On the Design and Architecture of Deployment Pipelines in Cloud- and Service-Based Computing – A Model-Based Qualitative Study

Uwe Zdun, Evangelos Ntentos, Konstantinos Plakidas, Amine El Malki
University of Vienna, Vienna, Austria
Faculty of Computer Science, Research Group Software Architecture
firstname.lastname@univie.ac.at

Daniel Schall, Fei Li
Siemens Corporate Technology
Vienna, Austria
firstname.lastname@siemens.com

***Abstract*—DevOps and Continuous Delivery (CD) are becoming the de-facto standard for software deployment in the cloud. Deployment pipelines are a core artefact in such practices, but so far their design is largely discussed informally, and existing sources on DevOps and CD practices are often inconsistent or incomplete. The lack of a generic, complete, tool-agnostic, and application-independent treatment of deployment pipeline design and architecture impedes their understanding by human designers and the creation of generic tools that work across different technologies and applications. To alleviate this problem, we have performed a qualitative, in-depth study of 25 deployment practice descriptions by practitioners containing informal deployment pipeline models. From our study we derived a precisely specified model of deployment pipeline architectures. We can show that the formal model substantially increases the precision of the modelling compared to the informally modelled pipelines in the original sources.**

I. Introduction

A core trend in cloud- and service-based computing is the steady increase of the frequency of change required in those systems: continuous releases are becoming the expected norm rather than the exception. Deployment pipelines [1] are the centrepiece of most DevOps and CD practices. So far the design of deployment pipelines is largely discussed informally (cf. [2], [3]). At a first glance, such informal models by practitioners give a good overview of all deployment structures. However, experience in the field and a detailed study of these informal descriptions immediately reveal that almost all realistic deployment pipelines require complex architectures of various tools and components, including deployment target environments and tool integration architectures (cf. [1]). The complexity of this overall architecture is usually not covered well in the informal models, nor are all aspects of the deployment pipeline itself. This impedes building generic tools operating on the whole pipeline, such as architecture consistency checkers or static analysis tools. Common abstractions, and consequently interfaces, are missing, which hinders understanding by human designers beyond a single set of technologies, environments, and applications.

To alleviate these problems, we have performed a qualitative, in-depth study of 25 deployment practice descriptions by practitioners containing informal deployment pipeline models. We followed the model-based qualitative research method described in [4], based on the established Grounded Theory (GT) [5] qualitative research method, together with methods for studying established practices like pattern mining and their combination with GT [6]. The knowledge-mining procedure is applied in many iterations until reaching theoretical saturation [5], as is widely accepted in qualitative research. In our study, we decided to stop our analysis when 7 additional knowledge sources did not add anything new to our understanding of the research topic. This is a rather conservative operationalisation of theoretical saturation; our study converged already after 10 knowledge sources in the sense that no substantial new formal model elements were created. We used this method to examine the following research questions:

RQ1 What are recurring established practices for designing deployment pipeline structures?

RQ2 What are the relevant environments in deployment pipelines?

RQ3 What are the architectural elements relevant for building a deployment pipeline infrastructure?

Our result is a precisely specified model comprising views for modelling deployment pipeline structures, deployment environments, and infrastructure architecture. We also precisely define the links between the model elements in each view, as well as consistent links between the different views. For each of the informal pipeline models studied, we contribute a precisely modelled instance of our model. Finally, in a preliminary evaluation, we can show that the formal models result in a total average improvement of 134.72% in modelling accuracy compared to the informally modelled pipelines in the original sources[1].

II. Related Work

Informal descriptions of deployment pipelines and associated architectures dominate the literature (cf. [7], [1], [3]),

[1]For space reasons, we have omitted the sources used and a thorough discussion and evaluation of the model process and results. We refer the interested reader to an online long version of this paper: https://doi.org/10.5281/zenodo.2671625

but they usually fail to fully cover either the complexity of these designs and architectures or all aspects of the deployment pipeline itself, and are often inconsistent. While many scientific works use and improve deployment pipelines (cf. [8], [9]), and first studies on deployment practices in organizations have emerged [10], [11], a generic, tool-agnostic, and application-independent treatment of deployment architectures is missing today. Our study aims to provide the first systematic and precise specification approach for CD architectures, laying the foundations for automated quality control of the design of deployment architectures. This would enable checking, e.g., whether a pipeline design performs too few or too manual quality controls, is missing important steps (like forgetting to model a commit trigger), links to the environment (like a cloud test environment that is launched but not torn down), or is performing time-consuming or resource-intensive steps too early. So far, such design issues have been identified in the literature as red flags [1], [2], but their automatic detection is not possible as it requires precise models of all elements of the deployment architectures.Many sources also point at the need to substantially alter the architecture of the target systems for supporting rapid releases [9], [10], [11]. For instance, decomposition of a system into microservices is extensively studied (cf. [8], [12], [13]), but the impact on the associated deployment architectures has not yet been studied in a systematic way. Our work aims to provide the groundwork needed to perform such studies, by enabling formal reasoning, validation, and verification through a precise and consistent modelling foundation for the CD parts.

