OULUN AMMATTIKORKEAKOULU

Patrik Teivonen

STAGING ENVIRONMENT IMPLEMENTATION IN A SOFTWARE DELIVERY AUTOMATION PIPELINE

STAGING ENVIRONMENT IMPLEMENTATION IN A SOFTWARE DELIVERY AUTOMATION PIPELINE

Patrik Teivonen
Final projects
Spring 2024
Degree Program in Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Program in Information Technology

Author(s): Patrik Teivonen
Title of the thesis: Staging Environment Implementation in a Software Delivery Automation Pipeline
Thesis examiner(s): Teemu Leppänen
Term and year of thesis completion: Spring 2024 Pages: 28

This thesis aims to evaluate the use of staging environments in enhancing the development processes for software delivery automation pipeline tooling. With the implementation of a staging environment for an existing delivery automation pipeline case, it was envisioned that adding it would ease the testing of changes to the tools and configurations while not disrupting the production environment and releases, thus improving the efficiency and the quality of the release process. The work was commissioned by The Qt Company Oy.

The first step in achieving the intended goals was to map out the current automation infrastructure and to identify the minimal required changes for the pipeline to support a multi-deployment scenario and which parts needed to be replicated. Then, a more refined version control branching strategy was chosen as a vehicle to make those adjustments without unnecessarily disrupting the production deployment and for the future development of the automation tooling. Finally, the deployment of the automation pipeline was conducted to provisioned machines in the staging environment, verifying that the configurations are correctly set up and that all parts of the pipeline functioned as expected.

The thesis work concluded with a functioning initial deployment of the staging pipeline. Additionally, the new proposed development and testing process for automation tooling was trialed with taking it in use for the subsequent changes. The new branching and production update strategy has already proved effective in speeding up the development workflow separating it from other release activities, although it will take some more time for everything to adopt the new process. While not all the benefits of the staging environment could be realized at this early stage, the results gathered were considered useful in detecting gaps in the current deployment and planning future enhancements.

Keywords: continuous delivery, packaging, releasing, Jenkins, Ansible, staging environment, automation pipeline, quality assurance, software development process

CONTENTS

1 TERMS AND ABBREVIATIONS

AD	Microsoft Active Directory
API	Application programming interface
AWS	Amazon Web Services
CD	Continuous delivery
CDN	Content delivery network
CI	Continuous integration
CLI	Command line interface
HTTP	Hypertext Transfer Protocol
INI	Initialization file, a file format for configurations
IT	Information technology, a process team for technical assistance and systems
JSON	JavaScript Object Notation, a file format
LDAP	Lightweight Directory Access Protocol
NFS	Network File System
README	An accompanying manual or documentation text file in a markdown format
REST	Representational state transfer, an architecture
RTA	Release test automation
SDLC	Software development life cycle
SSH	Secure Shell Protocol
UI	User interface
URL	Uniform Resource Locator, a type of Uniform Resource Identifier (URI)
VCS	Version control system
VM	Virtual machine
XML	Extensible Markup Language, a file format
YAML	YAML Ain't Markup Language, a data serialization language

2 INTRODUCTION

The Qt Company Oy is a multinational software company headquartered in Finland mainly known for its development framework Qt, whose technology is widely used globally by software developers across many different types of platforms and systems. It has undergone a growth transformation over the years to attain a global presence, diversifying its product portfolio to encompass all areas of software development, with the most recent acquisitions in the quality assurance category. (1.)

While the increase in products and the expansion of present offerings might have opened new avenues for additional growth and market penetration, it has also brought to light new challenges in handling the rising number of releases in the cycle with the current processes that have finally started to prove inadequate. The workflow with a single linear pipeline has worked well in the past, but the frequency of the releases combined with the complexity of the system has reduced the amount of time slots allocated for developing and testing the automation in place. The need for a more streamlined process was apparent to maintain the robustness and quality of continuous delivery.

To effectively address these concerns, the release team recognizes the significance of proposing to add a separate deployment of the automation pipeline, only for testing purposes. The changes to the automation tooling and configurations could be then thoroughly evaluated in this simulated staging environment before deploying them to the production environment, allowing for quicker incorporation of new features to the pipeline while simultaneously reducing the risk of delayed releases. It is envisioned that significant improvements will be seen in the efficiency, predictability, and reliability of the whole release process after this change, thereby also minimizing the impact of potential issues as these will be caught earlier.

This thesis aims to contribute to the domain of software delivery automation by documenting the implementation of the described staging pipeline deployment case for The Qt Company Oy and establishing a new proposed process model for the development of the automation tooling. While the suitable tooling and processes can vary depending on the use case, the findings of the work are also expected to underscore the importance of staging deployments and effective processes in the context of other continuous delivery systems alike.

