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

— Online Appendix[†] —

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 ways for software deployment in the cloud and in service-based computing. Deployment pipelines are a core artefact in modern DevOps and CD practices. So far the design of deployment pipelines is largely discussed informally, usually with a specific application focus and using a specific combination of tools. While most such informal discussions include detailed descriptions of the architectures in which the deployment pipeline operates, such as target environments or tool integration architectures, those aspects are usually not even informally modelled. For these reasons sources describing DevOps and CD practices are often inconsistent or incomplete, and a generic, complete, tool-agnostic, and application-independent treatment of deployment pipeline design and architecture is missing today. These issues impede building generic tools that work across different technologies and applications, and they also hinder the tool-agnostic and application-independent abstraction in the understanding of pipelines models by human designers. 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 today's cloud- and service-based computing is that the frequency of change required in those systems steadily increases, and continuous releases are becoming the expected norm rather than the exception. One main trend in this direction is the DevOps movement [1], [2] which aims to tear down the barriers between software development, deployment, and operations, and thus allowing software changes to be delivered to the customers more quickly. One core aspect of DevOps that specifically targets decreasing release cycle times is Continuous Delivery (CD) [3]. CD is primarily about automating the deployment and release engineering process but also has a strong focus on a variety of development, deployment, and quality assurance practices [4], [5]. DevOps and CD are becoming the de-facto standard ways for software deployment for cloud (micro-)services.

Deployment pipelines [3] are the centrepiece of most of today's DevOps and CD practices. So far the design of deployment pipelines is largely discussed informally, books [3], [4], practitioner blogs [6], [7], or in the documentations or developer guidelines of tools or cloud-based and (micro-)service-based applications. Whereas it might seem at the first glance that an informal model of a deployment pipeline (often modelled in a style similar to a UML activity diagram) gives a good overview of all deployment structures, experience in the field and a detailed study of the informal descriptions of such models as used by practitioners immediately reveals that almost all realistic deployment pipelines require complex architectures of various tools and components, including deployment target environments, tool integration architectures, and the environments in which the tools run (see e.g. [8], [3]). The complexity of this overall architecture is usually not covered in the informal models, nor are all aspects of the deployment pipeline itself. Consequently, the informal models and texts produced by practitioners for describing DevOps and CD practices are often inconsistent or incomplete. In addition, a generic, complete, tool-agnostic, and application-independent treatment of deployment pipeline design and architecture is missing today.

For getting a rough overview of a single deployment pipeline using a particular combination of tools for a particular application, this is usually not a problem. However, there are many scenarios where this level of understanding is not enough. Firstly, as most deployment pipelines operate on a complex combination of various independent tools and environments, this impedes building generic tools operating

[†] This is a long version, provided as an online appendix, of a paper published in 2019 at the IEEE International Conference on Services Computing (SCC 2019). Please cite this work as follows: U. Zdun, E. Ntentos, K. Plakidas, A. El Malki, D. Schall, F. Li. "On the Design and Architecture of Deployment Pipelines in Cloud and Service-Based Computing – A Model-Based Qualitative Study" in *2019 IEEE International Conference on Services Computing (SCC)*, 2019

on the whole pipeline, such as architecture consistency checkers or static analysis tools. Common abstractions, and consequently interfaces, are missing, meaning that such tools would need to be specially tailored for each and every combination of tooling, environments, and application domains. Secondly, the missing common abstractions also hinder understanding by human designers beyond a single set of technologies, environments, and applications. This is problematic for designing agreed-upon abstractions in the community across different tools, environments, and application foci. More specifically, some software designers face the need to understand practices across different projects; consider for instance an architect of a microservice-based software ecosystem [9] facing the need to understand and constrain possibly hundreds of different deployment pipelines and architectures.

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 have based our study on the model-based qualitative research method described in [10], which uses such documented practitioner sources as rather unbiased knowledge sources and systematically codes them through established coding and constant comparison methods [11], combined with precise software modelling, in order to develop a rigorously specified software model of established practices and their links. This paper aims to study the following resulting research questions:

- **RQ1** What are recurring established practices for designing deployment pipeline structures, how are they linked, and how can structures and links be precisely modelled?
- **RQ2** What are the environments relevant in deployment pipelines, how are they linked between themselves and with the pipelines, and how can the environments and links be precisely modelled?
- **RQ3** What are the architectural elements relevant for building a deployment pipeline infrastructure (such as deployment tools, build scripts, or Continuous Integration servers), how are they linked between themselves and with the pipelines, and how can the architecture elements and links be precisely modelled?

This paper makes three major contributions: Our result is a precisely specified model of deployment pipeline structures, along with models of the associated deployment environments and infrastructure architecture. In each of the parts of the model we also precisely define the links between the model elements, as well as consistent links between the different views. For each of the 25 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 increase the precision of the modelling compared to the informally modelled pipelines in the original sources by a total average improvement of 134.72%.

The remainder of this paper is organized as follows: In Section II we compare to the related work. Section III explains the research method we have applied in our study. Then Section IV contributes a precise specification of the CD pipeline model created in this study. Next, we briefly explain in Section V the tools we used for generating our models and show one model instance as an illustrative example of the generated output of our tool. Section VI contains a preliminary evaluation of our work, Section VII discusses threats to validity of our study, and finally in Section VIII we conclude this study. In addition, an Appendix contains the modelled pipelines derived from the sources examined.

II. Related Work

As pointed out above, informal descriptions of deployment pipelines and associated architectures dominate the literature today (cf. [8], [3], [6], [7]). These informal models, although commonly used to describe deployment pipelines, usually fail to fully cover either the complexity of these designs and architectures or all aspects of the deployment pipeline itself. As a result, the informal descriptions of DevOps and CD practices produced by practitioners are often inconsistent or incomplete. While many scientific works use and improve deployment pipelines (cf. [12], [13], [14], [15]), and first studies on deployment practices in organizations have emerged [16], [17], [18], a generic, but precise and consistent, tool-agnostic and application-independent treatment of deployment architectures is still missing.

Our study aims to improve this situation and provide the first systematic and precise specification approach for DevOps architectures, laying the foundations for automated quality control of the design of deployment pipelines and the associated environments and architectures. This would enable checking, for example, whether a pipeline design performs too few or too manual quality controls, is missing important steps (like forgetting to model a commit trigger) or 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 (e.g. running manual exploratory tests at the same time or even before automatic acceptance tests). So far, such design issues have been identified in the literature as red flags [3], [4], but their automatic detection is not possible as it requires precise models of all elements of the deployment pipeline and its associated environments and architectures; it is the goal of this study to establish such a precise modelling foundation.

In addition, many sources point for substantial alteration of the architecture of the target systems for supporting rapid releases [14], [17], [18]. For instance, different ways to decompose a system into microservices are discussed as options to improve the releasability of architectures [14],

[12], [19]. While the microservice decomposition itself is studied in the scientific literature extensively (see e.g. [12], [20], [21]), the associated deployment pipelines and architectures have not yet been studied in a systematic way. Another example is the move from a consistent data exchange model to eventual consistency models to improve releasability [22], [12], [23]. Again, the existing studies focus on consistency in microservice data exchange, not e.g. on how the deployment pipeline or environment architectures need to change to cope with eventual consistency. Our study aims to provide the ground work needed to perform such studies, by enabling formal reasoning, validation, and verification through a precise and consistent modelling foundation for the CD parts. Again, a precise modelling foundation is a prerequisite, e.g. to produce static analysis tools that can check whether the necessary release infrastructure for a microservice decomposition or eventual consistency is in place.

III. Research Method

This paper aims to systematically study the established practices in the field of designing or architecting deployment pipelines. We follow the model-based qualitative research method described in [10]. It is based on the established Grounded Theory (GT) [11] qualitative research method, in combination with methods for studying established practices like pattern mining (see e.g. [24]) and their combination with GT [25]. The method uses descriptions of established practices from the authors' own experiences as a starting point to search for a limited number of well-fitting, technically detailed sources from the so-called grey literature (i.e., practitioner reports, system documentations, practitioner blogs, etc.). These sources are then used as unbiased descriptions of established practices in the further analysis (in contrast to sources like interviews as used in classic GT). In contrast to pattern mining methods, the method does not aim to find all the forces, consequences, and detailed solution guidelines of the observed practices, but rather focuses on their identification and on their detailed relations in terms of formal modelling. Like GT, the method studies each knowledge source in depth. It also follows a similar coding process, as well as a constant comparison procedure to derive a model. In contrast to classical GT, the research begins with an initial research question, as in Charmaz's constructivist GT [26]. Whereas GT typically uses textual analysis, the method uses textual codes only initially and then transfers them into formal software models (hence it is model-based).

The knowledge-mining procedure is applied in many iterations. That is, we searched for one or a few new knowledge sources, applied open and axial coding [11] to identify candidate categories, and continuously compared the new codes with the model designed so far. We improved this model incrementally. A crucial question in qualitative methods is when to stop this process. Theoretical saturation [11] has attained widespread acceptance 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. While this is a rather conservative operationalisation of theoretical saturation (i.e., most qualitative research saturates with a much lower number of knowledge sources that add nothing new), our study converged already after 10 knowledge sources in the sense that no substantial new formal model elements were created. Minor changes to the model were introduced until we reached 18 knowledge sources. The knowledge sources included in the study are summarized in Table I). Our search for knowledge sources was based on our own experience, e.g., DevOps tools and environments we have access to, worked with, or studied before. We also used major search engines (e.g., Google, Bing) and topic portals (e.g., InfoQ) to find more sources.

We deliberately included descriptions that ranged from the very simple (e.g. BROIND), with a few stages and no tooling, to the more complex, with over 10 stages and detailed tooling (cf. Table III). Pipelines with different levels of automation (manual to fully automatic) and generality (generic to application-specific) were examined. Generally the sources either focused on describing generic CD pipeline practices, or more on the specific tooling required, often to the point of being a step-by-step user guides for using particular CD pipeline solutions. Table III summarizes the sizes of the source model in different element categories.