III. CD Pipeline Model

Our studies have led to a model DOM for CD pipelines which formally is a tuple (CP, CN, $dtype_{CP}$, $type_{CP}$, CPT, $stype_{CPT}$, $dtype_{CN}$, $type_{CN}$, CNT, $stype_{CNT}$, DN, NH, $type_{NH}$, NHT, DR, $type_{DR}$, DRT, DE, EE, $dtype_{EE}$, $type_{EE}$, EET, $stype_{EET}$, AN, AE, CON, IN, FIN, FON, JON, DEN, MEN, ACT, AEA, SSA, PE, PN, $dtype_{PN}$, $type_{PN}$, PNT, $stype_{PNT}$, PAE, $type_{PAE}$, $PAET$, PSS, $type_{PSS}$, $PSST$, PDN, $type_{PDN}$, $PDNT$). All the tuple elements are defined in the subsections below. We first discuss two prerequisites for modelling deployment pipelines: components of the deployment infrastructure and deployment environments. Then we discuss specifying the structure of the deployment pipeline. We have made our generated models available online[2].

Each aspect of our model is discussed in two parts: First we discuss generic modelling notions that should suffice for modelling the elements of a CD model instance and their relations. Second, based on the recurring CD-specific elements found in our study, we specify CD-specific set members and rules (all summarized in Table I). For instance, we first define the generic notion of a pipeline node and then specify all the possible pipeline node types and their type hierarchy relations that we have observed in our study in Table I. We expect that the generic aspects will likely remain stable in the future, whereas the elements in Table I might require changes or extensions. Please note that we consider this list of elements in Table I complete with regard to the sources we have studied, but these sets and rules can be extended or redefined when using or applying our model (e.g., for modelling CD/DevOps aspects we have not yet covered in our study, or for future technologies).

A. Modelling the Deployment Infrastructure Architecture

An important result of our study was that the focus of informal descriptions of deployment pipelines is only in exceptional cases solely on the structure of the pipeline. Instead, almost always the components which represent the infrastructure of the deployment pipeline, such as continuous integration (CI) tools or deployment pipeline orchestration components, and their interconnections are described as well. To formally capture this, we first model component nodes and their connectors: CP is a finite set of **component nodes**. $CN \subseteq CP \times CP$ is a finite set of **connector edges**.

CD infrastructure components are typically categorized along their main function, which can be modelled using types in type hierarchies. For example, deployment pipeline orchestration and package tools are important recurring types of such components, and package tool is a subtype of development tool. In our model, component types are defined as follows: CPT is a finite set of **component types**. $stype_{CPT} : CPT \rightarrow \mathbb{P}(CPT)$ is a function called **component type hierarchy**. $stype_{CPT}(cpt)$ (with $cpt \in CPT$) is the set of direct supertypes of cpt; cpt is called the subtype of those supertypes. The transitive closure $stype^*_{CPT} = \bigcup_{i=0}^{\infty} stype^i_{CPT}$ defines the inheritance in the hierarchy such that $stype^*_{CPT}(cpt)$ (with $cpt \in CPT$) contains the **direct and indirect supertypes** of cpt. The inheritance hierarchy is cycle free, i.e., $\forall cpt \in CPT : stype^*_{CPT}(cpt) \cap \{cpt\} = \emptyset$. $dtype_{CP} : CP \rightarrow \mathbb{P}(CPT)$ is a function that maps each component node $cp \in CPT$ to its set of **direct component types**. $type_{CP} : CP \rightarrow \mathbb{P}(CPT)$ is a function that maps each component node $cp \in CPT$ to its set of **direct and transitive types**, i.e., $\forall cp \in CP, dt \in dtype_{CP}(cp) : type_{CP}(cp) \supseteq \{dt\} \cup stype^*_{CPT}(dt)$.