3 THEORY

3.1 Software Delivery Automation

The term software delivery describes a process where a software product undergoes a transformation from the initial concept through development to the end users in the form of a final licensed product. This involves a series of steps that the company teams must execute to make the said product available for the customer to access via the chosen distribution method. (2.)

Software development in today's fast-paced world necessitates quick and reliable delivery of high-quality software. To measure the performance or efficiency of the software delivery process, various metrics can be used. Quantitative metrics include but are not limited to the lead and cycle time, the frequency of deployment, the change failure rate, and code refactoring ratios. On the other hand, one of qualitative aspects might focus on getting feedback to identify the strong and weak points regarding the expectations from the customer. Modern software products are increasingly complex, meaning traditional manual approaches risk not always fulfilling these requirements as measured by the metrics. Automation is seen as one of the most important ways to tackle the speed problem in software delivery reducing the time needed to complete the Software Development Life Cycle (SDLC). (3.)

Organizations that implement automation practices such as Continuous Integration (CI) or Continuous Delivery (CD) have been seen to score higher on performance metrics as documented by industry data. In 2021, one of such reports, "Accelerate State of DevOps" by DevOps Research and Assessment (DORA), the research showed that highest performing organizations adopting these practices have a change failure rate that is 3 times lower and can deploy code 973 times more frequently than their low performing counterparts. These findings highlight the substantial benefits of automation in improving both the speed and reliability of software delivery processes. (4, p. 12-13.)

3.2 Delivery Automation Pipeline

Pipelines are one way to represent the steps of the software delivery process. They can be defined as sets of automated processes which speed up the software delivery. There are no universally specific rules how a pipeline should look like and the tools which it must utilize considering software delivery processes can differ even within the same organization among various software products and teams. To give a simplified example of a delivery automation pipeline in the context of this thesis, the main products of The Qt Company such as the Qt Framework appear to follow a commonly found structure consisting of the components seen in figure 1 below.

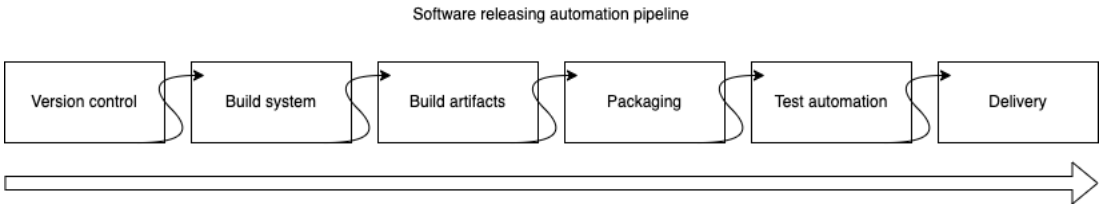

FIGURE 1. The main stages of a delivery automation pipeline

The delivery automation pipeline begins with a revision control system, Git (5), where the developers will commit their code. Before integration, contributions are reviewed (6) through the web-based code review system, Gerrit (7; 8). From there the code proceeds to being built in a Continuous Integration (CI) build system, such as the open-source automation server Jenkins (9) or the in-house built CI environment replacement, Coin (10). After completing the CI stage, the relevant build artifacts are usually exported for the various other Continuous Delivery (CD) processes which includes the packaging, deploying, testing, and finally publishing the software product release for distribution (11, p. 12-13.). These processes will be described in more detail in the requirements and implementation sections.

3.3 Staging Environments

In software development, staging is an environment that is a mirror of or resembles closely with the original production environment. Following the best practices, the staging and production environments should be isolated from each other, although this is not always possible. The basic usage of a staging environment in development is that a certain new revision of the software or configurations will be deployed to the staging environment from version control for testing,

demonstration, or training purposes, before deciding on whether it is ready for production deployment. Version control strategies such as branching can allow easy management of multi-deployment configurations. (12.)

3.4 Automating Deployments and Monitoring

With multiple deployments such as staging and production environments, a large set of tools, configurations, and services need to be deployed. As this process is often complex and demands time, automating the deployment process saves resources and helps in finding problems earlier in the development cycle. (13, p. 1.)

One of the tools that is used in The Qt Company for automating deployments is Ansible by Red Hat, which allows to manage machine configurations in a simple text format (14). Configurations for the company pipeline infrastructure can then be committed, reviewed, and stored in the version control where they can be deployed automatically during maintenance. Manual configuration and deployment examples and other guides take place in the version control README files or the intranet wiki pages.

The monitoring of delivery pipeline infrastructure is also crucial for many reasons, including quality assurance of deployments and operational efficiencies. Monitoring allows detecting problems early in the delivery process. By catching issues early, it prevents small issues from escalating into larger, more costly situations. It therefore plays a role in maintaining the health of the SDLC by reducing errors and stress for the development team while increasing deployment flexibility. (15, p. 4, 17-21.)