IV. 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 the code for the meta-model, our models of the individual pipelines, vector-based visualizations, and documentation on the interpretation and modelling of the original sources available in an online repository[1] for the purpose of verification and reproducibility. An example can be found in Section V. In addition, for ease of reference, visualizations of the pipeline models we derived after examining the sources listed in Table I are reproduced in the

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

Table I
KNOWLEDGE SOURCES INCLUDED IN THE STUDY

Name	Description	Reference
ADOBE	User guide for the Adobe Cloud Manager's CI/CD framework	https://bit.ly/2T16cfA, https://bit.ly/2ThrepA
AWSGO	Description of how to perform CI and CD for applications written in Go on AWS	https://bit.ly/2T16e7c
AZURE	Quickstart user guide for Azure DevOps pipelines	https://bit.ly/2To9zJI, https://bit.ly/2Tn02lZ
BITBAR	Basics of Mobile Continuous Integration workflow	https://bit.ly/2EAquUd
BROIND	High-level pipeline recommended as a starting point for moving to DevOps	https://bit.ly/2RtZqcH
CCAUT	Description of fully automated pipeline variant using ThoughWorks cruise control tool	https://bit.ly/2t20JWd
CCMAN	Descr. of semi-automated pipeline variant with some manual approvals using ThoughWorks cruise control tool	https://bit.ly/2t20JWd
FACEB	Description of the continuous deployment pipeline practices used by Facebook in 2013	[8]
GOOGLE	Tutorial for creating a CD pipeline on the Google Cloud platform	https://bit.ly/2UbYLyy
HEROKU	User guide on setting up a Heroku CI pipeline	https://bit.ly/2SvJRBM, https://bit.ly/2Suv2PF
HFCOMP	Description of a variant of the generic pipeline suggested in [3] with a separate pipeline per component of the application and in addition one for the whole application	[3], page 366 ff.
HFGEN	Description of the generic pipeline suggested in [3]	[3], page 111 ff.
IBM	User guide on working with an IBM Cloud Continuous Delivery pipelines	https://bit.ly/2DLhGL1
IOT	Description of a DevOps workflow for Internet of Things applications	https://bit.ly/2TluHA8
JENKINS	A typical deployment pipeline realized in the Jenkins tool	https://bit.ly/2VhtGd4
MOZILLA	Overview of Continuous Integration practices at Mozilla	https://bit.ly/2IFI79q
NETFLIX	Description of the continuous deployment pipeline practices used by Netflix in 2016	https://bit.ly/2tX3D1S
OPSHIFT	Description of foundational concepts related to building a CI/CD pipeline in OpenShift	https://bit.ly/2TmngvR
OPSHJEN	User guide on using CI/CD on OpenShift Container Platform with Jenkins	https://bit.ly/2IDTBdC
OVHAUT	Description of automated deployment practices used by the company OVH	https://bit.ly/2QF51wY
OVHMAN	Description of the largely manual pipeline used by OVH before moving to more automated deployment practices	https://bit.ly/2QF51wY
REND	Description of a deployment pipeline with detailed environment annotations by M. Rendell	https://bit.ly/2MG3dCX
SKYBASE	Description of automated deployment practices related to the SkyBase DevOps platform	https://bit.ly/2sZw2B3
SPINKR	A typical deployment pipeline realized in the Spinnaker tool using Jenkins as a CI tool	https://bit.ly/2Wz7MDl
STELLI	Description of deployment pipelines in AWS by the company Stelligent	https://bit.ly/2UpcCRQ

Appendix of this paper.

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 II). 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 II. We expect that the generic aspects will likely remain stable in the future, whereas the elements in Table II might require changes or extensions. Please note that we consider this list of elements in Table II complete with regard to the sources we have studied, but these sets and rules are not considered fixed in any sense. That is, they 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**. $stype_{CPT} : CPT \to \mathbb{P}(CPT)$ 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., $\forall cpt \in CPT : stype^*_{CPT}(cpt) \cap \{cpt\} = \emptyset$. $dtype_{CP} : CP \to \mathbb{P}(CPT)$ is a function that maps each component node $cp \in CPT$ to its set of **direct component types**. $type_{CP} : CP \to \mathbb{P}(CPT)$ 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 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 II.

B. Modelling Deployment Environments

A second prerequisite for precisely specifying a deployment pipeline, which is used in almost all our sources of informal deployment pipeline descriptions, is the notion of deployment environments. They are used in two ways in CD models: First, they are used to model the deployment environments to which the deployment 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 IV-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 II 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.

Table II
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}*)$\}$: $stype_{PDNT}(c) = SCS$

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 IV-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 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 II. 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.

V. Illustrative Example Generated from the CD Models

For modelling we used our existing tool CodeableModels[2], a Python implementation for precisely specifying meta-models, models, and model instances in code with an intuitive and lightweight interface. Based on CodeableModels, we specified meta-models for components, activities, deployments, microservice-specific extensions of those, and CD-specific extensions of those, as outlined above. In addition, we realized automated constraint checkers and PlantUML code generators to generate graphical visualizations of all meta-models and models, including pipeline and component architectures. In addition to the typical UML-style diagrams such as activity, component, and deployment diagrams, we can generate more detailed views rendered as UML object diagrams. Whereas the former can be used to get an overview, the later can be used to understand the precise types of each node and all properties.

We show those more detailed views in an illustrative example for the HFGEN pipeline in Figures 1 and 2. Internally all nodes are represented as objects of a specific type, as shown in this view. These can be rendered differently, e.g., as an Initial Node symbol as used in UML activities instead of an *InitialNode* object, in order to make the models look nicer, which is useful for tasks such as getting a quick overview. The detailed views presented here are useful for inspecting all details including types and properties of each node, which is e.g. important to write formal constraints based on our models that can be checked e.g. by a static analysis tool. The deployment pipeline views of all sources examined are included in the appendix at the end of this paper.

VI. Evaluation

In Table III we report in detail on the improvements we achieved through the precise modelling. For that we carefully counted the different categories of elements and relationships we observed in the different sources and the same in our models. Firstly, this gives an impression of the size of the models and what is modelled in them. Secondly, it shows how many elements are explicitly or implicitly described by the original authors informally or in other documents but are not included in their models. We also measured the improvement in each category and for the total number of model elements.

[2]https://github.com/uzdun/CodeableModels

Note that in three cases (AZURE, HEROKU and MOZILLA), the knowledge source does *not* contain a model of the pipeline; our models were interpreted from the text descriptions of these sources. These cases have been excluded in the calculations of improvements (IMP, AVGIMP) in Table III.

The table shows that overall for almost all pipelines we observe minor to substantial improvements, with an average total improvement of 134.72%. The models in the original sources are most accurate for pipeline elements and their relations, where we see only modest average improvements of about 28.27% and 33.43%, respectively. In contrast, component connectors, component – environment relations, pipeline – component relations, and pipeline – deployment node relations are rather weak in the source models and we can see average improvements of about 60%–75%. As even the modest improvements still indicate substantial gaps in the models, our results clearly indicate that our formal modelling approach can substantially improve the precision of the models at the modest cost of learning our rather intuitive (i.e., close to the model elements in the original models) modelling approach (please note that the mathematical notation used in this paper for precise reporting is not needed to apply the approach, but can be replaced by visual or textual modelling approaches).

VII. 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 missing 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 same applies to our coding procedure and the formal modelling: Other researchers might have coded or modelled differently, leading to different models. As our goal was only to find one model that is able to specify all observed phenomena, and this was achieved, we consider this threat not to be a major issue for our study.

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. Note that our procedure

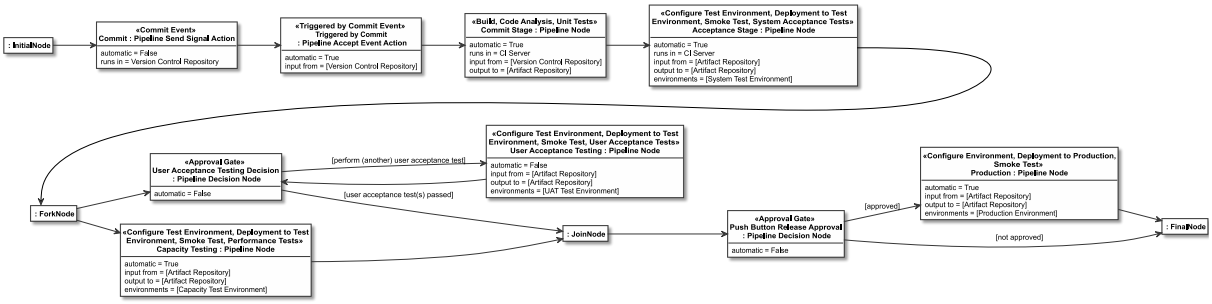


Figure 1. Detailed object view of the HFGEN deployment pipeline model generated from our CD models (manually broken in the middle to make it small enough to fit onto the page in the paper)

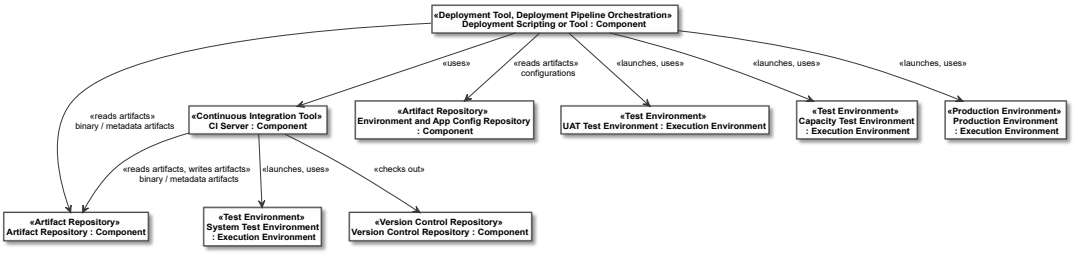


Figure 2. Detailed object view of the HFGEN infrastructure and environment model generated from our CD models

is in this regard rather similar to how interview partners are typically found in qualitative research studies in software engineering today. However, the threat remains that our procedures introduced some kind of unconscious exclusion of certain sources; we mitigated this by assembling an author team with many years of experience in the field, and performing very general and broad searches. Due to the many included sources, it is likely our results can be generalized to a larger population of similar pipeline models and architectures. As most of the sources use tools and environments well established in the field, it is probable this covers a large fraction of the existing 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.

VIII. Conclusions

We have performed a qualitative study in which we have studied the design and architecture of deployment pipeline from 25 unique and independent sources. Our study led to a detailed model precisely describing the recurring deployment 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, we have extended mainly component model abstractions, to answer RQ3. In all three cases, we observed theoretical saturation relatively earlier 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] M. Huttermann, *DevOps for Developers*. APress, 2012.

Table III
IMPROVEMENTS IN PRECISION OF MODELLING

Case	CNS	CNM	CES	CEM	ECS	ECM	DNS	DNM	DRS	DRM	PES	PEM	PRS	PRM	PCS	PCM	PDS	PDM	TOS	TOM	IMP
ADOBE	5	6	6	9	3	8	2	3	0	2	11	18	10	18	17	20	3	10	57	94	64.9%
AWSGO	2	5	0	6	0	4	2	3	0	2	4	8	3	8	2	13	2	0	15	49	226.67%
AZURE	0	5	0	6	0	4	0	3	0	2	0	9	0	9	0	15	0	3	0	56	–
BITBAR	5	6	1	6	0	5	2	5	1	4	6	10	5	9	5	17	2	6	27	68	151.85%
BROIND	0	0	0	0	0	0	0	0	0	0	6	10	6	10	0	0	0	0	12	20	66.67%
CCAUT	1	5	0	4	0	2	1	2	0	0	22	19	21	18	0	30	0	12	45	92	104.44%
CCMAN	1	5	0	4	0	2	1	2	0	0	24	29	23	32	6	44	0	20	55	138	150.91%
FACEB	3	7	0	6	1	7	3	4	6	2	6	23	8	26	3	18	3	7	33	100	203.03%
GOOGLE	6	5	6	4	2	5	2	1	0	0	10	13	9	13	0	17	0	6	35	64	82.86%
HEROKU	0	7	0	7	0	6	0	3	0	0	0	19	0	19	0	23	0	5	0	89	–
HFCOMP	2	5	0	5	0	4	0	4	0	0	14	30	15	40	10	46	0	8	41	142	246.34%
HFGEN	3	5	0	5	12	4	0	4	0	0	18	13	17	15	14	14	0	4	64	64	0%
IBM	0	7	0	8	0	2	2	2	0	0	3	7	2	6	0	15	2	2	9	49	444.44%
IOT	6	6	5	5	0	4	1	1	0	0	8	9	7	8	11	12	2	2	40	47	17.50%
JENKINS	8	8	0	7	0	2	0	2	0	0	10	12	9	11	0	23	0	4	27	69	155.55%
MOZILLA	0	5	0	4	0	6	0	3	0	0	0	18	0	21	0	18	0	0	0	75	–
NETFLIX	5	10	1	9	0	8	0	3	0	2	8	10	7	9	10	18	0	4	31	73	135.48%
OPSHIFT	2	9	0	8	0	2	5	2	3	0	5	21	4	21	1	25	4	3	24	91	279.17%
OPSHJEN	5	6	6	8	4	3	2	3	0	2	8	14	7	13	5	23	0	4	37	76	105.4%
OVHAUT	9	7	2	6	6	2	2	4	0	4	4	19	3	19	9	15	2	6	37	82	121.62%
OVHMAN	5	3	1	0	3	0	1	3	0	2	4	7	3	6	5	2	1	3	23	26	13.04%
REND	0	1	0	0	0	2	0	2	0	0	16	26	21	31	0	0	0	11	37	73	97.3%
SKYBASE	2	5	0	4	0	4	4	5	0	4	10	14	10	15	6	21	6	7	38	79	107.89%
SPINKR	5	9	7	10	0	7	2	4	1	0	12	9	11	8	11	17	3	0	52	64	23.07%
STELLI	5	9	1	7	3	6	0	6	0	5	6	15	5	17	8	15	4	5	32	85	165.63%
AVGIMP	38.76%		70.24%		59.04%		50.77%		62.07%		28.27%		33.43%		70.37%		75.80%		50.27%		134.72%

Legend:

CNS – Number of component nodes formally modelled in the knowledge source; **CNM** – The same in our model.

