Preprint - to be published at
28th Asia-Pacific Software Engineering Conference (APSEC 2021)

Automated Construction of Continuous Delivery Pipelines from Architecture Models

Selin Aydin
Research Group Software Construction
RWTH Aachen University
Aachen, Germany
aydin@swc.rwth-aachen.de

Andreas Steffens
Research Group Software Construction
RWTH Aachen University
Aachen, Germany
steffens@swc.rwth-aachen.de

Horst Lichter
Research Group Software Construction
RWTH Aachen University
Aachen, Germany
lichter@swc.rwth-aachen.de

***Abstract*—Continuous Delivery (CD) aims at reducing the cycle time from changes to software release while also increasing the software quality. To automate CD, delivery process models, defining all delivery activities need to be designed. Quality properties of delivery process models, such as maintainability, still oppose challenges. Previous research indicates that the quality of such models can be improved by aligning them with the software architecture. While software architecture knowledge is only incorporated implicitly, deep technical and process knowledge is required.**

On this basis, this paper introduces a new kind of delivery process models that focus mainly on software architecture knowledge. Hereby, we discard the current activity-centric view and shift to an artifact-centric view. Moreover, we outsource the required process- and technical knowledge to a transformation activities knowledge base.

In order to make an artifact-based delivery process model executable, we provide a model-to-model transformation which constructs a CD pipeline from an artifact-based model with the help of the transformation activities knowledge base. We evaluated our approach by conducting a small industrial qualitative user study. It showed that especially low-experienced developers benefit from the reduced knowledge requirements of the artifact-based modeling approach.

I. Introduction

Continuous Delivery (CD) aims at reducing the cycle time from changes to software release and hereby maintaining or improving the software quality [1]. It tries to ensure that a software is always releasable. For this, the delivery related development tasks such as building, testing, releasing and deploying are organized as a sequence of activities which are automated in order to save time and avoid human errors. The automation is represented by delivery process models that can be executed by various delivery systems.

When modeling activity-based delivery processes, the following problems arise: Developers need to know which specific activities are to be performed and how to specify them. This requires deep technical knowledge [2] about programming languages or the technology used on the target platform.

Furthermore, they need to define the execution order of the delivery activities. When modeling complex delivery processes consisting of several sub-processes, they need to figure out the order of all sub-processes. Thus, the required process-related knowledge is also high.

To solve these problems developers need to know how artifacts evolve from source code to executable/deployable systems and how they are organized.

The knowledge about all central software artifacts (e.g. all components of a software) is defined in the software architecture [3], [4]. Thus, architectural knowledge is required to model a delivery process and should be used as initial input to create a respective delivery process model.

Bass argues that the software architect fulfills as crucial role in the context of software delivery and devops [5]. It is the responsibility of software architects to ensure that software architecture, the developers and the CD pipeline are aligned. Further research shows, that the software architecture influences the shape and complexity of the necessary delivery models and processes [6], [7].

With current concepts of continuous delivery and its activity-based process models the software architecture knowledge cannot be represented explicitly, e.g. separate delivery models are used for each architectural component or subsystem, with additional models needed to manage the integration of the models [1].

Therefore, we propose novel modeling approach which focuses on the software architecture and integrates this knowledge explicitly, especially the architectural components and their artifacts created and used during the delivery process. Hence our approach shifts from an activity to an artifact-centred view.

We argue that developers need less process-related and technical know-how to design an artifact-based delivery process model, compared to the knowledge that is required to create classical activity-based delivery process models.

We claim that each artifact-based model can be automatically transformed into activity-based delivery process model. Hereby we provide reusability of existing continuous delivery systems which require an activity-based delivery process model as input.

Based on this motivation and our goals, we formulate the following research questions to be answered is this paper:

- (RQ1) How can we incorporate existing architectural knowledge to define an artifact-based delivery process model?

- (RQ2) Can an artifact-based delivery process model be transformed into an equivalent activity-based delivery process model?
- (RQ3) How are artifact-based delivery process models perceived by developers with different levels of CD experience?
- (RQ4) Is the needed architectural knowledge to create such delivery process models available and sufficient?

The paper is structured as follows. First, we introduce the concepts of artifact-based delivery process models and how software architecture knowledge is represented therein. As delivery systems execute activity-based process models, next we describe how an artifact-based model can be automatically transformed to a process model used by the delivery system JARVIS. Next, we present the results of a first user study in which developers were asked to model familiar delivery processes with the new artifact-based modeling approach. Finally, the related work is discussed. The paper concludes with a summary of our contributions and some remarks regarding future research.

II. Representing Architecture in Delivery Processes

In order to illustrate the concepts introduced in this chapter, we use the following simple running example.

The modular designed web application "webapp" consists of a frontend and a backend component. The frontend is implemented in HTML/Javascript while the backend is implemented in Java. The project is based on one source-code repository and the web application is deployed as a Docker container.

Usually, delivery processes are structured by means of stages. Researchers and practitioners propose stages which are based on the essential tasks performed by developers [1], [8]–[10], leading to stages such as building, testing, provisioning, deploying and releasing. To construct a staged delivery process for a complex system, each architectural component has to pass the aforementioned stages on its own. Due to the different technologies used, the activities performed differ. Hence, the complete delivery process consists of multiple connected subprocesses for each component [8].

As we aim to create a delivery process model based on the software architecture, architectural components need to be adequately represented in the delivery process. To do so, the concept of a stage cannot be used because it modularizes the process only according to the delivery related development tasks. Therefore, we introduce another modularization concept, called "delivery unit".

A. Delivery Units

*Definition 1 (**Delivery Unit**):* A delivery unit encapsulates all delivery activities and artifacts needed for a specific architectural component.

Just as with stages, a delivery unit processes a determined set of artifacts, which are either consumed or produced within the unit, and exchanges certain artifacts with other units.

In our webapp example, we distinguish the four delivery units "frontend" (fe), "backend" (be), "application" (app) and "container' (co).

As the delivery unit is the top-level concept of artifact-based delivery process models, next we introduce its elements by formalizing the modeling concept of artifacts and their transformation within a delivery unit.

