations in this section. In addition, we only gonna present the most important concepts which leads to the proposed architecture.

A delivery process comprises of various activities to be performed during its execution, e.g., compilation of source code, running unit tests or deploying the built software artifacts to a repository or even a production environment. These activities can be grouped into stages which forms the delivery process. We introduce a classification on these activities to distill central concepts for our approach. Activities encapsulate delivery process behavior. So they either *transform*, *assess* or *promote* artifacts. Consequently, each activity can be assigned to one of the following classes: Transformation, Assessment and Quality Gate. Figure 3 depicts these classes and their purpose inside a delivery process.

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 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. A delivery process must at least comprise one transformation.

Assessments perform measurements to be able to evaluate certain properties of the input artifact. They publish these results as a report. The assessment realization decides which measurements are performed. 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 and produces a single report for this artifact. The report might contain multiple measurement results.

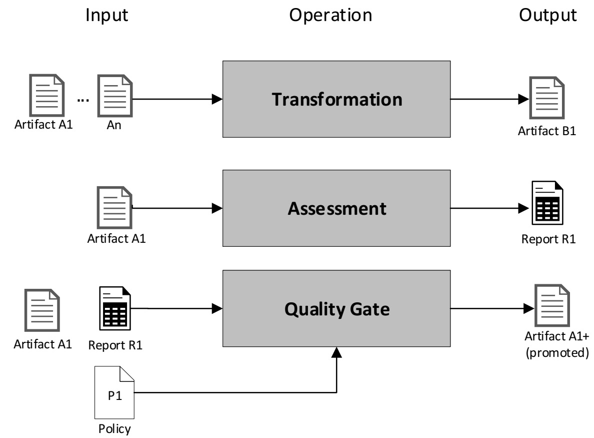

Figure 3: Activity Classification - Overview

Quality Gates represent decision points in the delivery process to ensure that defined 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. The quality gate interprets the result in the given input report artifact and evaluates if the reported quality characteristics fulfill the expected values specified in a policy. If so, the artifact is promoted. Otherwise, the 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.

Table 1 classifies some well known delivery process activities according to our classification.

4th International Workshop on Rapid Continuous Software Engineering (RCoSE '18). ACM, New York, NY, USA, 1-7
DOI: https://doi.org/10.1145/3194760.3194768

© Copyright 2018 ACM
This is a draft version of the paper, for private use only!

Activity	Activity Type	Input	Output
Compile	Transformation	Source Code	Binaries
Unit Tests	Assessment	Binaries	Test Report
Static Code Analysis	Assessment	Source Code Binaries	Analysis Report
Check Smells	Quality Gate	Report Binaries Policy	Promoted Binaries
Bake/Package	Transformation	Binaries	Deployable Package
Deploy	Transformation	Deployable Package	Deployed System
Acceptance Testing	Assessment	Deployed System	Acceptance Report

Table 1: Delivery Process Activities

This classification of activities and the resulting abstraction of the delivery model forms the internal delivery model (highlighted in light grey in Figure 2) , which can act as a meta-model. In a later section we will introduce models, which can be mapped to the meta-model. They are represented as external delivery models in our domain model.

Each activity provides an activity specification which includes an unique identifier, a configuration or input model and a result model. The configuration model specifies the admissible input properties and the result model defines permitted results like artifacts and meta-data. Using this activity specifications the delivery system can reason about the whole delivery model, e.g. the compatibility of consecutive activities. It also allows the system to automatically calculate dependencies between activities. With the possibility to compute dependencies, validate sequences of activities or in general to reason about a model the delivery system minimizes the required information provided by the user.

The remaining concepts of our domain model are related to the execution of a delivery process. During the execution instances of the modeled activities will be triggered and will produce activity results, which mainly consists of artifacts and meta-data, e.g., performance data. The core domain includes the static and dynamic aspects of the software delivery domain and how these interact with each other inside a delivery system.