CES – Number of connector edges formally modelled in the knowledge source; **CEM** – The same in our model.

ECS – Number of environment–component relations formally modelled in the knowledge source; **ECM** – The same in our model.

DNS – Number of deployment nodes formally modelled in the knowledge source; **DNM** – The same in our model.

DRS – Number of deployment node relations formally modelled in the knowledge source; **DRM** – The same in our model.

PES – Number of pipeline elements formally modelled in the knowledge source; **PEM** – The same in our model.

PRS – Number of pipeline element relations formally modelled in the knowledge source; **PRM** – The same in our model.

PCS – Number of pipeline element–component relations formally modelled in the knowledge source; **PCM** – The same in our model.

PDS – Number of pipeline element–deployment node relations formally modelled in the knowledge source; **PDM** – The same in our model.

TOS – Total number of elements formally modelled in the knowledge source; **TOM** – The same in our model.

IMP – Improvement in our model in %; **AVGIMP** – Average improvement per category in %.

[2] L. E. Lwakatare, P. Kuvaja, and M. Oivo, "Dimensions of devops," in *Agile Processes, in Software Engineering, and Extreme Programming - 16th International Conference (XP), Helsinki, Finland, May 25-29, 2015, Proceedings*, 2015, pp. 212–217.

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

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

[5] C. Parnin, E. Helms, C. Atlee, H. Boughton, M. Ghattas, A. Glover, J. Holman, J. Micco, B. Murphy, T. Savor, M. Stumm, S. Whitaker, and L. Williams, "The top 10 adages in continuous deployment," *IEEE Software*, vol. 34, no. 3, pp. 86–95, 2017.

[6] D. Caroff, "How we switched to a continuous delivery pipeline in 3 months," https://medium.com/devopslinks/how-we-switch-to-a-continuous-delivery-pipeline-in-3-months-9667b9f65f7a, 2018.

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

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

[9] K. Plakidas, D. Schall, and U. Zdun, "Software migration and architecture evolution with industrial platforms: A multi-case study," in *European Conference on Software Architecture*. Springer, 2018, pp. 336–343.

[10] 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.

[11] B. G. Glaser and A. L. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research*. de Gruyter, 1967.

[12] 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.

[13] M. Leppanen, S. Makinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. V. Mantyla, and T. Mannisto, "The highways and country roads to continuous deployment," *IEEE software*, no. 2, pp. 64–72, 2015.

[14] 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.

[15] G. Schermann, D. Schöni, P. Leitner, and H. C. Gall, "Bifrost: supporting continuous deployment with automated enactment of multi-phase live testing strategies," in *Proceedings of the 17th International Middleware Conference*. ACM, 2016, p. 12.

[16] J. Cito, P. Leitner, T. Fritz, and H. C. Gall, "The making of cloud applications: An empirical study on software development for the cloud," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 393–403.

[17] 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.

[18] 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.

[19] O. Zimmermann, "Microservices tenets," *Computer Science-Research and Development*, vol. 32, no. 3-4, pp. 301–310, 2017.

[20] L. Baresi, M. Garriga, and A. De Renzis, "Microservices identification through interface analysis," in *European Conference on Service-Oriented and Cloud Computing*, 2017, pp. 19–33.

[21] 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.

[22] C. Richardson, "A pattern language for microservices," http://microservices.io/patterns/index.html, 2017.

[23] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. M. Josuttis, "Microservices in practice, part 2: Service integration and sustainability." *IEEE Software*, vol. 34, no. 2, pp. 97–104, 2017.

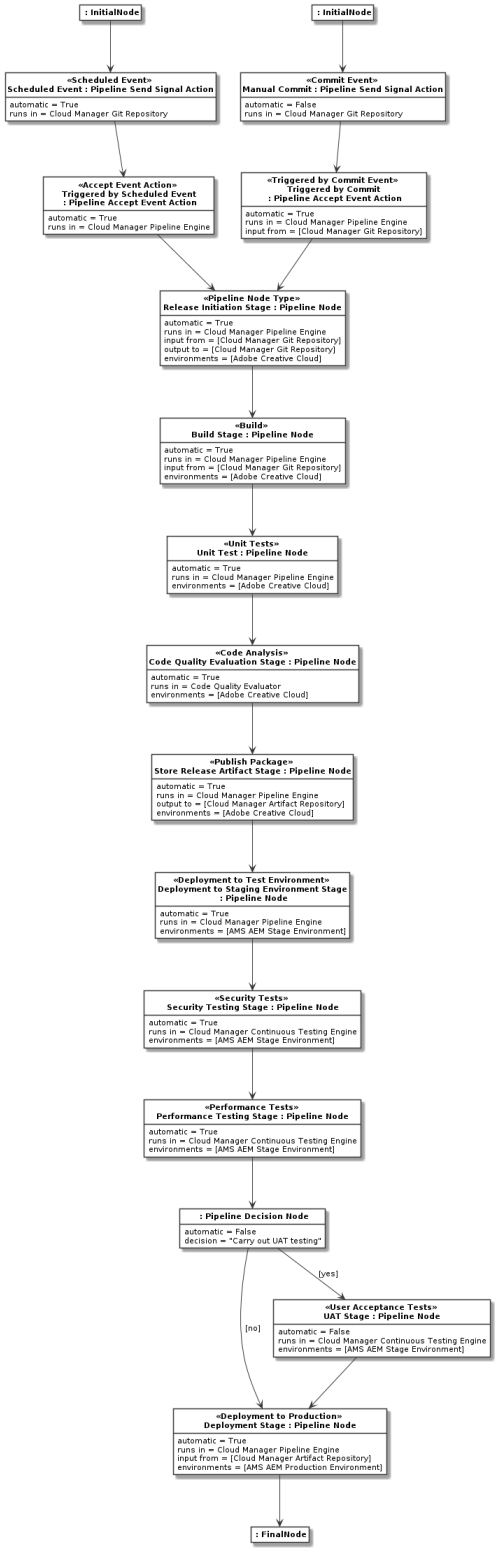
[24] J. Coplien, *Software Patterns: Management Briefings*. SIGS, New York, 1996.

[25] 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.

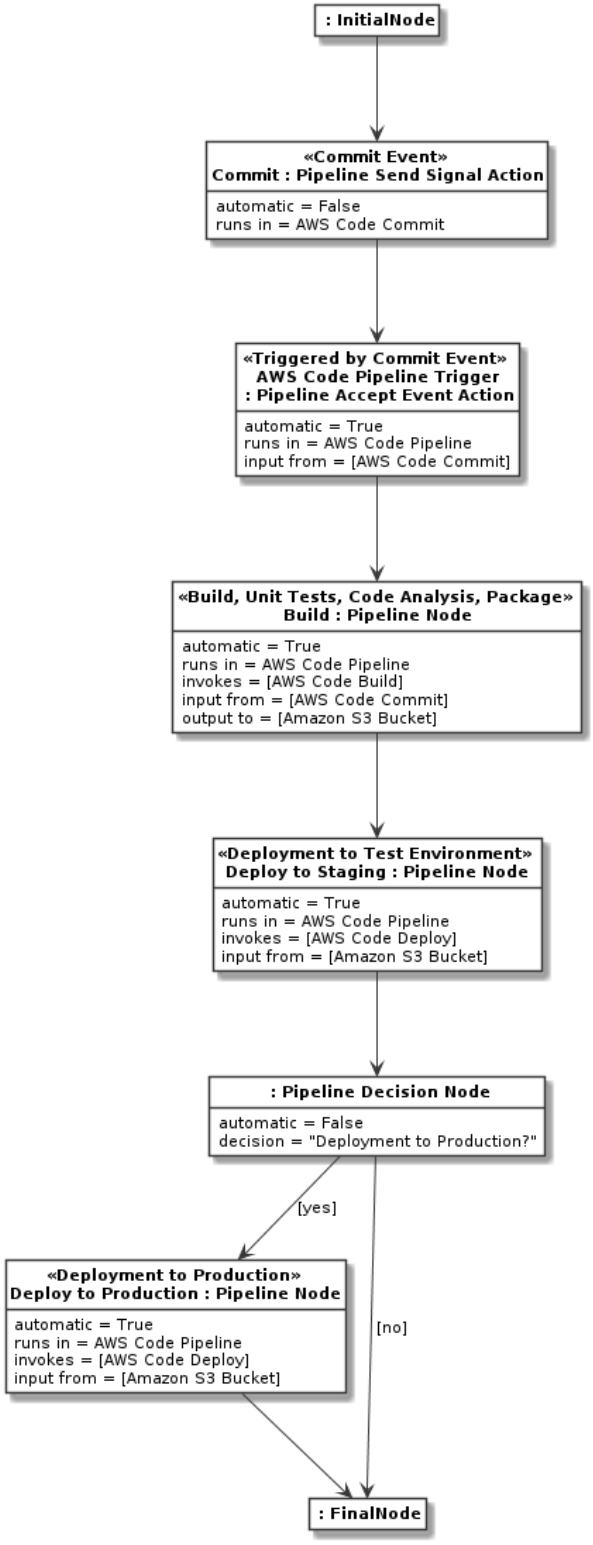
[26] K. Charmaz, *Constructing grounded theory*. Sage, 2014.