B. Artifacts

Let A be the set of all artifacts, D be the set of all delivery units, and $A_{du_i} \subseteq A$ be the artifacts assigned to delivery unit $du_i \in D$. The artifacts A_{du_i} can be divided into:

- Required artifacts, $A^{req}_{du_i} \subseteq A_{du_i}$, are all artifacts that are needed to create the desired final artifact of the delivery unit. They either exist in an artifact repository or are artifacts resulting from other delivery units.
- Provided artifacts, $A^{prov}_{du_i} \subseteq A_{du_i}$, are the final artifacts produced by this unit. They are exposed to other delivery units for further processing.
- Intermediate artifacts, $A^{inter}_{du_i} = A_{du_i} \setminus (A^{req}_{du_i} \cup A^{prov}_{du_i})$, which are created and used internally.

Since delivery units exchange artifacts it holds that there exist i, j with $i \neq j$ and $A^{req}_{du_i} \cap A^{prov}_{du_j} \neq \emptyset$. A provided artifact from $A^{prov}_{du_i}$ is *reused* as required artifact in $A^{req}_{du_j}$.

Based on our running example we have the following sets:

- A: all artifacts described in the example, i.e. a directory, Java code files, JavaScript code files, JavaScript distribution package, Java classes, a jar file and a Docker image.
- A_{fe}, A_{be}, A_{app}, A_{co} are the sets of artifacts associated with the corresponding delivery units.

A_{app} contains the artifacts: directory, the JavaScript distribution package, Java classes and a jar file, where

- A^{req}_{app} contains the directory and the JavaScript distribution package
- A^{prov}_{app} contains only a jar file.
- A^{inter}_{app} contains Java classes.

In general, delivery units require the existence of certain artifacts in order to create the provided artifacts. To specify such artifacts, artifact type information can be used.

Let R be the set of all possible artifact types and R_{du_i} be the set of artifact types of all artifacts contained in A_{du_i}. The artifacts types R_{du_i} can be divided into:

- Source artifact types $R^{src}_{du_i} \subseteq R_{du_i}$ containing the types of all artifacts $a \in A^{req}_{du_i}$
- Target artifact types $R^{tar}_{du_i} \subseteq R_{du_i}$ containing the types of all artifacts $a \in A^{prov}_{du_i}$
- Intermediate artifact types $R^{inter}_{du_i} = R_{du_i} \setminus (R^{src}_{du_i} \cup R^{tar}_{du_i})$

Using the example artifact types described in table I, we get the following sets with respect to the delivery unit du_{app}:

- R^{src}_{app}={"workspace", "js-dist"}
- R^{tar}_{app}={"java-package"}
- R^{inter}_{app} ={"java-bytecode"}

All artifacts within a delivery unit are either produced or consumed by some transformation activities. A transformation

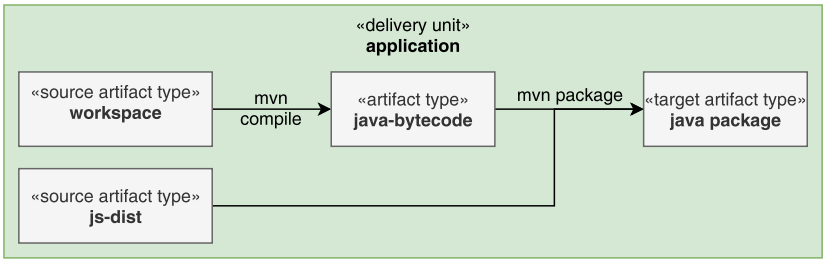

Fig. 1. Example delivery unit artifact model

TABLE I
POSSIBLE ARTIFACT TYPES

Artifact Type	Description
workspace	File container with source code & auxiliary files
java-bytecode	Collection of Java class files
java-package	The Java packaging file format: jar
docker-image	Docker image
js-dist	A JavaScript distribution package

activity presupposes that its input artifacts are of certain artifact types and that it produces output artifacts of defined artifact types.

Based on previous definitions, we further refine the concept of artifact transformations within delivery units.

C. Artifact Transformations

Given $A_1 \subseteq \mathcal{P}(A)$ and $a_1 \in A$. Let T be the set of all transformation functions t with $t : \mathcal{P}(A) \rightarrow A$ modeling existing transformation activities. An application of the transformation function t_1 on artifact set A_1 results in $t_1(A_1) = \{a_1\}$. Thus, the artifacts A_1 can be transformed to artifact a_1 by the respective transformation activity.

A chain of transformation activities, as used in delivery processes, can be expressed by a concatenation of transformation functions. E.g. $(t_2 \circ t_1)(A_1) = t_2(t_1(A_1))$. The artifacts created by applying transformation t_1 on artifact set A_1 are next transformed by t_2.

Accordingly, $T_{du_i} \subseteq T$ is the set of all transformation functions needed in delivery unit du_i. The chain of all transformations within a delivery unit du_i can be expressed as $(t_n \circ ... \circ t_1)(A^{req}_{du_i}) = A^{prov}_{du_i}$ with $t_1,...,t_n \in T_{du_i}$. Hereby, concrete artifact instances of source artifact types $R^{req}_{du_i}$ are transformed to artifact instances with target artifact types $R^{tar}_{du_i}$.

III. ARTIFACT-BASED DELIVERY MODELS

A. Delivery Artifact Models

Next we introduce the *delivery unit artifact model*, which models all transformations and transformation chains of artifacts within a delivery unit. In the sense of abstraction, we replace concrete artifacts by their artifact type.

*Definition 2 (**Delivery Unit Artifact Model**):* A delivery unit artifact model (DU-AM) represents a delivery unit du_i by a tuple $(R_{du_i}, R^{src}_{du_i}, R^{tar}_{du_i}, T_{du_i})$ with

- R_{du_i}: set of artifact type sets associated with du_i
- $R^{src}_{du_i}$: set of source artifact types of du_i
- $R^{tar}_{du_i}$: set of all target artifact types of du_i
- T_{du_i}: set of transformation functions corresponding to transformation activities needed to create artifacts of types $R^{tar}_{du_i}$ based on artifacts of types $R^{src}_{du_i}$

An example is shown in figure 1 which depicts the DU-AM of the delivery unit du_{app}. As the web-application is a jar file which is an instance of the artifact type "java-package", this type is the only target artifact type of this delivery unit. This type is the result of applying a package transformation activity on artifacts of type "java-bytecode" and "js-dist". The "java-bytecode" is the result of a compile transformation activity applied to artifacts of type "workspace". Thus, "workspace" and "js-dist" are required artifact types.