CD infrastructure connectors have types and a type hierarchy, too; e.g., components can launch another component or read an artefact from another component, and here *launch* and *read* are connector types. In our model, CNT is a finite set of **connector types**. It has a type hierarchy definition exactly identical to the one of CPT (see specification above) with analogous function definitions for $stype_{CNT}$, $dtype_{CN}$, and $type_{CN}$ (omitted here for brevity).

In our study we found a number of recurring types of components and connectors used in infrastructure architectures

[2]https://swa.univie.ac.at/cd-pipeline-models/cd-pipeline-models.zip

of deployment pipelines. All component and connector types that were included in our study according to our inclusion criteria, as well as their relations in two type hierarchies, are formally defined in the first four rows of Table I.

B. Modelling Deployment Environments

A second prerequisite for precisely specifying a deployment pipeline, which is used in almost all our sources, is the notion of deployment environments. They are used first to model the deployment environments to which the pipeline deploys, such as a test environment in a virtual private cloud or a production environment in a public cloud. Second, they are used to describe the environments in which the deployment infrastructure (see previous section) itself is deployed. For instance, sometimes the deployment pipeline orchestrator or a continuous integration tool run in the same cloud environment the system is deployed to, or a local or server environment are distinguished from a cloud environment if both are used in a pipeline.

The main deployment environment elements of our model are the deployment nodes: DN is a finite set of **deployment nodes**. These can be connected with each other, such as a production and a test environment running on a cloud environment: $DNR \subseteq DN \times DN$ is a finite set of **deployment node relations**. Different types of relations might exist such as *part-of*, *connects-to*, or *runs-on*: $DNRT$ is a finite set of **deployment node relation types**. $type_{DNR} : DNR \rightarrow DNRT$ is a function that maps each deployment node relation $dnr \in DNRT$ to its **type**.

Components of the deployment infrastructure have relations to these deployment nodes: $DR \subseteq CP \times DN$ is a finite set of **deployment relations**. Different types of deployments exist such as *deployed-on*, *uses*, or *launches*: DRT is a finite set of **deployment relation types**. $type_{DR} : DR \rightarrow DRT$ is a function that maps each deployment relation $dr \in DRT$ to its **type**.

There are specific kinds of deployment nodes: $DE \subseteq DN$ is a finite set of **devices**. $EE \subseteq DN$ is a finite set of **execution environments**. EE is used to model the environments a system can be deployed to, which is modelled in a type hierarchy: EET is a finite set of **execution environment types**. It has a type hierarchy definition exactly identical to the one defined for CPT (see specification above in Section III-A) with analogous function definitions for $stype_{EET}$, $dtype_{EE}$, and $type_{EE}$ (omitted here for brevity).

Again, we have specified those CD-specific set members and type hierarchy rules that we have observed in our study as well. Rows 5–8 of Table I contain formal definitions for the environment types, their type hierarchy, and their relations that we have observed in the informal deployment pipeline descriptions analysed in this study.

C. Modelling Deployment Pipeline Structures

Deployment pipelines are often modelled as behaviour models resembling UML activities. As a basis for modelling pipeline specifics we thus have chosen abstractions resembling the basic elements of activities – but excluded all abstractions of activities in UML that we have not empirically observed in our study – to keep our model much simpler than UML activities: AN is a finite set of **activity nodes**. $AE \subseteq AN \times AN$ is a finite set of **activity edges**. $CON \subseteq AN$ is a finite set of **control nodes**. $IN \subseteq CON$ is a finite set of **initial nodes**. $FIN \subseteq CON$ is a finite set of **final nodes**. $FON \subseteq CON$ is a finite set of **fork nodes**. $JON \subseteq CON$ is a finite set of **join nodes**. $DEN \subseteq CON$ is a finite set of **decision nodes**. $MEN \subseteq CON$ is a finite set of **merge nodes**. $ACT \subseteq AN$ is a finite set of **actions**. $AEA \subseteq ACT$ is a finite set of **accept event actions**. $ATA \subseteq AEA$ is a finite set of **accept time event actions**. $SSA \subseteq ACT$ is a finite set of **send signal actions**.