APPENDIX
PIPELINE MODELS

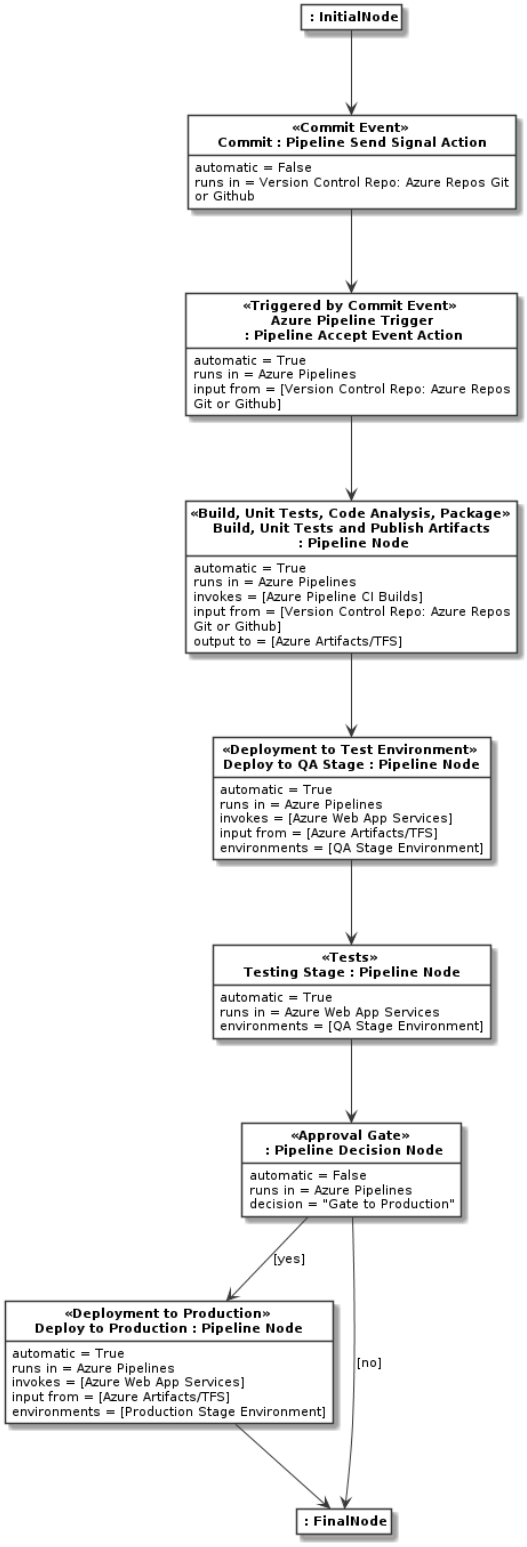
A. *Model for the Adobe Cloud Manager Pipeline (ADOBE)*



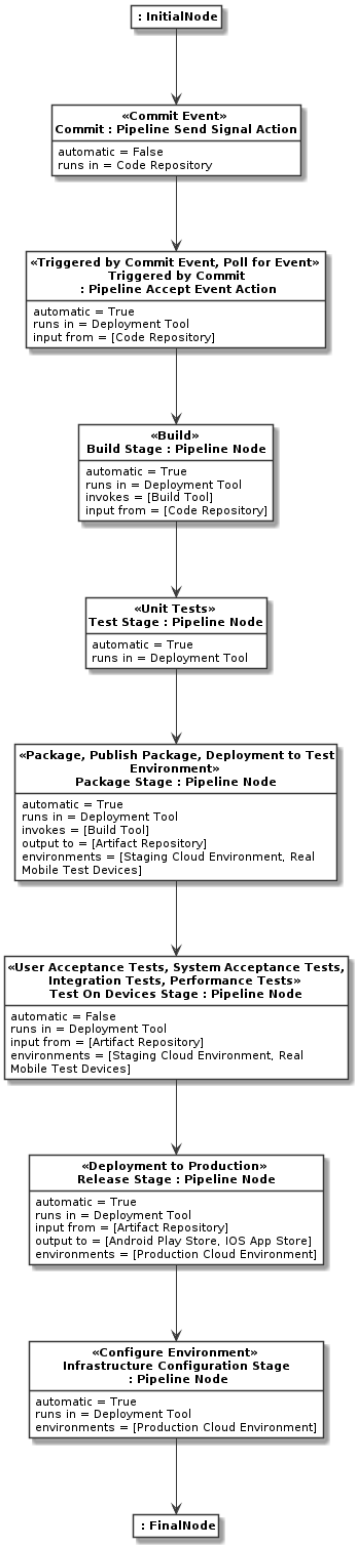
B. Model for the AWS in Go Pipeline (AWSGO)



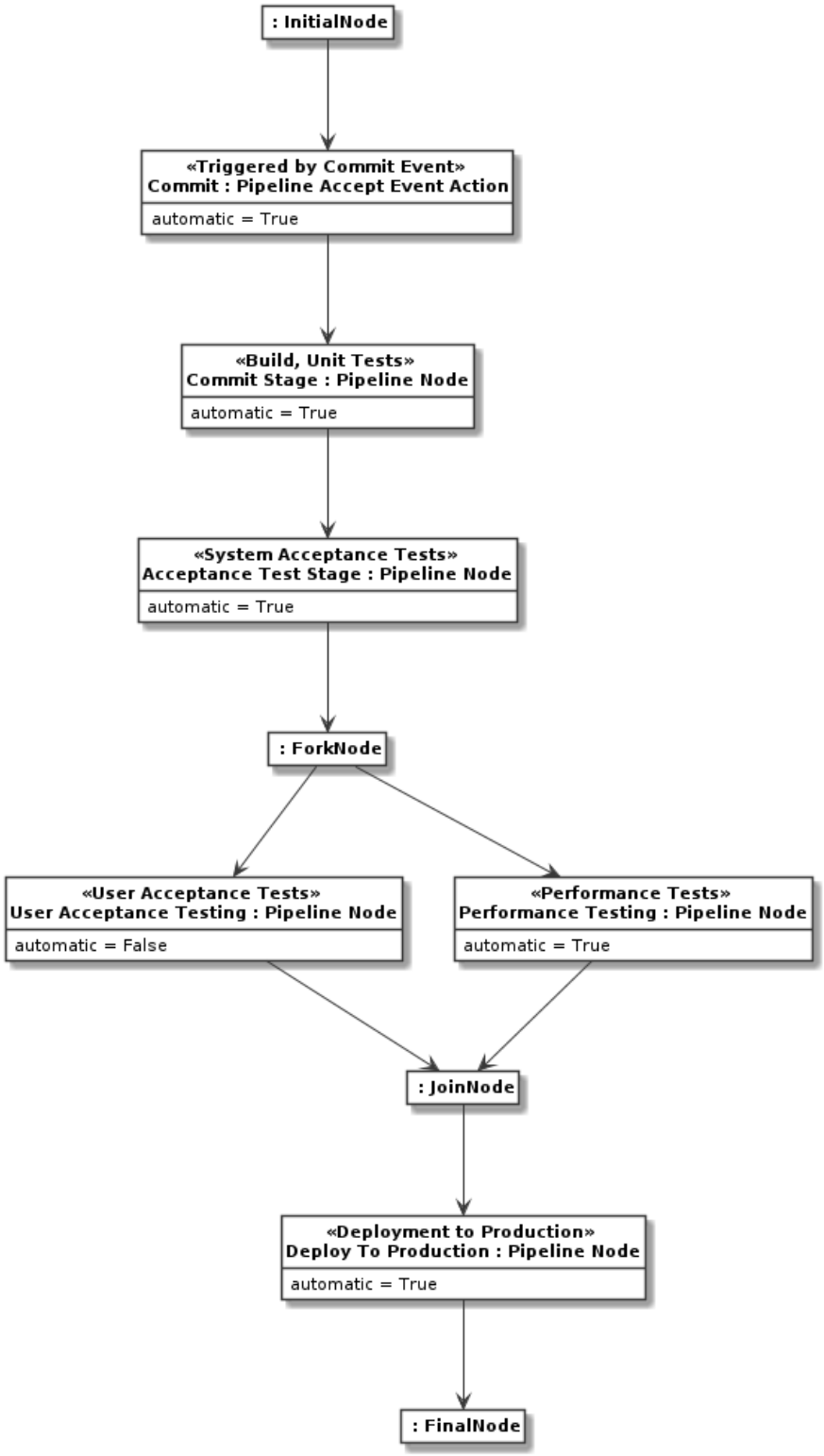
C. Model for the Azure DevOps Pipeline (AZURE)



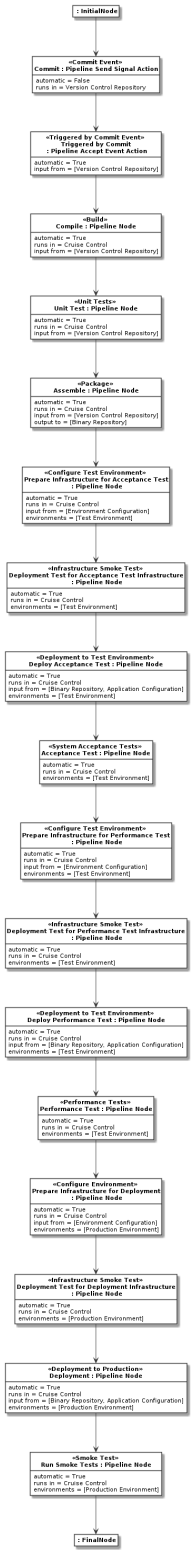
D. Model for the Bitbar Mobile CI Pipeline (BITBAR)



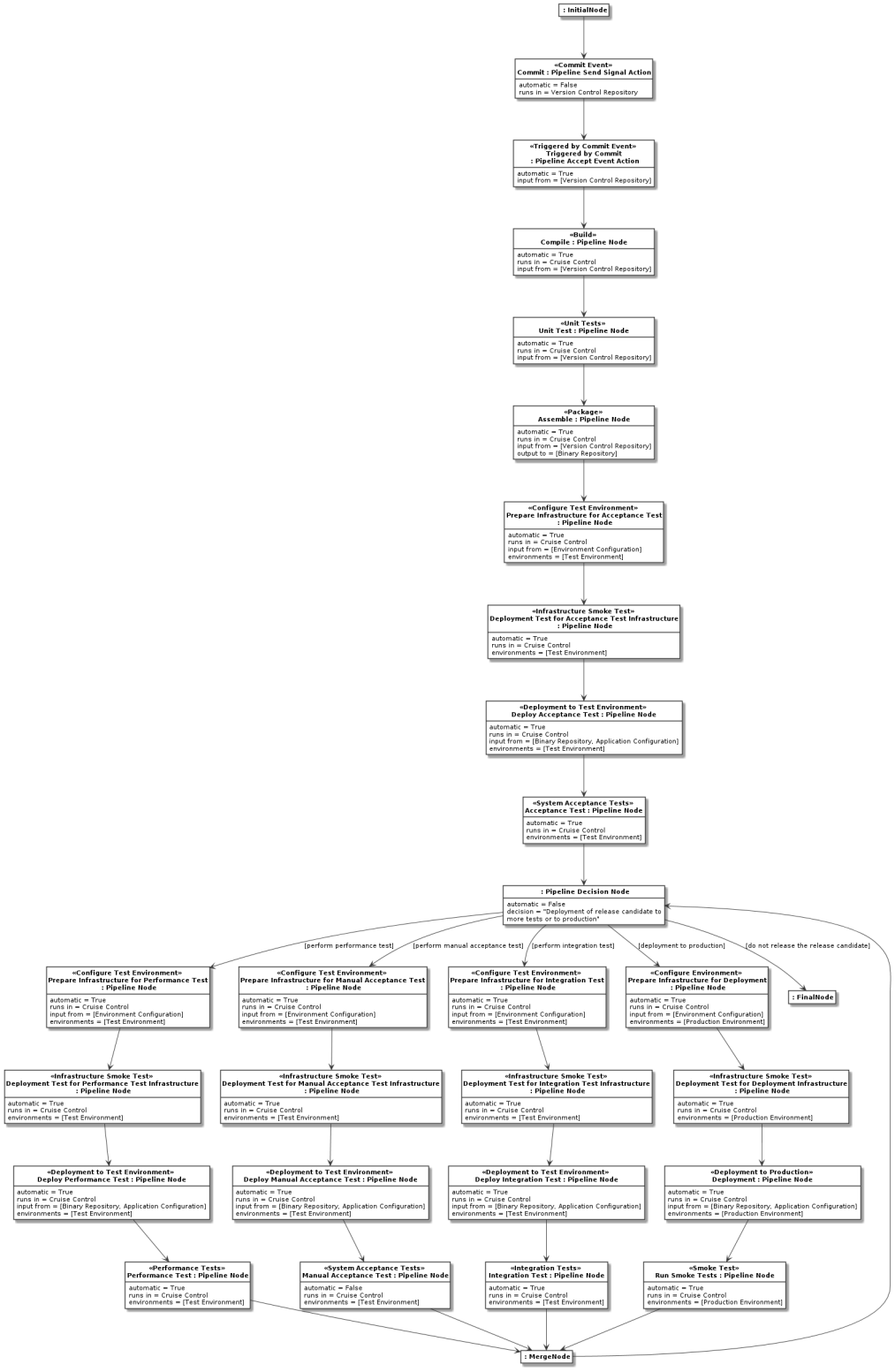
E. Model for the R. Brown & R. Indugula Pipeline (BROIND)