Within a complex delivery process, artifacts which are created should be reused whenever possible to create other artifacts that build upon them. To enable artifact reuse across different DU-AMs, we introduce the so called *delivery artifact model* (D-AM), which explicitly defines which artifacts are reused in the delivery process. Again, its definition is based on the corresponding artifact types.

*Definition 3 (**Delivery Artifact Model**):* A delivery artifact model(D-AM) is represented by a tuple $(DU\text{-}AM, \gamma)$ with:

- $DU\text{-}AM$: a set of $DU\text{-}AM_i$, with $1 \leq i \leq n$.
- γ: artifact type reuse relation with $\gamma_{\subseteq} \cup R^{tar}_{du_i} \times \cup R^{src}_{du_j}$, with $(r^{tar}, r^{src}) \in \gamma$ if the target type $r^{tar} \in R^{tar}_{du_i}$ of deliver unit du_i is reused as source artifact type $r^{src} \in R^{src}_{du_j}$ of delivery unit du_j, with $i \neq j$. The concrete artifact instances of artifact types r^{src} and r^{tar} are identical.

A D-AM is a hybrid model, as it implicitly describes the process of how artifacts are transformed by transformation activities. Each delivery unit could be directly translated to a delivery process model by mapping the included transformation functions to the associated transformation activities. The execution order of the resulting delivery processes could be automatically determined by their artifact dependencies. Hereby, we achieve a partial reduction of process knowledge.

In addition to this, we reduce the required technical knowledge as developers do not need to specify the activities but include the transformation function names.

However, the determination of the execution order of activities within a delivery process still requires a lot of process knowledge. Omitting them would significantly reduce the

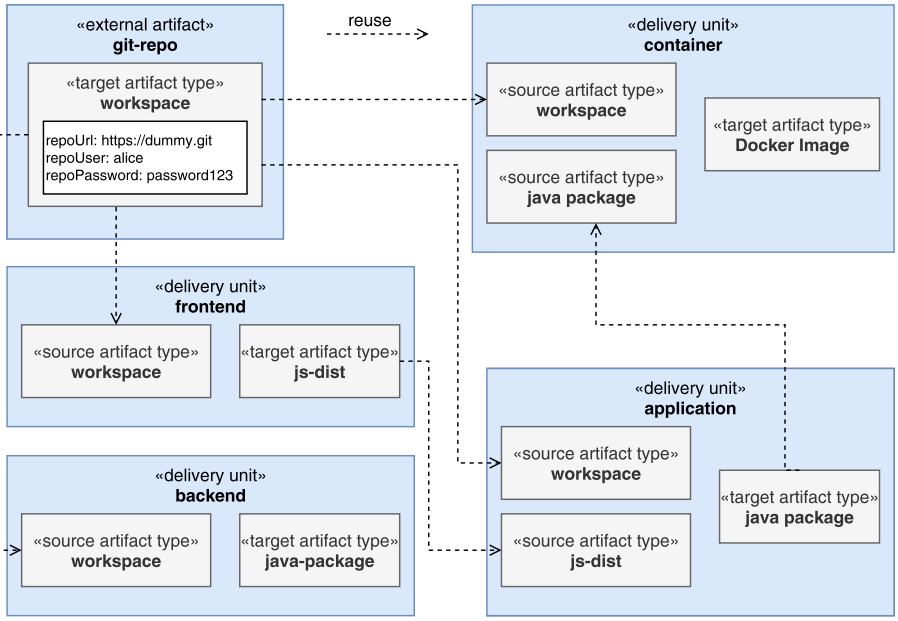

Fig. 2. Graphical visualization of an example D-AIM

process knowledge required, but then a delivery system would not know what to execute given the model.

Instead, we propose to shift the responsibility for providing the required process knowledge to the delivery system in form of a transformation activity knowledge base. It should store information about each transformation activity and their relation to each other in terms of the consumed and produced artifact types.

B. Transformation Activity Knowledge Base

*Definition 4 (**Transformation Activity Knowledge Base**):* A transformation activity knowledge base $KB^{t\text{-}act}$ stores tuples $(R^{in}, t\text{-}act, r_{out})$ with

- R^{in}: a set of artifact types $\subseteq \mathcal{P}(R)$
- $t\text{-}act$ transformation activity name
- r_{out} an artifact type $\in R$

which specifies that transformation activity $t\text{-}act$ takes artifacts of type R^{in} as input and produces an output artifact of type r_{out}

We can then reduce a D-AM by leaving out all transformation activities and including only the source and target artifact types in the models. The intermediate transformation activities can be derived using the knowledge base. Reuse relationships between artifact types are preserved, since they apply to source and target artifact types.

By doing this, we reduce the required process knowledge since we do not have activities for which the execution order needs to be known.

As only the source and target artifacts are modeled, we call this reduced DU-AM variant delivery unit artifact interface model.

C. Delivery Artifact Interface Models

*Definition 5 (**Delivery Unit Artifact Interface Model**):* A delivery unit artifact interface model (DU-AIM) of a delivery unit du_i is represented by a tuple $(R^{src}_{du}, R^{tar}_{du}, P)$ with

- $R^{src}_{du_i}$: set of required artifacts needed to create du_i
- $R^{tar}_{du_i}$: set of target artifacts of du_i
- P: a set of key-value parameters

Analogous to the D-AM, we define the delivery artifact interface model (D-AIM) containing only DU-AIMs.

In concrete delivery processes, information regarding the source of the first input artifacts needs to be specified. In general, we need to further add information about "external artifact" locations by providing parameters like credentials or URLs to those artifacts. This is exemplary shown in figure 2 where the "workspace" input artifact type contains repository location information.

Thus, we introduce the external artifact interface model to represent external artifacts as a subelement of a D-AIM.

*Definition 6 (**External Artifact Interface Model**):* An external artifact interface model (E-AIM) represents a single artifact along with associated parameters that can only be created by a specific transformation activity. Based on the specified artifact type, this artifact is already linked to the producing transformation activity.

For example, an artifact of type "workspace" can only be produced by a specific version control system activity that requires special parameters like credentials. Each D-AIM contains at least one E-AIM, since the existence of a "workspace" artifact is necessary in order to perform further activities. Similarly, parameters like paths to Dockerfiles or Maven POM-files can be assigned to DU-AIMs, because some activities can not work without specific parameters.

Outsourcing the "workspace" artifact from delivery units and instead providing them to delivery units from an E-AIM increases the reusability of artifacts. By doing this, each delivery unit doesn't have to include the same transformation activity that downloads the corresponding directory.