To ease the understanding of a domain model, Evans suggests to present the concepts in an explanatory model [7]. This model does not need to correspond in every detail with the domain model.

Figure 4 provides an explanatory model for our core domain presented above. It focuses on modeling the general delivery process building blocks and dynamics. At its heart the modeled delivery process consists of a series of transformations, which constitute an artifact value stream. Only these transformations are mandatory. The transformation activities can be parametrized with configuration from an environmental level. In case of a quality gates, this configuration typically comprises a policy defining artifact acceptance criteria, e.g. coverage thresholds or the amount or identified code smells. Following our activity classification, the quality characteristics 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. As assessments are optional, a valid delivery process may consists only of transformations, a minimal process may only have one transformation.

5 ARCHITECTURE

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 [7]. Our software delivery domain model, which represents our core domain, is accompanied by four sub-domains. We can identify the model sub-domain, which is responsible for handling the various possibilities of describing a delivery process. Our approach introduces the concept of external delivery model which is transformed to an internal delivery model based on our domain model. Via this separation a delivery system is able to support various external models or languages.

A key concept in the domain model is the abstraction of activities as they represent delivery process building blocks. While the concept of an activity and its specification is stable, each concrete activity respectively its implementation has several related concepts not relevant for the intended delivery system, e.g. a deployment activity to Amazon EC2 uses concepts like buckets. This is a key decision to achieve a technical agnostic delivery system and to manage the required heterogenous technologies.

The orchestration and execution of tasks is a needed functionality for a delivery system. As there exists proven concepts for orchestration and many off the shelf solutions for this, we define orchestration as a generic sub-domain. The same holds for the area of artifact management and storage. In our case all subdomains correspond exactly to one bounded context[27].

Given the focus on providing an architecture providing a high degree of flexibility and maintainability, 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 the *microservice architectural style* [19].

James Hugh [14] states that this architectural style realizes the SOLID principles, a set of known and proven design principles that improve both flexibility and maintainability. [1]

Overall, the microservices architectural style harmonizes with the software delivery domain model and the corresponding design decisions, like the encapsulation of activities.

5.1 Architecture Blueprint

Applying microservices to our core domain and bounded contexts results in the architecture depicted in Figure 5, which organizes the microservices in layers [8]. To prevent misunderstandings we explicitly want to highlight that each depicted service is isolated and individually deployable.

The Activity Layer houses *Activity Services* that realize coherent activities in a self-contained manner. Typically, each activity microservice encapsulates the functionality of a tool or technology. All activity service register their implemented activities during

4th International Workshop on Rapid Continuous Software Engineering (RCoSE '18). ACM, New York, NY, USA, 1-7
DOI: https://doi.org/10.1145/3194760.3194768

© Copyright 2018 ACM
This is a draft version of the paper, for private use only!