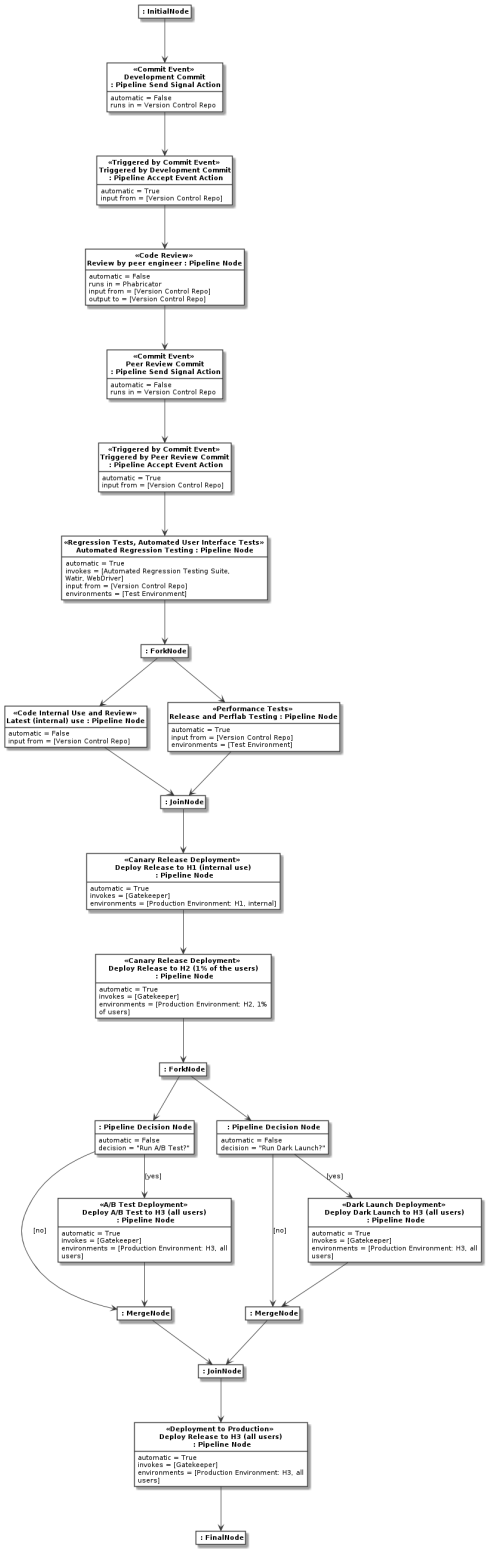
F. Model for the Fully Automated Cruise Control Pipeline (CCAUT)



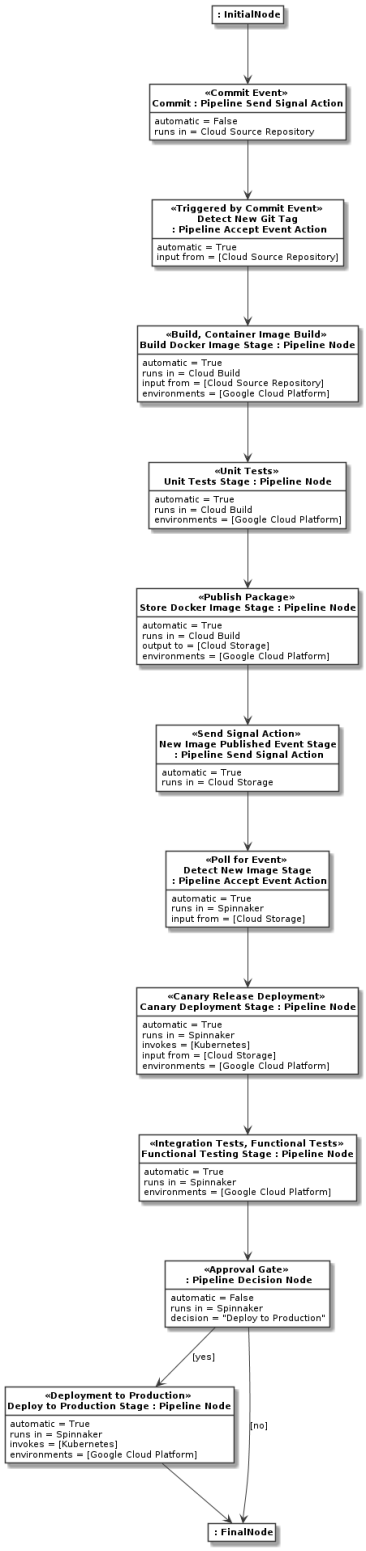
G. Model for the Manual-Gated Cruise Control Pipeline (CCMAN)



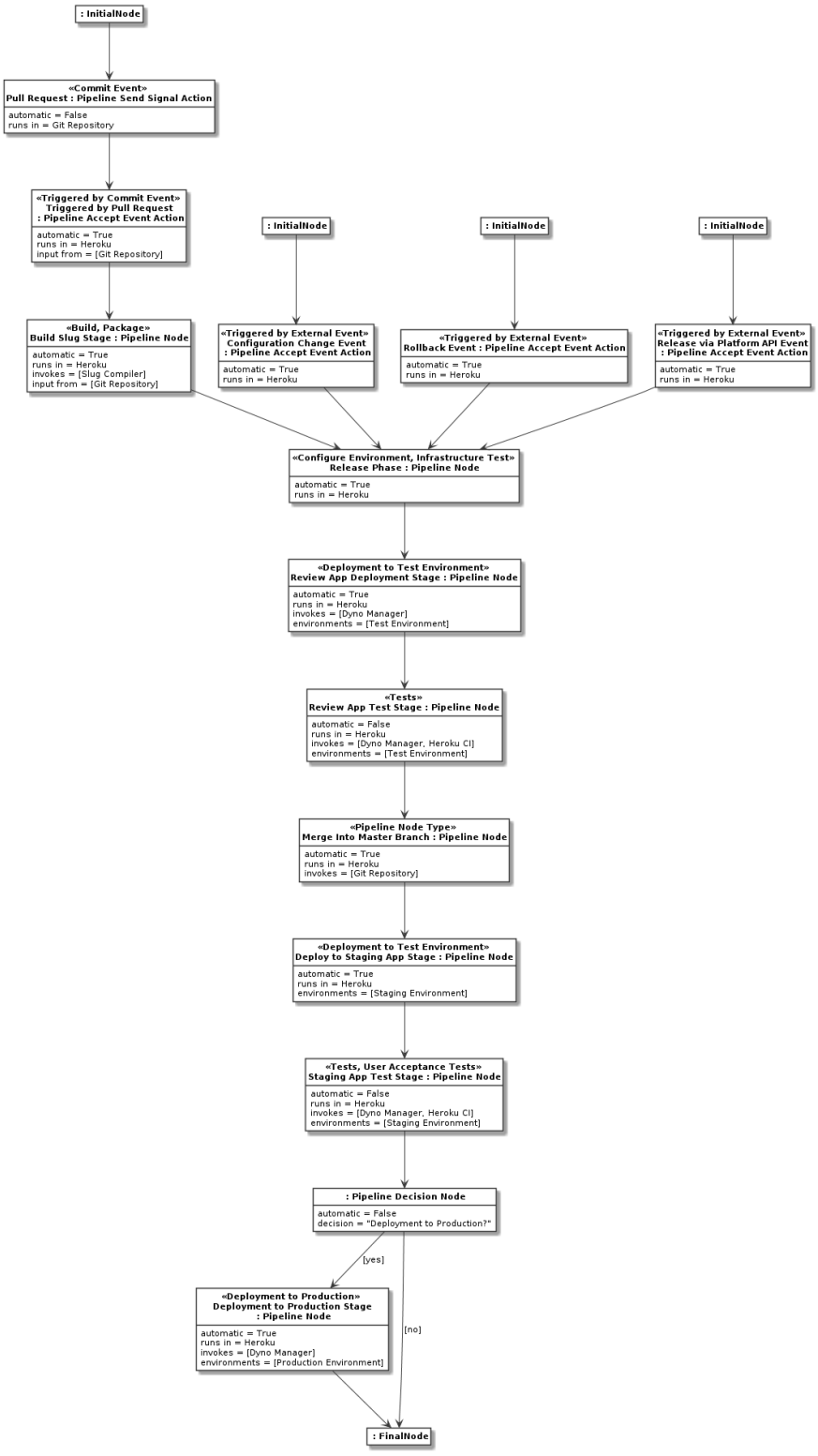
H. Model for the Facebook Pipeline (FACEBOOK)



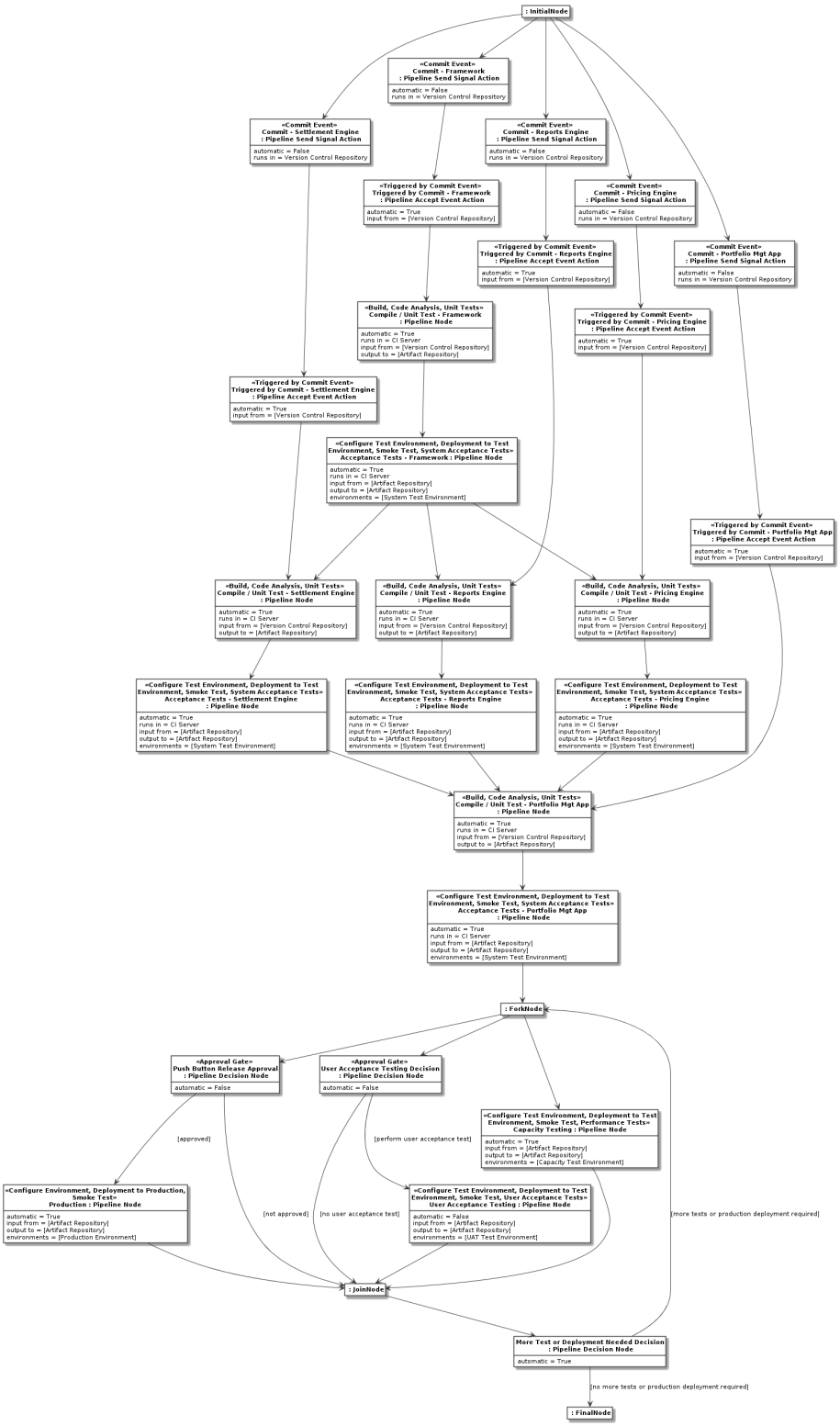
I. Model for the Google Cloud CI Pipeline (GOOGLE)