*Definition 7 (**Delivery Artifact Interface Model**):* A delivery artifact interface model (D-AIM) is represented by a tuple $(DU\text{-}AIM, E\text{-}AIM, \gamma)$ with

- $DU\text{-}AIM$: set of $DU\text{-}AIM_{du_i}$, with $1 \leq i \leq n$.
- $E\text{-}AIM$: set of $E\text{-}AIM_i$, with $1 \leq i \leq n$.
- γ: artifact reuse relation with $\gamma \subseteq (\cup R^{tar}_{du_i} \times \cup R^{tar}_{e_k}) \times \cup R^{src}_{du_j}$, with $(r_{tar}, r_{src}) \in \gamma$ if the provided artifact r_{tar} of a delivery unit du_i or of an external artifact e_k is reused as required artifact $r_{src} \in R^{src}_{du_j}$ of delivery unit du_j, with $i \neq j$ and concrete artifact instances of artifact types r^{src} and r^{tar} are identical.

The graphical visualization of an D-AIM for our running example is depicted in figure 2. It models the different delivery units for the architectural components, their corresponding important input/output artifact types and the reuse relationships.

Based on a given D-AIM, we can restore the complete D-AM model, by applying a backtracking technique on each contained DU-AIM independently to automatically define possible chains of transformation activities starting at the target and finishing at the source artifact types.

IV. Transformation Strategy

We designed the D-AIMs by starting with a classic delivery process model, modularizing it along architectural components and gradually removing process-related parts.

Based on section III, we can now create delivery process models that are mainly based on architectural knowledge (RQ1).

To execute these models with existing delivery systems, we need to transform these artifact-based model into activity-based models. We design a strategy with two transformation steps. In each a dedicated model-to-model transformation is performed. The overall transformation strategy is illustrated in figure 3.

The first step is called **Artifact-based Delivery Model Reconstruction** in which the given D-AIM is split into its individual DU-AIMs and E-AIMs. Each DU-AIM is transformed into a equivalent DU-AM by determining the missing transformation activities and intermediate artifact types. To this end, we utilize a backtracking and optimization algorithm which incorporates the activity specifications available in the transformation activity knowledge base.

The second step is the **Activity-based Delivery Model Derivation**, which transforms each DU-AM and E-AIM into separate activity-based process models. In order to merge these separate sequences of activities into a single process model, we connect the artifact references within each activity based on the reuse relations between artifact types.

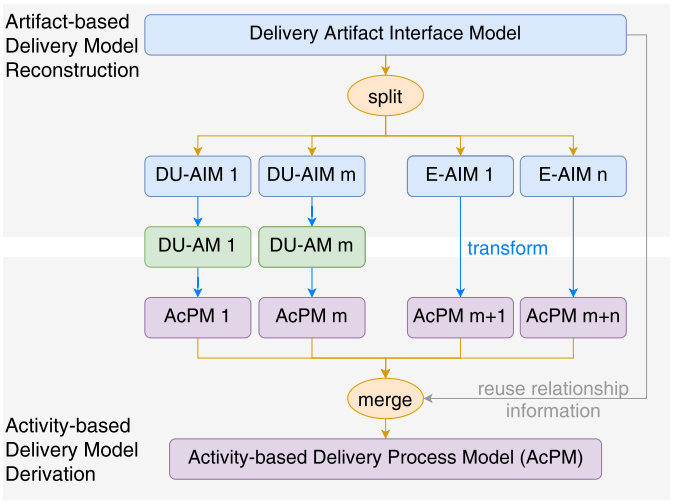

Fig. 3. Model Transformation Strategy

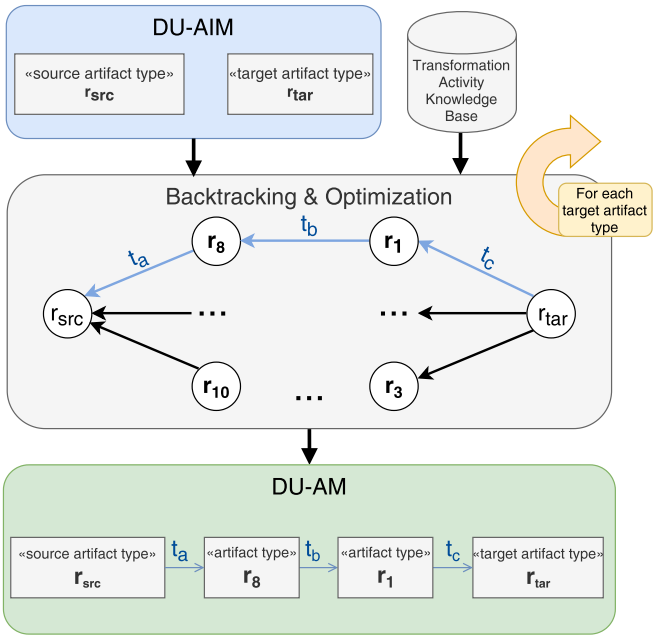

Fig. 4. Model Reconstruction: from DU-AIM to DU-AM

V. Artifact-based Delivery Model Reconstruction

The main techniques applied to realize the model-to-model transformation between DU-AIMs and DU-AMs are backtracking and optimization. Figure 4 visualizes the transformation inside the artifact-based model reconstruction step.

A. Backtracking

Given the source and target artifact type sets R^{src} and R^{tar} of a DU-AIM du, we need to find possible chains of transformation activities that transform artifacts with artifact types in R^{src} to artifact types in R^{tar}. In order to do this, we

apply the following approach on each $r_{tar} \in R^{tar}$. Since we can extract the input and output types of each transformation activity from $KB^{t\text{-}act}$, we start at the target type r_{tar} of the considered provided artifact and gradually append transformation activities with fitting output type. In doing so, we need to consider that some transformation activities consume multiple artifacts in order to produce a single one. There may also exist redundant transformation activities that consume and produce the same artifacts. This can occur for example when multiple build tools exist for the same technology.