By continuously monitoring the delivery pipeline, teams can also streamline processes in operations, identify bottlenecks, and optimize resource usage. This then leads to quicker and more efficient software releases. (16.) This is especially true for The Qt Company, which employs software monitoring tools such as Grafana (17) to create visualized dashboards from infrastructure real time data, with alerting configured for excessive resource usage or errors in pipeline infrastructure services allowing for various optimizations and quick resolution of problems.

4 REQUIREMENTS

The goal of the implementation work in the subsequent sections is to create an isolated environment in which the changes and improvements to tooling related to artifact exporting, packaging, and delivery automation could be tested without interrupting the releasing workflows in the production environment (R6). In this chapter, the requirements for this staging environment will be outlined in relation to the different stages in the pipeline, so that the results can be then evaluated against the targets which are set before the implementation. For an overview of the main requirements with their identifiers and descriptions, see table 1 on page 11.

Beginning with the version control, a new branching strategy must be proposed for the Git repositories containing the tools and configurations. Furthermore, the tools and configurations should reside in the same repository to keep them in sync better, and other benefits associated with the so-called “monorepos” (18). The deployment in production should keep the current main branches and separate branches can be created where needed for the staging environment implementation work and future use. It should be possible to deploy the latest configurations and tools directly from the branch in the version control (R2). A merging strategy is to be developed where the changes for the staging and production branches stay in sync periodically, with the staging containing all the latest changes and production containing the tested revision of changes and the latest configurations for release activities.

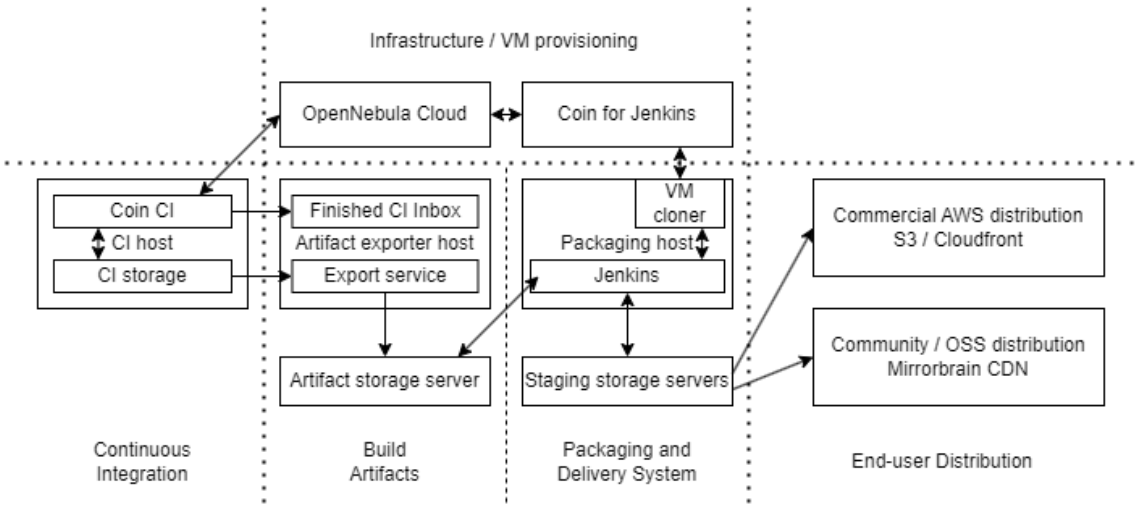

FIGURE 2. The architecture overview of the current pipeline infrastructure top-level components.

Next, it is important to define the infrastructure setup needed for the staging environment. The current pipeline architecture components can be seen in figure 2. To start with, to get the build artifacts into the staging environment, it was decided that a separate artifact exporter instance would be set up, but it was to use the same CI storage to save resources (R3). In addition to that, for the same reason, automatic exporting of build artifacts will be disabled where necessary, so the finished integration inbox will not be linked to the staging instance (R4). The packaging and delivery area also needs a separate Jenkins CI instance, which will be configured to use the same packaging and deploy configurations as in production (R1) but use the staging versions of the storage servers instead.

Furthermore, the virtual machine pool from OpenNebula as well as the provisioning configurations will be shared between the two, so to isolate the builds, the server addresses and access keys need to be reliably configurable if further network isolation is not possible to prevent leaking between the two deployments. A separate Coin for Jenkins instance will be deployed to request the cloud resources with different user credentials. (R6)

To offer more isolation, the other staging instance machine infrastructure as well as artifact and package storage servers are to be put into a separate network, controlled by firewall rules to allow only the required shared access to virtual machines and CI storage (R6). Release test automation (RTA) will test the content from the production environment, and the CI artifacts are the same in both environments, so triggering RTA tests will be disabled in the staging environment (R4). For final software packages in the staging environment, only dummy storage buckets and servers are to be created and used for testing delivery functionality with no external access.