All special kinds of nodes in a deployment pipeline are subsets of some of those activity nodes. In addition they are subsets of PE which is a finite set of **pipeline elements**. For PE we define a number of functions used to specify important properties of pipeline elements. $aut : PE \rightarrow \{True, False\}$ is a function that determines whether a pipeline element is automatically processed in the pipeline or requires manual work. The following functions are used to **specify important links to infrastructure components and environments** (as defined in the previous sections): $run : PE \rightarrow CP$ is a function that determines the component this pipeline element runs in. $inv : PE \rightarrow \{(l_1, l_2, ..., l_n) : l_1 \in CP, l_2 \in CP, ..., l_n \in CP\}$ is a function that determines the components this pipeline element can invoke. $inp : PE \rightarrow \{(l_1, l_2, ..., l_n) : l_1 \in CP, l_2 \in CP, ..., l_n \in CP\}$ is a function that determines the components providing inputs to a pipeline element (like an artefact passed to a pipeline element). $out : PE \rightarrow \{(l_1, l_2, ..., l_n) : l_1 \in CP, l_2 \in CP, ..., l_n \in CP\}$ is a function that determines the components providing outputs of a pipeline element (like an artefact produced by a pipeline element) $env : PE \rightarrow \{(l_1, l_2, ..., l_n) : l_1 \in DN, l_2 \in DN, ..., l_n \in DN\}$ is a function that determines the deployment nodes used by a pipeline element.

The core element in a typical deployment pipeline are pipeline nodes, modelled as elements of PN (with $PN \subseteq PE$, $PN \subseteq AN$) which is a finite set of **pipeline nodes**. PNT is a finite set of **pipeline node types**. Exemplary CD-specific PN members are pipeline nodes for building, packaging, unit testing, and so on. PNT has a type hierarchy definition exactly identical to the one defined for CPT (see specification above in Section III-A) with analogous function definitions for $stype_{PNT}$, $dtype_{PN}$, and $type_{PN}$ (omitted here for brevity).

Mainly for triggering the pipeline, we further define special accept event actions: PAE (with $PAE \subseteq PE$, $PAE \subseteq AEA$) is a finite set of **pipeline accept event actions**. $PAET$ is a finite set of **pipeline accept event action types**; it is used to model for instance a trigger by a commit