We capture all possible chains of transformation activities for each target artifact type of a given DU-AIM in its own artifact dependency graph:

*Definition 8 (**Artifact Dependency Graph**):* An artifact dependency graph (ADG) is a directed acyclic graph $G = (N,\ E \subseteq N \times N, \alpha,\ n_{src},\ n_{tar})$ with:

- N: finite set of nodes representing artifact types.
- E: set of directed and labeled edges E $\subseteq N \times N$. Edges which have their origin in the same node can have an AND-relation and indicating that these artifact nodes belong together (e.g. when a transformation activity requires more than one artifact as input).
- $n_{src} \subseteq N$: set of nodes representing all source artifact types $r \in R^{src}_{du}$
- $n_{tar} \in N$: a node representing one target artifact type $r_{tar} \in R^{tar}_{du}$.

The graph is built as follows. First, we create all nodes contained in n_{src} and the node n_{tar}. Since n_{tar} represents an artifact type $r_{tar} \in R^{tar}$, we extract all tuples $(R^{in}, t\text{-}act, r_{out})$ from $KB^{t\text{-}act}$ with $r_{tar} = r_{out}$ as output artifact type. For each selected tuple, nodes for all its input artifact types R^{in} are added to the graph and connected AND-edges labeled with the activity name are created from node n_{tar} to the newly added nodes. In the next step, one of the newly added nodes is examined for fitting transformation activities. Analogous to breadth first search, this backtracking procedure is repeated until nodes in n_{src} are reached by an edge or a threshold is exceeded. An example ADG is shown in figure 5. The depicted activities are specified in the JARVIS registry as follows. The compile-activity consumes a "workspace" artifact and produces a "java-bytecode" artifact. "Workspace" and "java-bytecode" are required for the assemble-activity which produces a "Java package" artifact.

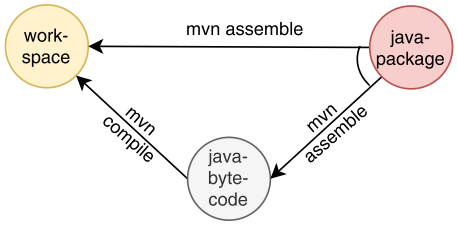

Fig. 5. Example ADG: From n_{tar} (red) to n_{src} (yellow)

Pruning

After building the graph, there may exist paths starting at n_{tar} that end in nodes that are not contained in n_{src}. These leaf nodes can be recursively removed until all leaf nodes are elements of n_{src}.

A further problem while building the ADG are transformation activities with intersecting input types. This occurs when an artifact can be produced by several activities sharing common input types. The graph can not reflect such a relation. In such a case, we decide which activity to keep in the graph. First, we look at the number of source artifact types that would be covered and select that activity with the most covered source types. A second decision criterion is the number of new nodes that each activity introduces into the graph. A lower number is favored.

B. Optimization

An ADG captures all possible transformation activity sequences that produce one $r_{tar} \in$ R^{tar} for a given DU-AIM. Because we assume that the shortest transformation activity sequence is favorable, we apply an optimization procedure to find the shortest sequence within the graph. Hereby, the solution is not always a single path, since paths starting at the node n_{tar} can fork in case of transformation activities with multiple input types.

We divide the optimization problem into two sub-problems.

- *P1:* Find the minimal set of connected paths from n_{tar} to nodes of the set n_{src}.
- *P2:* Since edges on these paths can be labeled with multiple activity names in case of redundant transformation activities, select one label per edge. While doing so, ensure a homogeneous use of technology by e.g. using always the same build tool when using Java technology

We use integer linear programming (ILP) to solve theses problems. The corresponding ILP models and their explanations are provided in the following.

1) Optimization P1:

Variables: The variable $x_{i,j}$ refers to an edge from node i to node j with $i, j \in 0, ..., n$ for an ADG containing n artifact nodes. In order to clearly assign nodes to indices, each node is mapped to an integer number. Hereby, the node n_{tar} is mapped to zero. $x_{i,j}$ indicates whether an edge is part of the shortest chain of transformation activities. Such an edge is called *marked*, otherwise unmarked.

Objective function: Since we are interested in the shortest activity chain, we want to minimize the number of marked edges.

Constraints: The mapping of nodes to integers introduces further possible edges that are not part of the graph. These are forced to be unmarked in the first constraint. The next two constraints guarantee that paths of marked edges have their origin at node n_{tar}, whereas each $n \in n_{src}$ has at least one incoming marked edge. Next, two constraints lead to the formation of paths. If a node, that is not in n_{src}, has an incoming marked edge, then it must also have an outgoing

Fig. 6. ILP for optimization problem P1

$$\text{minimize} \quad \sum_{i=0}^{|N|-1} \sum_{j=0}^{|N|-1} x_{i,j}$$

$$\text{subject to} \quad x_{i,j} = 0, \qquad ,\forall (i,j) \notin E$$

$$\sum_{\substack{j=0 \\ i \neq j}}^{|N|-1} x_{0,j} \geq 1$$

$$\sum_{\substack{i=0 \\ i \neq r}}^{|N|-1} x_{i,n} \geq 1 \qquad ,\forall n \in n_{src}$$

$$x_{i,j} \leq \sum_{\substack{k=0 \\ j \neq k}}^{|N|-1} x_{j,k} \qquad ,\forall (i,j) \in E : i \neq j \wedge j \notin n_{src}$$

$$x_{i,j} \leq \sum_{\substack{k=0 \\ i \neq k}}^{|A|-1} x_{k,i} \qquad ,\forall (i,j) \in E, i \neq j \wedge i \neq 0$$

$$x_{i,j} \leq x_{i,k} \qquad ,\forall (i,j) \in E : label(i,j) = label(i,k)$$

marked edge. The last constraint realizes the coupling between edges by an AND-relation. These can only be marked together or not at all.

Optimization result: After applying a solver on this ILP we receive the optimal solution x* which corresponds to the minimal set of shortest paths in the ADG that fulfill the above mentioned constraints. A closer look to the indices of our variable shows which edges are marked and therefore relevant for the transformation process. Besides that, after mapping the indices back to nodes, we receive all intermediate artifact types.

We define $R^{inter}_{r_{tar}}$ as the set containing all intermediate artifact types created when producing the concrete artifact behind r_{tar} using the required artifacts represented by R^{src}.

However, some marked edges may contain multiple labels in case of redundant transformation activities. We store each of those label candidate lists in another list C. In order to provide a unique transformation activity between all artifacts we apply a second optimization procedure.

2) Optimization P2: Each activity in the JARVIS activity registry belongs to a service which indicates the technology. Based on that, we create a list of all services that are included in the registry. For simplicity reasons we provide explanations instead of a formal ILP.