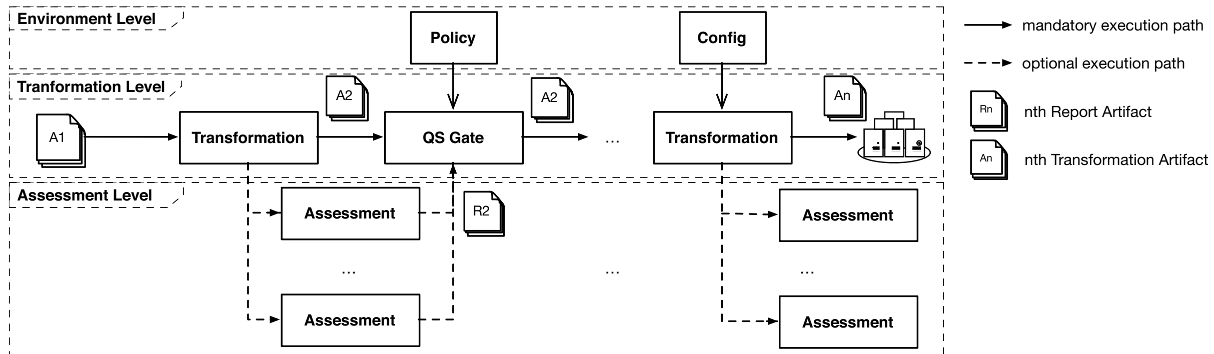

Figure 4: Core Domain - Explanatory Model

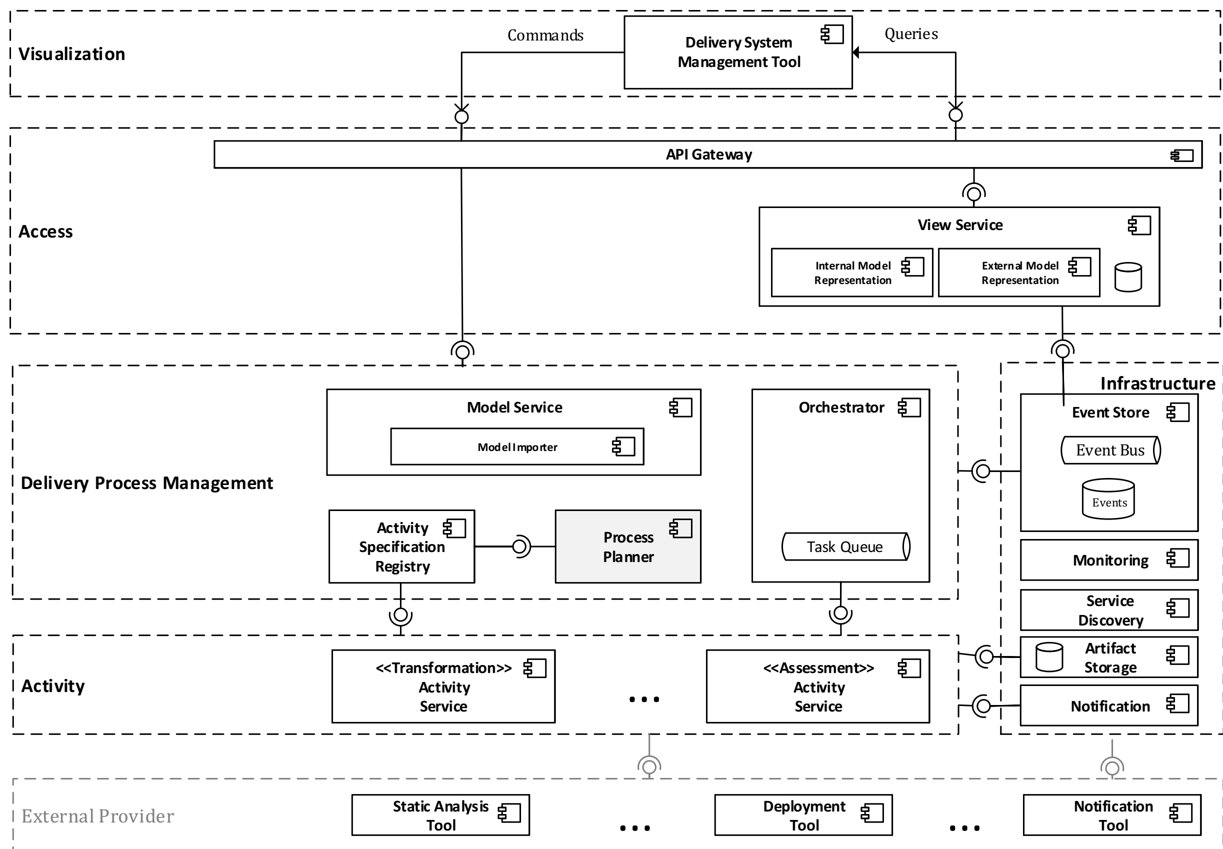

Figure 5: Layers and services in the delivery system architecture

startup by publishing the corresponding activity specifications at the Activity Specification Registry. 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. An activity acts in this case as an adapter to these external services. Following our core domain 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 it only provides assessment activities. The service can also provide activities of both types. We then consider it a *hybrid service*.