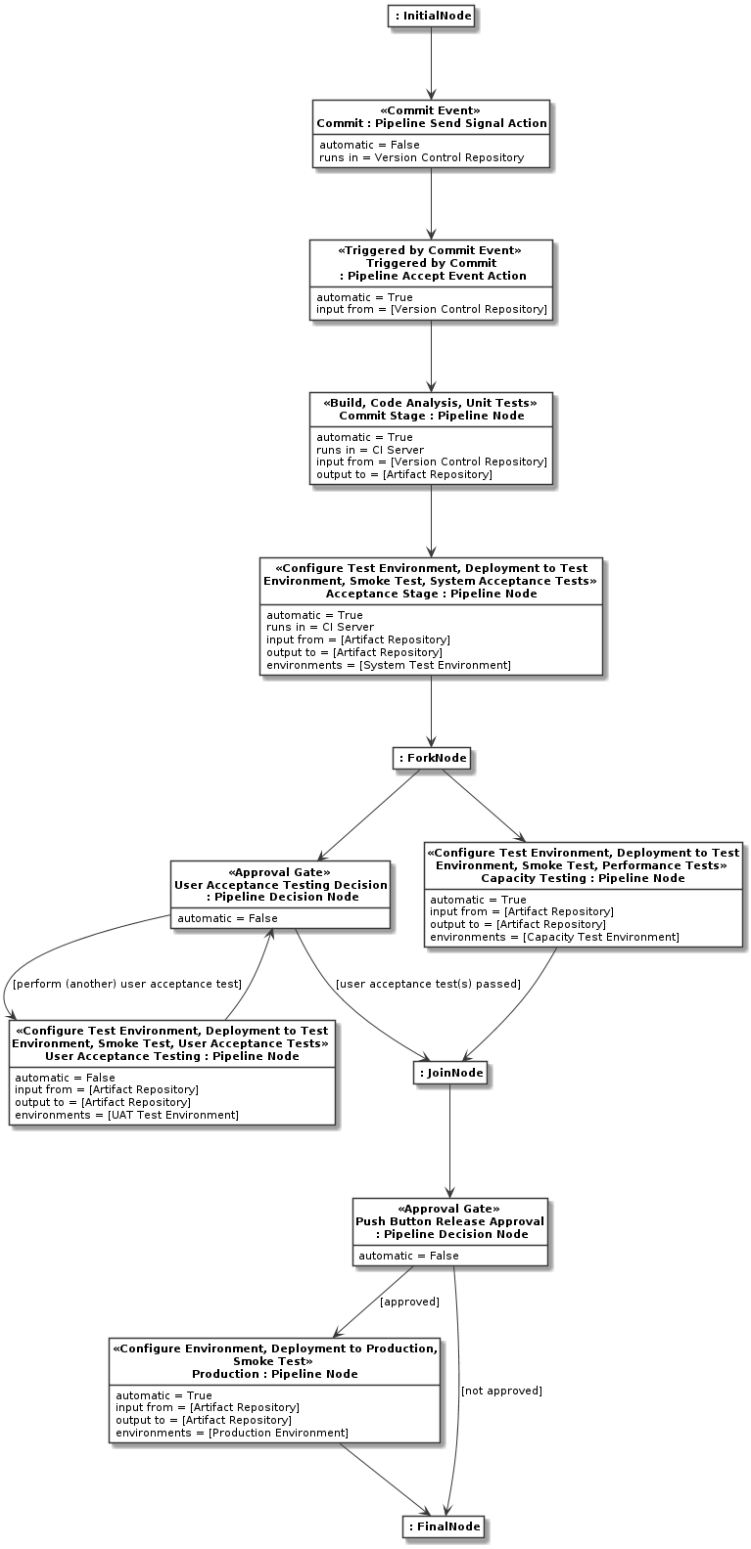
J. Model for the Heroku CI Pipeline (HEROKU)



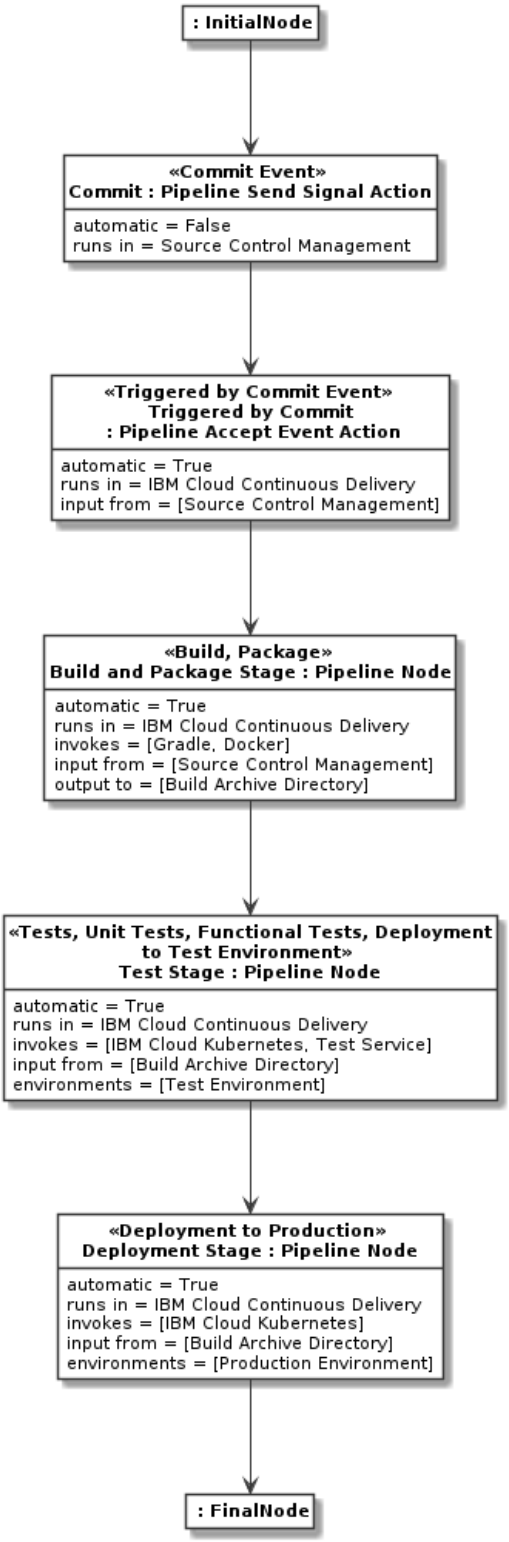
K. Model for the Component-Oriented J. Humble & D. Farley Pipeline (HFCOMP)



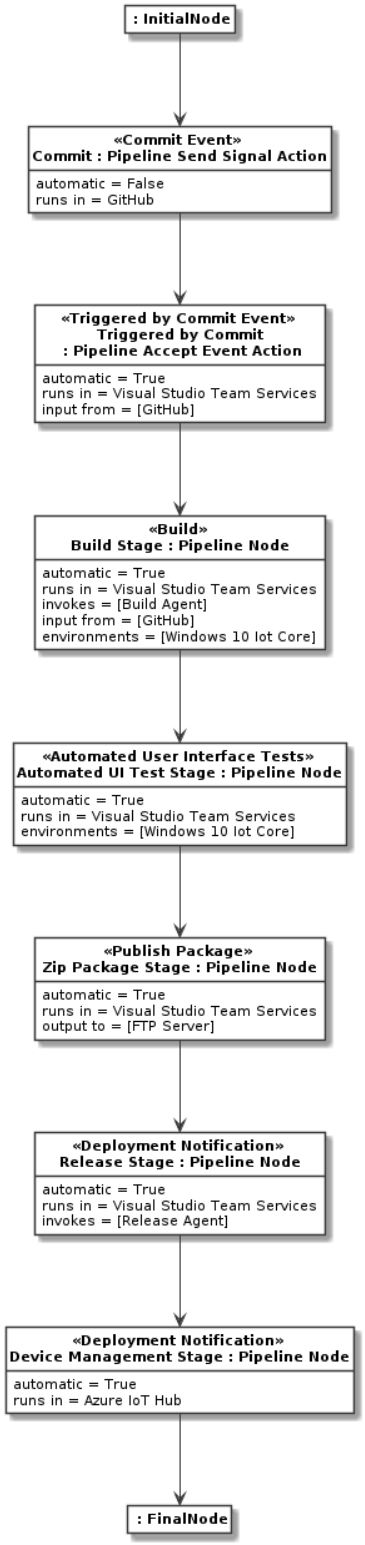
L. Model for the Generic J. Humble & D. Farley Pipeline (HFGEN)



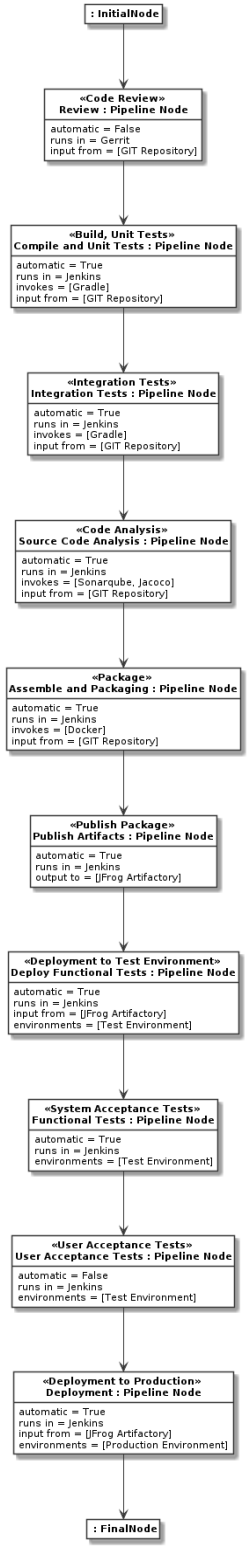
M. Model for the IBM Cloud CD Pipeline (IBM)