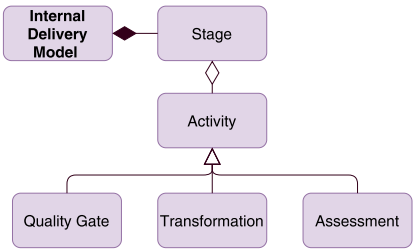

Fig. 7. JARVIS internal delivery model

Objective function: We want to minimize the overall number of services included in the transformation process from n_{src} to n_{tar}.

Constraints: For each candidate list $C' \in C$ we state that:

- if C' contains only one transformation activity label, then this label is kept and the corresponding service selected
- otherwise, only one activity label is allowed and other labels should be discarded
- besides that, if we select a transformation activity, then the corresponding service is also included.

A solver for the corresponding ILP returns a minimal and unique set $T_{r_{tar}}$ of transformation activities to produce an artifact of type r_{tar}. After obtaining all sets of intermediate artifact types $R^{inter}_{r_i}$ and all transformation activities set T_{r_i} for all source artifact types $r_i \in R^{src}$ of a delivery unit du, we create the DU-AM that corresponds to the given DU-AIM as follows: (R_{du} = $R^{src}_{du} \cup \mathrm{R}^{tar}_{du} \cup (\bigcup_i R^{inter}_{r_i})$, R^{src}_{du}, R^{tar}_{du}, T_{du} = $\bigcup_i T_{r_i}$) for $0 \leq i \leq |\mathrm{R}^{tar}_{du}|$.

VI. Activty-based Delivery Model Derivation

Next, we need to define a mapping procedure that automatically transforms a set of DU-AMs to a single process model to solve RQ2. In order to do this, we will create a delivery process model compatible with the delivery system JARVIS as it already includes a transformation activity knowledge base.

A. JARVIS

JARVIS is a continuous delivery platform that aims at reducing the technical and process knowledge required for implementing delivery processes [2].

In order to allow developers to specify delivery process models in their preferred modeling language, JARVIS differentiates between external and internal delivery process models. JARVIS currently accepts different activity-based external delivery process models, e.g. its own DSL but also BPMN, and transforms these to corresponding internal delivery models (IDM) automatically. Thus, for each kind of external delivery model, JARVIS offers transformation capabilities which perform the needed model-to-model transformation.

JARVIS executes internal delivery process models (see figure 7), which define a CD pipeline consisting of ordered stages, each defining its specific activities. The stages are executed sequentially. A stage cannot contain multiple occurrences of the same activity or artifact type due to ambiguity in the chaining of activities.

An activity is defined by input and output artifact types and a encapsulated execution logic, e.g. a "compile"-activity consumes a project directory and produces a set of java classes by executing the "mvn compile" internally. An activity registry stores all information about available transformation activities. This registry represents the transformation activity knowledge base needed for our approach.

JARVIS differentiates between three different types of activities:

- A transformation takes one or more artifacts as an input and produces an artifact of a different artifact type
- An assessment takes one or more artifacts as an input and produces a test report
- A quality gate takes an artifact and a test report as an input and promotes or rejects the artifact based on the report

In order to model a delivery process, the user needs to define for all stages all activities which should be contained in the respective stage. The specification of an activity consists of the name of an activity and parameters. It can also contain references to artifacts of other activities to express the activity dependencies explicitly. References can be omitted. The complete delivery process model is generated by so called process planners which analyze the input and output artifact types of all activities, concatenates them accordingly and optimize for execution. Multiple process planners can be combined and applied in a sequential manner. This way, different optimization criteria can be considered.

In general terms, a D-AM describes how artifacts of all contained delivery unit artifact models (DU-AM) are transformed within a delivery process. It depicts how the artifacts evolve by the usage of transformation functions and finally are composed to the provided artifacts. Therefore, although this delivery model type doesn't name the concrete artifacts, it can serve as an external input model for a delivery process in JARVIS. Note that JARVIS internal delivery models list activities and no concrete artifacts. As JARVIS needs to know all transformation activities to perform a delivery process, they have to be determined based on the artifact types specified in an D-AM. In the following, we describe how a D-AM can be transformed to an internal delivery model (IDM) in JARVIS.

B. Model Derivation with JARVIS

The overall transformation strategy is illustrated in figure 3. For model derivation we need to map each DU-AM to an activity-based process model of JARVIS. For this purpose, we represent each DU-AM as a JARVIS stage (see pseudo-algorithm 1). Furthermore, each transformation function is modeled as a JARVIS transformation activity and added to the stage. To this end, the JARVIS activity specification is retrieved from the JARVIS activity registry (KB). Each E-AIM, which consists only of one target artifact is directly transformed to a JARVIS activity. Based on defined rules which depend on the specified artifact type within, a JARVIS activity is constructed. Passed parameters in DU-AMs and E-AIMs are matched with parameters of the same name in the activity specification.

After this transformation step, a separate JARVIS IDM exists for each DU-AM and E-AIM. To merge those IDMs to create a complete JARVIS IDM, all artifact reuse relations within the original D-AIM are captured in a hash map data structure. Hereby, the reuse relations describe dependencies between DU-AIMs and E-AIMs and are based on architectural information. Based on these dependencies, input artifact references are added to the activities in the corresponding stages.

Since JARVIS executes stages sequentially, stages need to be ordered. To this end, stage dependencies are captured in a directed acyclic graph and topological sort is applied so that the stages are sorted by their derived dependencies. The transformation result can then be executed by JARVIS as a single delivery process.

Algorithm 1: DU-AIM i to JARVIS IDM i Transformation

```
stages ← new list;
jarvisModel ← new list;
foreach DU-AM du in D-AM do
    stage ← new JARVIS stage;
    stage.name ← du.name;
    foreach TransformationFct tf in
     du.transformationFcts do
        activity ← KB.getActivitySpecification(tf);
        activity ← matchParams(activity,du.params);
        stage.insert(activity);
    end
    jarvisModel.insert(stage);
end
return jarvisModel;
```

VII. Case Study

Using artifact-based models to describe delivery processes is a break of modeling conventions. Nevertheless, we assume that our approach makes it easier to model complex delivery processes. In order to check our assumption and investigate the last two research questions, we performed a small scale case study with the following objectives:

- O1: How do study participants evaluate the ease of artefact-based modeling?
- O2: Is the given architectural information enough for creating complete models?

Be defining O1, we aim to answer RQ3, whereas the insights provided by O2 will help us regarding RQ4.