Table I
CD-SPECIFIC SET MEMBERS AND RULES FOR APPLICATION OF THE CD PIPELINE MODEL

Name	Definition
Component Types	$CPT \supseteq$ *{Version Control Repository, Deployment Pipeline Control UI, Artifact Repository, Deployment Tool, Deployment Pipeline Orchestration, Machine Images Builder, Deployment Target, Collaborative Review Tool, API, Cloud API, Administration Tool, Review Tool, Build Tool, Code Analysis Tool, Test Tool, Package Tool, Continuous Integration Tool, Database, Binary Repository, App Store, Container Manager}*
Component Type Hierarchy	$\forall(c, SCS) \in \{$(*Binary Repository, {Artifact Repository}*), (*App Store, {Binary Repository}*)$\}$: $stype_{CPT}(c) = SCS$
Connector Types	$CNT \supseteq$ *{checks in, checks out, reads artifacts, writes artifacts, deploys artifacts, reads images, writes images, deploys images, uses, extends, launches, API call}*
Connector Type Hierarchy	$\forall(c, SCS) \in \{$(*reads images, {reads artifacts}*), (*writes images, {writes artifacts}*), (*deploys images, {deploys artifacts}*)$\}$: $stype_{CNT}(c) = SCS$
Deployment Relation Types	$DRT \supseteq$ *{deployed on, uses, launches, provides deployment artifacts}*
Deployment Node Relation Types	$DNRT \supseteq$ *{part of, runs on, connects to}*
Execution Environment Types	$EET \supseteq$ *{Cloud, Public Cloud, Private Cloud, Virtual Private Cloud, Server, Virtual Machine, Container, Cluster, Test Environment, On-Premises, Datacenter, Production Environment}*
Execution Environment Hierarchy	$\forall(c, SCS) \in \{$(*Public Cloud, {Cloud}*), (*Private Cloud, {Cloud}*), (*Virtual Private Cloud, {Cloud}*)$\}$: $stype_{EET}(c) = SCS$
Pipeline Node Types	$PNT \supseteq$ *{Deployment to Production, Partial Rollout, Canary Release Deployment, Blue/Green Deployment, Dark Launch Deployment, A/B Test Deployment, Deployment Notification, Build, Package, Publish Package, Code Analysis, Code Review, Code Internal Use and Review, Code Peer Review, System Tests, Formal Code Review, Machine Image Build, Container Image Build, Generate Documentation, Tests, Unit Tests, Regression Tests, Integration Tests, Quality Assurance Tests, System Acceptance Tests, User Acceptance Tests, Production Validation Tests, Automated User Interface Tests, Performance Tests, Security Tests, Operations Tests, Smoke Test, Infrastructure Smoke Test, Infrastructure Test, Exploratory Test, Resilience Test, Create Environment, Teardown Environment, Configure Environment, Create Test Environment, Teardown Test Environment, Configure Test Environment, Deployment, Deployment to Test Environment}*
Pipeline Node Hierarchy	$\forall(c, SCS) \in \{$(*Deployment to Production, {Deployment}*), (*Partial Rollout, {Deployment}*), (*Canary Release Deployment, {Partial Rollout}*), (*Blue/Green Deployment, {Partial Rollout}*), (*Dark Launch Deployment, {Partial Rollout}*), (*A/B Test Deployment, {Deployment}*), (*Code Internal Use and Review, {Code Review}*), (*Code Peer Review, {Code Review}*), (*System Tests, {Tests}*), (*Formal Code Review, {Code Review}*), (*Unit Tests, {Tests}*), (*Regression Tests, {Tests}*), (*Integration Tests, {Tests}*), (*Quality Assurance Tests, {Tests}*), (*System Acceptance Tests, {Tests}*), (*User Acceptance Tests, {Tests}*), (*Production Validation Tests, {Tests}*), (*Automated User Interface Tests, {Tests}*), (*Performance Tests, {Tests}*), (*Security Tests, {Tests}*), (*Operations Tests, {Tests}*), (*Smoke Test, {Tests}*), (*Infrastructure Smoke Test, {Tests}*), (*Infrastructure Test, {Tests}*), (*Exploratory Test, {Tests}*), (*Resilience Test, {Tests}*), (*Create Test Environment, {Create Environment}*), (*Teardown Test Environment, {Teardown Environment}*), (*Configure Test Environment, {Configure Environment}*), (*Deployment to Test Environment, {Deployment}*)$\}$: $stype_{PNT}(c) = SCS$
Accept Event Action Types	$PAET \supseteq$ *{Accept Event Action, Poll for Event, Triggered by Commit Event, Triggered by Manual Start, Triggered by External Event}*
Accept Event Action Hierarchy	$\forall(c, SCS) \in \{$(*Poll for Event, {Accept Event Action}*), (*Triggered by Commit Event, {Accept Event Action}*), (*Triggered by Manual Start, {Accept Event Action}*), (*Triggered by External Event, {Accept Event Action}*)$\}$: $stype_{PAET}(c) = SCS$
Send Signal Action Types	$PSST \supseteq$ *{Send Signal Action, Scheduled Event, Scheduled Commit Event, Trigger Event, Commit Event}*
Send Signal Action Hierarchy	$\forall(c, SCS) \in \{$(*Scheduled Event, {Send Signal Action}*), (*Scheduled Commit Event, {Commit Event}*), (*Trigger Event, {Send Signal Action}*), (*Commit Event, {Send Signal Action}*)$\}$: $stype_{PSST}(c) = SCS$
Decision Node Types	$PDNT \supseteq$ *{Pipeline Decision Node, Approval Gate}*
Decision Node Hierarchy	$\forall(c, SCS) \in \{$(*Approval Gate, {Pipeline Decision Node}*)$\}$: $stype_{PDNT}(c) = SCS$

event vs. a manual trigger. $type_{PAE} : PAE \rightarrow PAET$ is a function that maps each pipeline accept event action $pae \in PAET$ to its **type**.

To model commit events (which can also happen during a pipeline run), we model a special send signal action: PSS (with $PSS \subseteq PE$, $PSS \subseteq SSA$) is a finite set of **pipeline send signal actions**. $PSST$ is a finite set of **pipeline send signal action types**. $type_{PSS} : PSS \rightarrow PSST$ is a function that maps each pipeline send signal action $pss \in PSST$ to its **type**.

Finally, for defining decision such as approval gates, we model a special decision node: PDN (with $PDN \subseteq PE$, $PDN \subseteq MEN$) is a finite set of **pipeline decision nodes**. $PDNT$ is a finite set of **pipeline decision node types**. $type_{PDN} : PDN \rightarrow PDNT$ is a function that maps each pipeline decision node $pdn \in PDNT$ to its **type**.

Those CD-specific set members and type hierarchy rules that we have observed for pipeline elements in our study are specified in rows 6-11 of Table I. They contain formal definitions for the environment types, their type hierarchy, and their relations we have observed in the informal deployment pipeline descriptions analysed in this study.