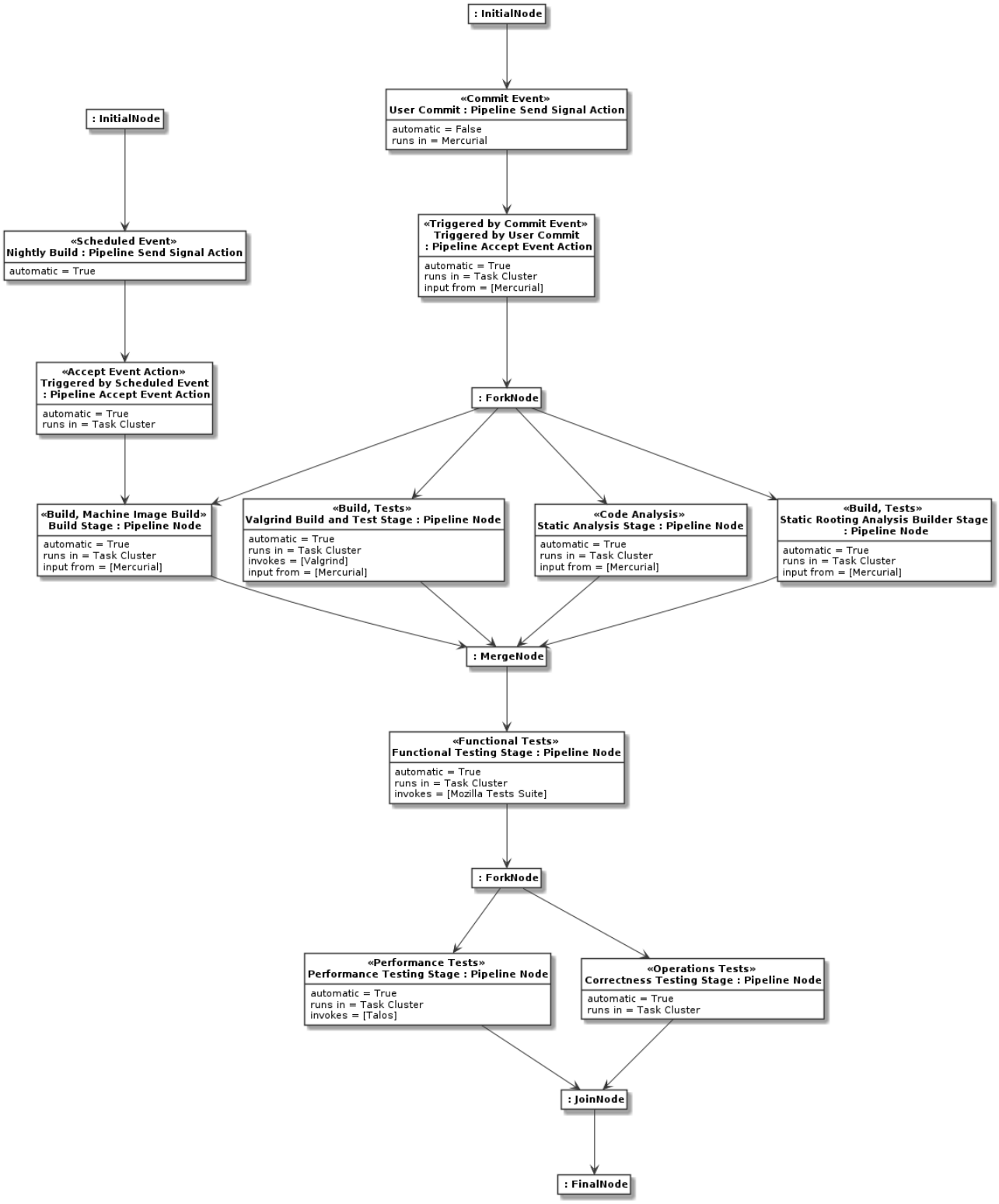
N. Model for the DevOps for IoT Pipeline (IOT)



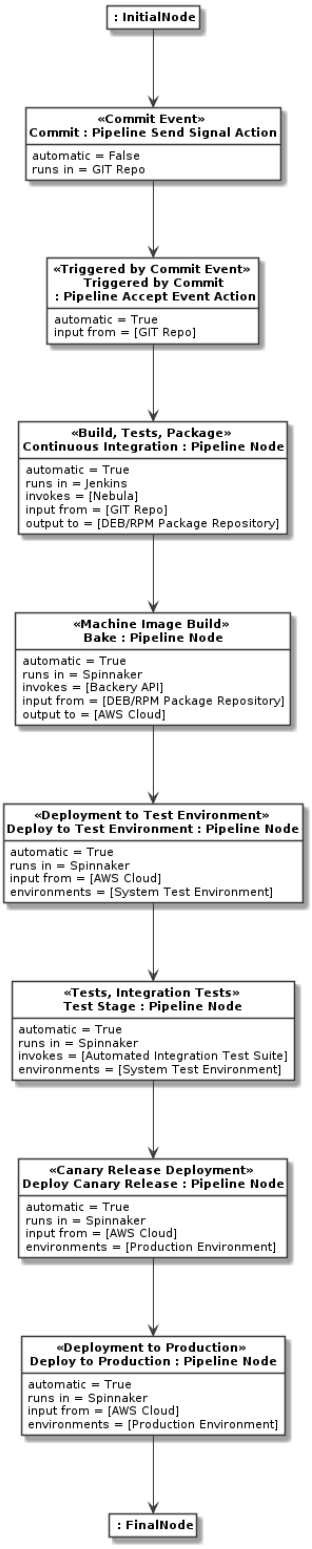
O. Model for the Jenkins-based CD Pipeline (JENKINS)



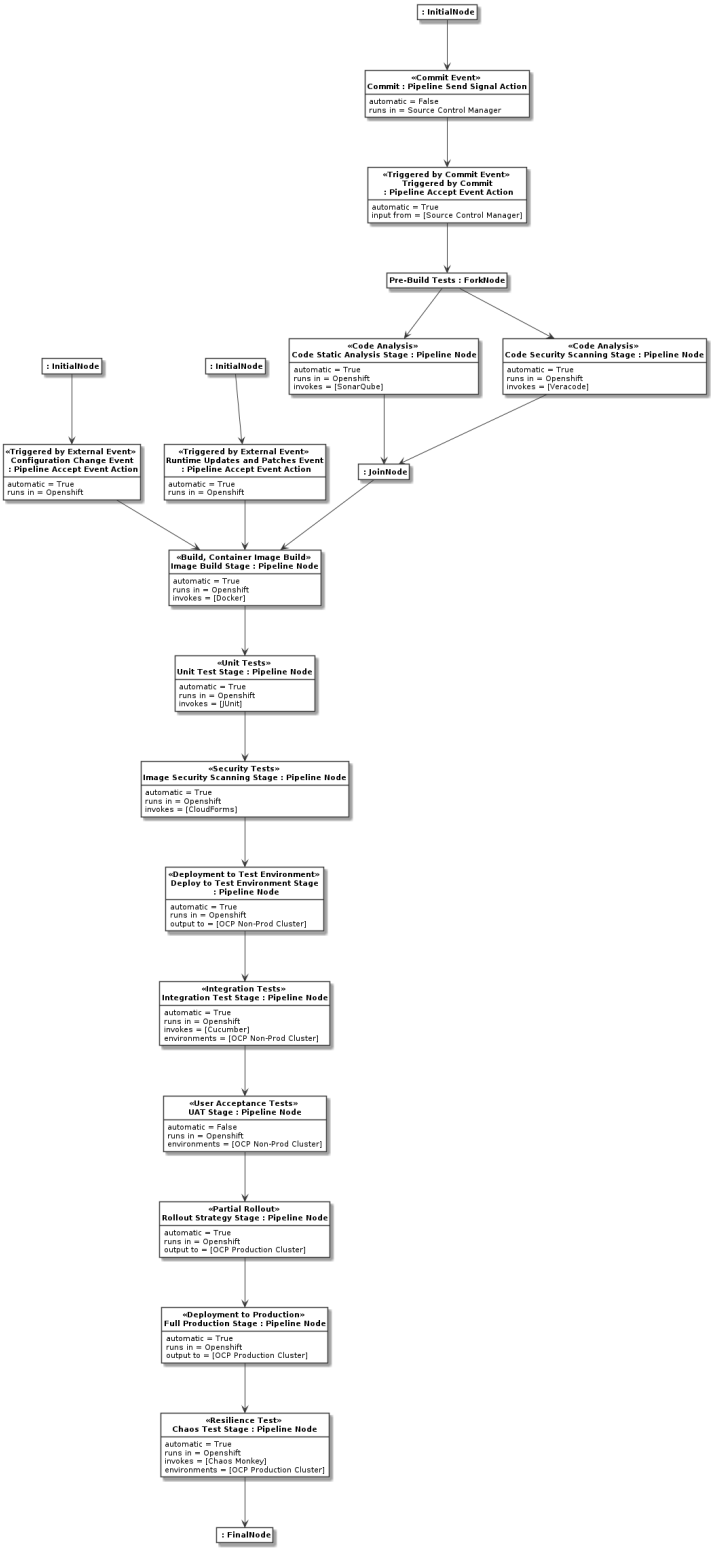
P. Model for the Mozilla CI Pipeline (MOZILLA)



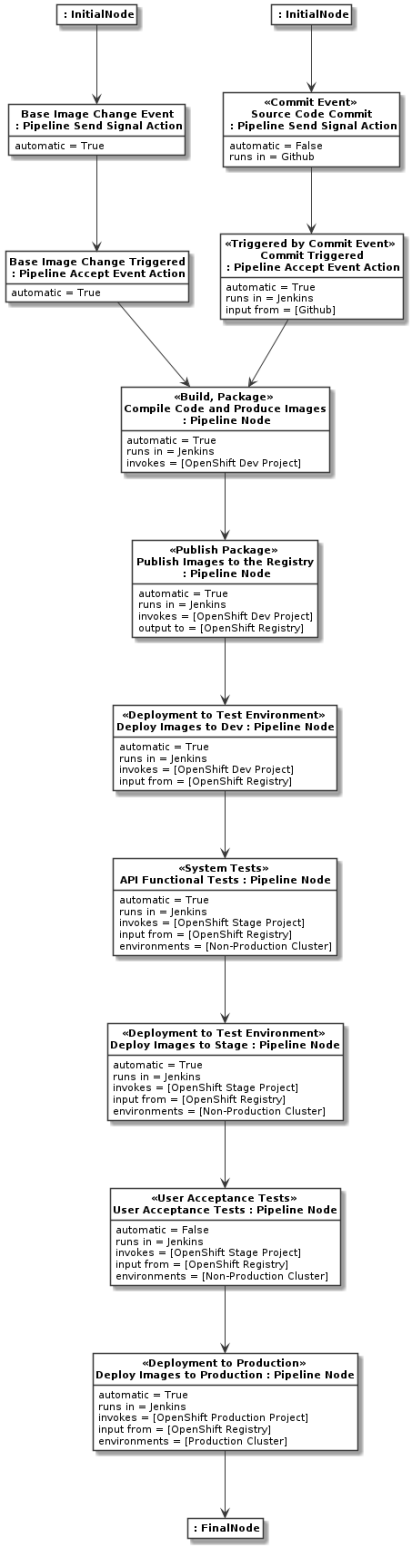
Q. Model for the Netflix CD Pipeline (NETFLIX)