A. Participants

Eight employees of an industry partner participated on a voluntary basis. The group consisted of 4 junior- and 4 senior-developers. The senior developers are highly experienced in DevOps topics and technologies.

B. Methodology

The following steps were performed out with each participant separately. First of all, the participant was informed about the objectives of the study and how data will be collected and used. Next, the participant received an introduction to modeling with D-AIMs and was presented some graphical modeling examples. After that, the participant was asked to model the delivery process of a company project graphically and explain the resulting model. Hereby, all participants were given architectural information as a common basis. The highly-experienced group was already familiar with the delivery process behind the project, in contrary to the low-experienced group. Finally, the participant had to answer some structured interview questions.

We collected and analyzed the following data: created models by participants, video recordings of modeling process and answers/feedback given during the interview.

C. Results

When creating a D-AIM for a familiar delivery process, two participants created a complete and correct model, while the remaining participants failed to add the input artifact type "workspace", which corresponds to all files in the version control system of the project, to each DU-AIM. It was only specified in the first one.

With regard to the difficulty of creating D-AIMs, it was generally perceived as easy except for identifying the delivery units in the first step. It received a moderate rating.

Observing the participants while modelling revealed serious differences in the thought process. Especially the highly-experienced participants started listing all artifact types that are produced along the delivery process and grouped them to delivery units afterwards, based on architectural components. This corresponds to our approach in reverse, which is starting with identifying delivery units and then specifying associated artifact types. Likewise, this indicates process-based thinking.

Moreover, the highly trained group does not summarize multiple artifact types in one model and omit the intermediate artifact types. Instead, each DU-AIM implicitly contains exactly one transformation activity resulting in a large and finely grained model. Hereby, they enforce the occurrence of certain transformation activities and bypass the backtracking and optimization procedure. This shows that our approach allows to implicitly model an activity-based delivery model.

In conclusion, the highly-experienced group can not let go of the process view while modeling. This gives them a feeling of more control over the delivery process. These developers already have profound technical and process-related knowledge, therefore it is not a challenge for them to incorporate the knowledge into a delivery model.

This is also supported by the interview answers. The low-experienced group welcomes the artifact-based way of modeling delivery processes and experiences it as easier. This group lacked in-depth technical and process-related knowledge and therefore clearly sees a benefit in using a model that does not require that knowledge. Despite their lack of knowledge they were able to construct almost correct models (as some "workspace" artifact types were missing) solely based on the given architectural information.

Besides that, the whole approach is based on the existence of an activity registry. Participants see a hidden complexity in this, because the activities along with their deposited commands need to be designed before modeling.

D. Threats to Validity

Threats to *External Validity* concern the ability to generalize the results of our study. For instance, in our study, the *sample size* is quite small. A higher number of participants would be beneficial. As we rely on volunteers to invest their working time and provide an open mindset, the pool of potential participants is limited. For this study we made sure that pre-existing knowledge was not required.

Threats to *Internal Validity* relates to the trustworthiness of identified relationships. The participants had to be trained on our modeling approach. The design of the training itself can influence the capability to apply the approach, We mitigate this by using a known and quite simple project. Furthermore, the training was carried out in a short time frame which could also influence the acceptance negatively. In addition the potential bias of the authors conducting the study can influence the results.

VIII. Related Work

Applying an artifact-centric view to business processes has already been implemented in [11]. The authors constructed a business process model which depicts how artifacts evolve along the business process. Hereby artifacts are grouped according to their function and may have dependencies to artifacts of other functional groups. As a result, business experts were able to maintain the process model without help of IT experts, as they are already familiar with the existing artifacts.

Furthermore, an artifact-centric process model is easier to modify as changing the order of artifacts requires significantly less effort than rewriting a sequence of activities [12]. Moving the activities out of process models allows for more flexibility, as their specification can be modified without changing the process model itself.

Models and DSLs for delivery processes has been introduced by various system vendors or projects, e.g. Jenkins, a popular CI/CD platform, uses imperative and declarative DSLs for modeling delivery processes [13]. Even BPMN can be adopted as shown by Ahmad and Thiel [14]. To the best of our knowledge, all available models incorporate an activity-centric approach for delivery processes. The capability of creating delivery models automatically has not been introduced in many systems. Gitlab integrated AutoDevOps which creates a delivery process based on a opinionated template process [15], in Concourse CI the execution flow of jobs is computed and scheduled based on modeled constraints on jobs and resources [16].

Research on CI, CD and DevOps has focused on the benefits and obstacles in adopting these emerging practices. Laukkanen et al. discover one obstacle which is important to the approach in this paper: the increasing complexity of the underlying delivery system, the delivery models and delivery processes caused by the software architecture [6]. A study conducted by Debroy et al. suggests a tight coupling between architecture and delivery process, as architectural changes lead to incompatible CD pipelines [7].

The comparison of different delivery process designs by Zhang et al. indicate that less experienced developers tend to favor simplified models with lower complexity and maintenance effort [17]. Chen argues in his paper that to adopt CD successfully, it requires a dedicated team to develop, maintain and operate the delivery systems [18].

Delivery process models and their textual representation can be classified as a special kind of Infrastructure as Code (IaC). Duvall and Olsen suggested a catalogue of CI Anti-Patterns [19], which is the basis for the automated detection tool of Vassallo et al. [20]. The corresponding empirical study suggests that inexperienced developers can benefit from tool support. Gallaba and McIntosch studied the misuse of CD features [21], their findings of common anti-patterns in CI specifications indicate that developers lack the required knowledge to construct and maintain these specifications. The suggested solution includes additional tooling for analyzing and fixing CI specifications. The authors emphasize the emerging security risks. Rahman et. al discovered various security risks embedded in IaC [22], [23]. Paule et al. found indications, that the lack of knowledge and awareness of security issues lead to vulnerabilities in delivery processes [24].

A common denominator of current research is that developers need additional support when constructing and managing delivery models and processes. Often additional tools are suggested to inform developers of problems, which are mitigating the effects but do not tackle the underlying structural problem of lacking knowledge and training. Our approach instead reduces the needed knowledge to architectural information about the software project.

IX. Conclusion

To answer RQ1 we take a closer look at the modeling elements. To create an artifact-based model a developer needs to identify the existing architectural components, the respective input/output artifact types and the reuse relationships between them. These information can be directly extracted from architecture models (component and deployment diagrams), available documentation can be provided by architects and developers. The D-AIMs represent these architectural components and their relations.