The Delivery Process Management Layer dedicated contains services required for managing, analyzing, optimizing and executing delivery models. From a domain driven design perspective this

4th International Workshop on Rapid Continuous Software Engineering (RCoSE '18). ACM, New York, NY, USA, 1-7
DOI: https://doi.org/10.1145/3194760.3194768

© Copyright 2018 ACM
This is a draft version of the paper, for private use only!

includes the model context, the orchestration context and the core context. Thus, we consider the management layer as the main area of our framework. The *model service* imports, validates and transforms external delivery models into the internal delivery model, a component called the process planner which optimizes a given internal delivery model, and the *orchestrator* which acts as a process manager [9], that controls the orchestration of the aforementioned activity services. The activity specification registry, following the registry pattern [8]) is also included. It keeps track of all available activities and their activity specifications provided by the activity services.

Summing up, the separation of concerns between the import, the planning and the execution control increases robustness and allows to easily extend and modify each aspect, e.g. to use off-the-shelf solutions e.g. the orchestrator. To employ isolation, loose coupling and location transparency between these services, they communicate asynchronously via events.

5.2 Architecture Control Flow

Figure 6: Architecture Control Flow

The delivery system architecture provides several hotspots to adapt functionality. To control the flexibility at individual service level, the architecture defines a higher-level control flow. We call this flow *Architecture Control Flow*. Figure 6 depicts its three phases.

The import phase of the architecture control flow fetches the latest version of the referenced external delivery model and transforms the model into a representation of the internal delivery model. Its implemented by the model service. Providing additional model to model transformation components or services the delivery system can be extended to support additional models and languages, e.g. the Pipeline DSL from Jenkins[6].

The second phase initiated by the architecture control flow is planning. In this phase the automatic completion and optimization of an internal delivery model will be performed. On receiving a internal delivery model, the planning process starts.

Following defensive programming practices, the planner first validates the received delivery model. Since the model can be incomplete at this early stage in the process, the validation is limited to a technical level (e.g. syntax). Our architecture supports the usage of multiple planners to process a delivery model. We identified two strategies for planning, the *model-based* and the *project-based* planning.

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

Project-based Planner apply the project-based planning strategy. As such, they are typically technology-specific. They use additional project-specific information sources like the code, configuration or documentation. An example for a project planner is a maven multi-module planner, which 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. Project-based planning therefore can be more expensive than model-based planning but may be more powerful as it uses a larger knowledge base (project sources).

During the planning process matching planners are dynamically selected and will be executed sequentially. A project-based planner therefore can optimize a model, which was derived or completed by a model-based planner before. So the model can be iteratively improved. A delivery system can be extended by providing additional planner implementation tailored to the needs and scope of the development process and used technologies.

The power of this approach can produce a fully automated delivery model and process generation without any required information about the build process provided by the users. In the end this would solve the usability challenge and is the envisioned long-term result of our approach.

The last phase of the architecture control flow is the execution, in which the orchestrator processes the final generated and optimized delivery model and triggers the required activity services. In this phase the architecture can be extended by adding additional activity.

Our approach forms an architectural framework for software delivery systems.

6 RELATED WORK

The previous sections identified challenges software delivery systems face and introduced an architecture to meet these challenges. This section evaluates existing software delivery systems. We only evaluate bleeding-edge software delivery systems.

Spinnaker[1] is a cloud-deployment-focused delivery system, open-sourced by Netflix in November 2015. Main focus of Spinnaker is to decouple deployment activities from a specific cloud provider. As of today various cloud providers like Amazon, Google, Microsoft and Openstack are supported. Spinnaker originated from Netflix's previous cloud-deployment focused delivery system, Asgard. The Netflix team noticed that Asgard was not flexible and extensible enough to support their growing needs, thus they chose to design Spinnaker using the microservice architectural style.