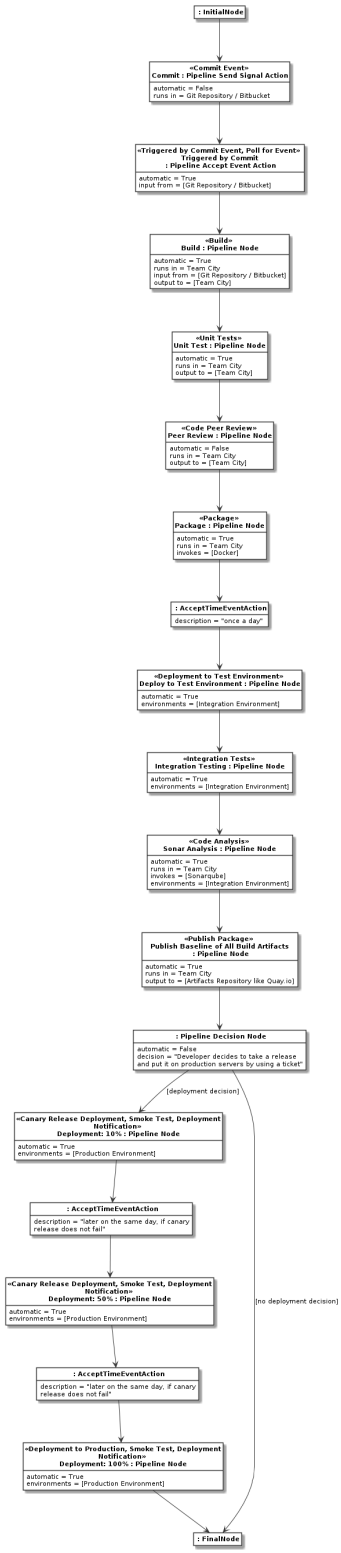
R. Model for the OpenShift CI/CD Pipeline (OPSHIFT)



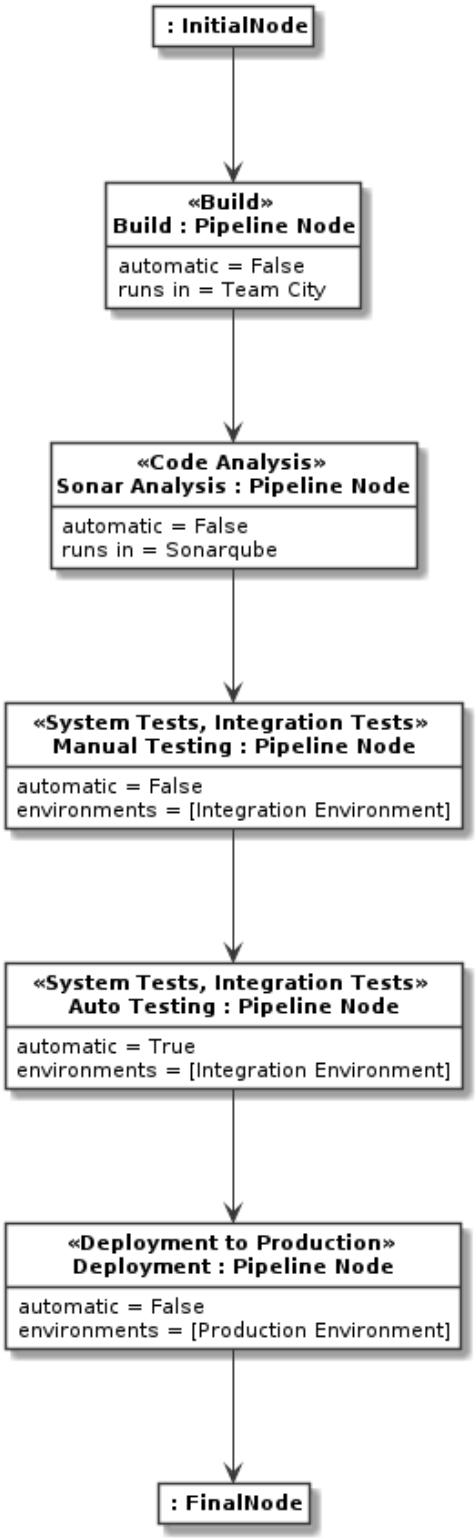
S. Model for the OpenShift Container Platform with Jenkins Pipeline (OPSHJEN)



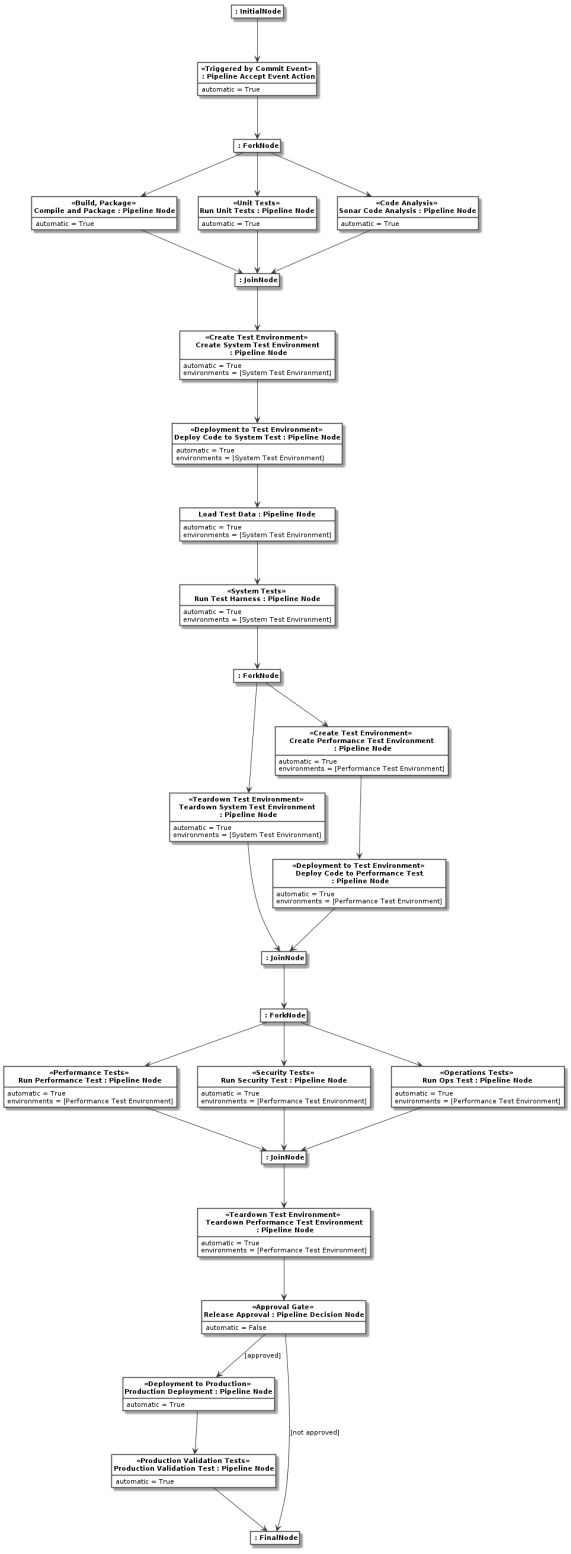
T. Model for the OVH Aut. Deployment Pipeline (OVHAUT)



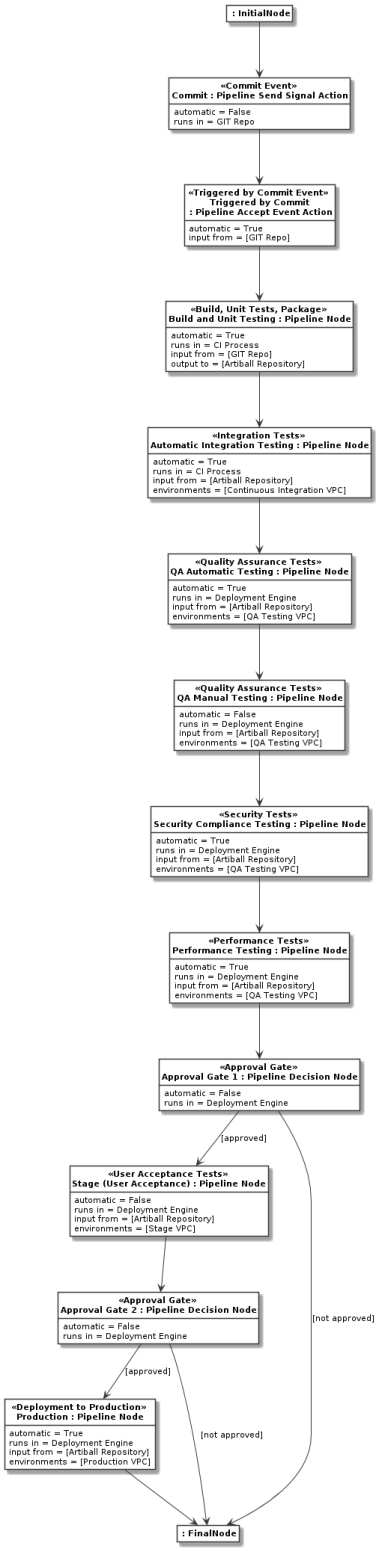
U. Model for the OVH Manual Deployment Pipeline (OVHMAN)



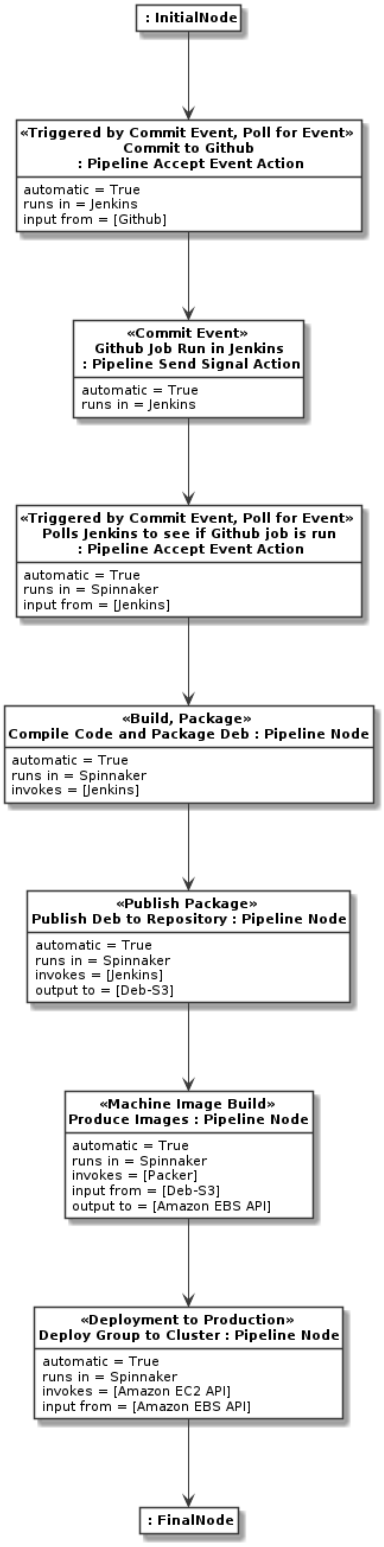
V. Model for the M. Rendell CD Pipeline (REND)



W. Model for the SkyBase DevOps Platform Pipeline (SKYBASE)



X. Model for the Spinnaker with Jenkins CD Pipeline (SPINKR)



Y. Model for the Stelligent CD Pipelines in AWS (STELLI)