With the presented automatic artifact-based model to activity-based model transformation strategy we can answer research question RQ2. Regarding the preconditions for our transformation strategy we see the transformation activity knowledge base as a crucial component. It incorporates required technical and process knowledge, which developers do not have to provide in the delivery model. Based on the knowledge base, the backtracking and optimization technique gains the necessary knowledge by itself.

Regarding RQ3, we conclude that when developers lack process knowledge, they perceive it as easy to model based on solely architectural information, since they don't need to invest time to improve their process knowledge. On the other hand, developers with high process knowledge seem to have problems with 1. trusting a system to construct the process on itself and 2. ignoring the process knowledge while modeling.

With respect to RQ4, the given architectural information was indeed sufficient as all participant delivered an almost correct model. A common mistake was to leave out a "workspace" input artifact type, which indicates an error in understanding rather than insufficient information.

X. Summary and future work

In this paper we present a novel artifact-based modeling approach for delivery processes. We abstract the artifacts that are used during the process by introducing the notion of artifact types. Thus, instead of modeling the activities of the process directly, we model architectural components and their required and target artifact types. As a result, less technical and process-related knowledge is needed for modeling. The overall modeling complexity for complex delivery processes is reduced.

In order to deduce an executable activity-based model from our artifact-based model, we define an automatic model-to-model transformation strategy, which consists of multiple steps and incrementally builds a activity-based delivery model for each architectural component.

To gain more insight into the applicability and acceptance of our new approach, a qualitative study in an industrial context was conducted. Higher-experienced developers prefer a rather activity-based approach due to more control over the resulting process. In contrary, low-experienced developers benefit from the reduced required prior knowledge.

Regarding future research this paper provides multiple opportunities. First, additional studies need to be performed to further validate our approach and collecting more insights how developers interact with delivery models and processes. Second, different optimization techniques and criteria can be evaluated for the automated construction of delivery models.

In this paper, we focus on the transformation aspects of delivery processes. In CD and DevOps quality is the most important aspect besides automation. Modeling quality activities for artifacts in a similar way like for transformations can be integrated in our approach. These activities can be assigned to artifacts automatically. JARVIS already offers the required concepts and knowledge about these assessment activities. Quality metrics can be enforced by automatically introducing quality gates in the delivery process.

To sum up, our approach has the potential to create a stakeholder driven perspective on CD, involve new interested parties into the delivery process and thus provide a more holistic approach to Continuous Delivery in general.

REFERENCES

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

[2] A. Steffens, H. Lichter, and J. S. Doering, "Designing a next-generation continuous software delivery system: Concepts and architecture," in *2018 IEEE/ACM 4th International Workshop on Rapid Continuous Software Engineering (RCoSE)*, 05 2018, pp. 1–7.

[3] Q. Tu and M. W. Godfrey, "The build-time software architecture view," *IEEE International Conference on Software Maintenance, ICSM*, pp. 398–407, 2001.

[4] C. Hofmeister, R. Nord, and D. Soni, *Applied Software Architecture*, 1st ed. Addison-Wesley Professional, 2009.

[5] L. Bass, "The software architect and devops," *IEEE Software*, vol. 35, no. 1, pp. 8–10, 2018.

[6] E. Laukkanen, J. Itkonen, and C. Lassenius, "Problems, causes and solutions when adopting continuous deliveryâa systematic literature review," *Information and Software Technology*, vol. 82, pp. 55–79, 2017.

[7] V. Debroy and S. Miller, "Overcoming challenges with continuous integration and deployment pipelines: An experience report from a small company," *IEEE Software*, vol. 37, no. 3, pp. 21–29, 2020.

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

[9] M. Soni, "End to end automation on cloud with build pipeline: The case for devops in insurance industry, continuous integration, continuous testing, and continuous delivery," in *2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, 2015, pp. 85–89.

[10] L. Zhu, L. Bass, and G. Champlin-Scharff, "Devops and its practices," *IEEE Software*, vol. 33, no. 3, pp. 32–34, 2016.

[11] N. S. Caswell, "Business artifacts : An approach to operational," *Information Systems*, vol. 42, no. 3, 2003.

[12] R. Hull, "Artifact-centric business process models: Brief survey of research results and challenges," in *On the Move to Meaningful Internet Systems: OTM 2008*, R. Meersman and Z. Tari, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1152–1163.

[13] V. Armenise, "Continuous delivery with jenkins: Jenkins solutions to implement continuous delivery," in *2015 IEEE/ACM 3rd International Workshop on Release Engineering*. IEEE, 2015, pp. 24–27.

[14] M. Ahmad and L.-R. Thiel, "Devops meets bpm," *JavaSpektrum*, vol. 6, 2014.

[15] "Gitlab," https://docs.gitlab.com/ee/topics/autodevops/, accessed: 1.04.2021.

[16] "Concourse," https://concourse-ci.org/scheduler.html/, accessed: 1.04.2021.

[17] Y. Zhang, B. Vasilescu, H. Wang, and V. Filkov, "One size does not fit all: An empirical study of containerized continuous deployment workflows," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 295–306. [Online]. Available: https://doi.org/10.1145/3236024.3236033

[18] L. Chen, "Continuous delivery: Overcoming adoption challenges," *Journal of Systems and Software*, vol. 128, pp. 72–86, 2017.

[19] P. Duvall and M. Olsen, "Continuous delivery: Patterns and anti-patterns in the software lifecycle," https://dzone.com/refcardz/continuous-delivery-patterns, 2018, accessed: 20.04.2021.

[20] C. Vassallo, S. Proksch, H. C. Gall, and M. Di Penta, "Automated reporting of anti-patterns and decay in continuous integration," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 105–115.

[21] K. Gallaba and S. McIntosh, "Use and misuse of continuous integration features: An empirical study of projects that (mis) use travis ci," *IEEE Transactions on Software Engineering*, vol. 46, no. 1, pp. 33–50, 2018.

[22] A. Rahman and L. Williams, "Characterizing defective configuration scripts used for continuous deployment," in *2018 IEEE 11th International conference on software testing, verification and validation (ICST)*. IEEE, 2018, pp. 34–45.

[23] A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 164–175.

[24] C. Paule, T. F. Düllmann, and A. Van Hoorn, "Vulnerabilities in continuous delivery pipelines? a case study," in *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2019, pp. 102–108.