Spinnaker handles only a part of the delivery process and set it focus to deployment. Therefore a dedicated Continuous Integration Tool (e.g. Jenkins) is required to handle the previous stages. 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 [22] to allow for *pipeline as code*.

Concourse [2] 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

[1] https://www.spinnaker.io/
[2] https://concourse.ci/

4th International Workshop on Rapid Continuous Software Engineering (RCoSE '18). ACM, New York, NY, USA, 1-7
DOI: https://doi.org/10.1145/3194760.3194768

© Copyright 2018 ACM
This is a draft version of the paper, for private use only!

to encapsulates and run all delivery process activities. Overall concourse uses three core concepts: Resource, Task and Job.

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.

Initially developed as a web-based git repository manager, Gitlab[3] evolves to - as they call themselves - *the leading integrated product for the entire software development lifecycle*. Delivery processes in GitLab CI / CD are defined declaratively via pipeline as code. The delivery model contains a set of jobs with constraints specifying certain execution conditions (e.g. only execute on master branch). Shell scripts are defined for single jobs, which are executed sequentially. To execute such a job, Gitlab uses several independent GitLab Runners. The delivery process activities are executed inside such runner. Thus runners provide means to integrate new technology.

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. The idea is to automatically detect the project technologies and to generate an opinionated delivery process based on best practices. The Auto DevOps feature is based on a delivery model template.

In summary Spinnaker, Concourse and Gitlab adopt similar ideas like microservices to tackle similar challenges. But in general they miss to tackle the usability challenge and focus on the technical challenge to support various technologies and tools. Mechanisms to automatically derive and process delivery models are an very early stage.

7 CONCLUSION & FUTURE WORK

Adopting Continuous Delivery is hard. An organization needs introduce a lot of new concepts, methods and techniques. Developers are often overwhelmed by the requirements when switching to continuous delivery. This paper identified two key challenges which provides severe obstacles for the adoption of continuous delivery. The challenges can be observed in the software system implementing the delivery process.

The concepts and the architecture introduced in this paper offer a consistent blueprint to build a next-generation of continuous delivery systems. These systems can adapt to the evolution of the software projects, processes and the organization.

Therefore, the approach provides an extensible framework to include more user support mechanisms into the process of modeling and executing the delivery process. A long-term vision is to achieve a fully automated generation of the delivery process by analyzing all available sources of information, which would free developers to deal with the delivery process and to focus developing value for the customer.

Our approach has been implemented in a prototype called JARVIS and has been validated by conducting a small case study. The case study showed promising results like improving fast feedback on errors and a higher performance. Due to the limited space in this paper the case study could not be included.

[3] https://docs.gitlab.com/ce/development/architecture.html

In the future, we want to evaluate the potential of our approach in more detail with more and bigger case studies. Especially incorporating more and advanced analytics, AI planning and machine learning to build more powerful and smarter process planners promises great potential. Another possibility will be to include additional stakeholders from other areas like business IT alignment to integrate continuous delivery with further activities inside the organization. But in general the open design of our approach offers a broad range of future work.