IV. THREATS TO VALIDITY

To increase internal validity we decided to use practitioner reports that were produced independent of our study. This avoids any bias, e.g. compared to interviews in which the practitioners would have known that their answers are used in a study. However, this introduces a different internal validity threat: Some important information might be miss-

ing in the reports, which would have been revealed in an interview. We tried to mitigate this threat by looking at many more sources than needed to reach theoretical saturation, as it is unlikely that all different sources miss the same important information. The different members of the author team have cross-checked all models independently to minimize researcher bias. The threat to internal validity that the researcher team is biased in some sense remains, however. The experience and search-based procedure for finding knowledge sources may have introduced some kind of bias as well. However, this threat is mitigated to a large extent by the chosen research method, which requires just additional sources corresponding to the inclusion and exclusion criteria, not a specific distribution of sources. Due to the many included sources, it is likely our results can be generalized to a larger population of similar pipeline models and architectures. However, the threat to external validity remains that our results are only applicable to similar kinds of pipeline models and architectures; generalization to novel or unusual pipeline models and architectures might not be possible without modification of our models.

V. Conclusion

We have performed a qualitative study in which we have studied the design and architecture of deployment pipelines from 25 unique and independent sources. Our study led to a detailed model precisely describing the recurring pipeline structures and their links as an extension of mainly activity model abstractions to answer RQ1. To answer RQ2, we have extended mainly deployment model abstractions to specify the environment in which deployment pipelines run and to which they deploy. Finally, to model the deployment pipeline infrastructures for RQ3, we have extended mainly component model abstractions. In all three cases, we observed theoretical saturation relatively early and could precisely model almost all main concepts described in the original sources. This leads us conclude that the found models are very likely adequate representations of the original sources and can express almost all major concepts expressed therein. We have thoroughly cross-checked all models independently by the different researches in the author team to minimize researcher bias. In addition to the DevOps models as our major contribution, another contribution of our study is a set of 25 formal CD model instances. These might be useful as a basis for further research in the area. In this paper we have used formal models and detailed object notations to present the details of our approach; in practice the models should be rendered using more appealing notations e.g. akin to UML component, deployment, and activity diagrams, which is easily possible as an extension of our model-driven tools. As future work we plan to realize constraint checkers to implement static analysis tools for deployment pipeline architectures. For such tools, precise abstractions as provided in this paper are a necessary prerequisite.

Acknowledgments. This work was supported by: FFG (Austrian Research Promotion Agency) project DECO, no. 846707; FWF (Austrian Science Fund) project ADDCompliance: I 2885-N33

References

[1] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley, 2010.

[2] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*. Addison-Wesley, 2015.

[3] C. Posta, "The hardest part of microservices: Calling your services," http://blog.christianposta.com/microservices/the-hardest-part-of-microservices-calling-your-services/, 2018.

[4] U. Zdun, M. Stocker, O. Zimmermann, C. Pautasso, and D. Lübke, "Supporting architectural decision making on quality aspects of microservice apis," in *16th International Conference on Service-Oriented Computing (ICSOC 2018)*. Hangzhou, Zhejiang, China: Springer, November 2018.

[5] B. G. Glaser and A. L. Strauss, *The Discovery of Grounded Theory*. de Gruyter, 1967.

[6] C. Hentrich, U. Zdun, V. Hlupic, and F. Dotsika, "An approach for pattern mining through grounded theory techniques and its applications to process-driven soa patterns," in *Proceedings of the 18th European Conference on Pattern Languages of Program*, 2015, pp. 9:1–9:16.

[7] K. L. Beck, D. G. Feitelson, and E. Frachtenberg, "Development and deployment at facebook," *IEEE Internet Computing*, vol. 17, pp. 8–17, 2013.

[8] W. Hasselbring and G. Steinacker, "Microservice architectures for scalability, agility and reliability in e-commerce," in *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*. IEEE, 2017, pp. 243–246.

[9] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

[10] G. Schermann, J. Cito, P. Leitner, U. Zdun, and H. C. Gall, "We're doing it live: A multi-method empirical study on continuous experimentation," *Information and Software Technology*, vol. 99, pp. 41–57, 2018.

[11] 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 *38th Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2012, pp. 392–399.

[12] L. Baresi, M. Garriga, and A. De Renzis, "Microservices identification through interface analysis," in *European Conference on Service-Oriented and Cloud Computing*, 2017.

[13] U. Zdun, E. Navarro, and F. Leymann, "Ensuring and assessing architecture conformance to microservice decomposition patterns," in *International Conference on Service-Oriented Computing*. Springer, 2017, pp. 411–429.