REFERENCES

[1] Derick Bailey. 2009. S.O.L.I.D. Software Development, One Step at a Time. *CODE Magazine, 2010 Jan/Feb* (2009). http://www.codemag.com/article/1001061
[2] Len Bass, Ingo Weber, and Liming Zhu. 2015. *DevOps: A Software Architect's Perspective* (1st ed.). Addison-Wesley Professional.
[3] Kent et al. Beck. 2001. Agile Manifesto. (2001), 28–35 pages. http://agilemanifesto.org/
[4] Lianping Chen. 2017. Continuous Delivery: Overcoming adoption challenges. *Journal of Systems and Software* 128 (jun 2017), 72–86.
[5] Gerry Gerard Claps, Richard Berntsson Svensson, and Aybüke Aurum. 2015. On the journey to continuous deployment: Technical and social challenges along the way. In *Information and Software Technology*, Vol. 57. 21–31.
[6] Cloudbees Inc. [n. d.]. Jenkins. ([n. d.]). https://jenkins-ci.org/
[7] Eric Evans. 2003. *Domain-Driven Design: Tacking Complexity In the Heart of Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
[8] Martin Fowler. 2002. *Patterns of Enterprise Application Architecture.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
[9] Gregor Hohpe and Bobby Woolf. 2003. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* 736 pages.
[10] Jez Humble and David Farley. 2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* (1st ed.). Addison-Wesley Professional. http://dl.acm.org/citation.cfm?id=1869904
[11] Ieee. 1990. IEEE Standard Glossary of Software Engineering Terminology. *Office* 121990, 1 (1990), 1. https://doi.org/10.1109/IEEESTD.1990.101064
[12] IEEE. 2002. *IEEE Standard 610.12-1990 Glossary of Software Engineering Terminology (Reaffirmed 2002).* Vol. 121990. 1 pages. http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=159342
[13] ISO 25000. 2015. ISO 25010. (2015), 3 pages. http://iso25000.com/index.php/normas-iso-25000/iso-25010
[14] James Hugh. 2013. Micro Service Architecture. (2013). https://yobriefca.se/blog/2013/04/29/micro-service-architecture/
[15] Eero Laukkanen, Juha Itkonen, and Casper Lassenius. 2017. Problems, causes and solutions when adopting continuous delivery—A systematic literature review. *Information and Software Technology* 82 (feb 2017), 55–79.
[16] Meir M. Lehman. 1980. Programs, Life Cycles, and Laws of Software Evolution. *Proc. IEEE* 68, 9 (1980), 1060–1076.
[17] Martin Fowler. 2012. SnowflakeServer. (2012). https://martinfowler.com/bliki/SnowflakeServer.html
[18] Martin Fowler. 2016. InfrastructureAsCode. (2016). https://martinfowler.com/bliki/InfrastructureAsCode.html
[19] Martin Fowler and James Lewis. 2014. Microservices. (2014). https://martinfowler.com/articles/microservices.html
[20] Shane McIntosh, Bram Adams, Thanh H.D. Nguyen, Yasutaka Kamei, and Ahmed E. Hassan. 2011. An empirical study of build maintenance effort. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11.*
[21] Perforce Software Inc. 2015. Continuous Delivery: The New Normal for Software Development. (2015). http://www.perforce.com/continuous-delivery-report
[22] Rob Zienert. 2017. Codifying your Spinnaker Pipelines – The Spinnaker Community Blog. (2017). https://blog.spinnaker.io/codifying-your-spinnaker-pipelines-ea8e9164998f
[23] Pilar Rodríguez, Jouni Markkula, Markku Oivo, and Kimmo Turula. 2012. 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.* ACM Press, New York, New York, USA, 139.
[24] Pilar et al. Rodriguez. 2017. Continuous deployment of software intensive products and services: A systematic mapping study. *Journal of Systems and Software* 123 (2017), 263–291.
[25] Ken Schwaber and Mike Beedle. 2001. *Agile Software Development with Scrum* (1st ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.
[26] Thoughworks Inc. 2017. A single CI instance for all teams | Technology Radar | ThoughtWorks. (2017). https://www.thoughtworks.com/de/radar/techniques/a-single-ci-instance-for-all-teams
[27] Vaughn Vernon. 2016. *Domain-driven design distilled.* Addison-Wesley.

4th International Workshop on Rapid Continuous Software Engineering (RCoSE '18). ACM, New York, NY, USA, 1-7
DOI: https://doi.org/10.1145/3194760.3194768