During the staging environment implementation, Jira management tool (19) must be used to track the current progress with the tasks. For tooling changes, unit tests should be created and linters and commit hooks utilized for quality assurance. At the beginning, during the implementation, and after successful deployment, there must be meetings held with the release team and other stakeholders discussing the details of the staging environment and to provide training. Email and informal communication can be used as well, as a form of knowledge transfer, to keep everyone updated with the latest developments. Documentation should also be created for the new functionality and processes in intranet pages or version control README text files depending on which one is more accessible to the intended audience. The existing diagrams and guides are also to be updated to include the staging deployment details. (R5)

TABLE 1. Requirements overview

Requirement ID & Description	Acceptance Criteria	Source & Justification
R1: The staging environment must replicate the current production environment as closely as possible.	Use comparable configurations and versions for infrastructure deployment, virtual machine pools, automation systems, software packaging, and delivery methods.	Accurate and reliable simulation of the production environment, best practices. Maintaining and keeping duplicate configurations up to date is not feasible.
R2: The pipeline must support automated deployment from version control to both the staging and production environment during updates.	All necessary pipeline tools, configurations, and infrastructure successfully deployed automatically via Ansible or other existing CI/CD tools. Strategy defined for production updates.	Efficiency improvement with reduction of manual work. Follow the CI/CD best practices. Controlled production update mechanism.
R3: The staging must use the artifacts from the same integrations as production,	No staging Coin CI deployment. Artifact exports from shared CI storage.	Resource usage minimization. Coin CI system is already testable via existing test deployments. Shared release content tested only once by RTA.
R4: Automated pipeline flows enabled only as needed for testing purposes.	Separately configurable automatic exports. Automated triggering of timers and RTA tests disabled.	
R5: The staging pipeline must be documented	Tasks created in Jira project management tool (19), Availability of intranet documentation and README files, internal training	Progress tracking, knowledge transfer, onboarding, maintenance instructions
R6: Must be reasonably isolated from production and unnecessary outside access.	Separated network segments where possible. Verification of configuration overlap. Version control branching strategy.	Minimize interruption to production. Security best practices, data integrity, access controls.

5 IMPLEMENTATION

This section with its subsections focuses on the practical parts of the thesis work and the implementation of the necessary changes to the current tooling and configurations in preparation for the staging pipeline deployment. The CI infrastructure responsible team was also tasked with cloning the current Ansible production configurations for the staging pipeline counterpart machines with those changes not being mentioned here. One of the anticipated challenges was that the Ansible configurations might be incomplete as far as the current production deployment is concerned, since some of the environment set up has been done manually in the past.

5.1 Version Control System (VCS)

The version control system related process changes were the first step in establishing the proper development environment for implementing the changes required for the staging pipeline. Firstly, the focus was on the unification of two releasing team's single branch Git repositories containing the tools and configurations mainly used by the packaging and distribution stage of the pipeline. Before introducing more complex branching strategies, the scripts and other tools were moved to a single repository structure with the configurations, streamlining the process of making new features that would have required simultaneous changes to both repositories previously. The old leftover repository was archived to keep it still functional until the automation configurations were updated to only clone the new unified repository. Active contributors were notified of this change via email.

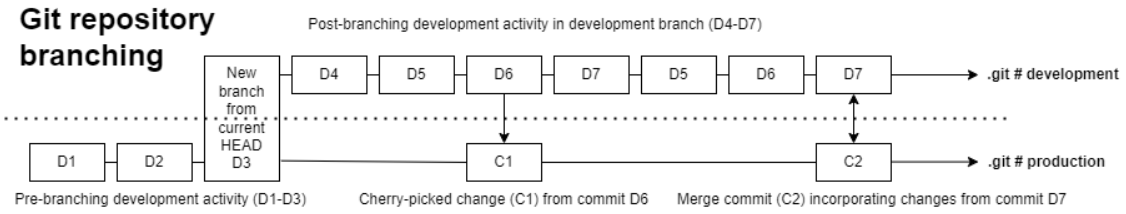

FIGURE 3. Git repository branching demonstration.

After the transition period to the unified repository was completed, a new development branch was added in addition to the existing main or production branch. This would be used initially to make the staging pipeline changes and later merged back once everything was complete. It was decided

that following the deployment of the staging pipeline, the process for new commits was to be changed so that all the changes would then go first to this development branch where they are then tested if necessary and when needed cherry-picked to the production in the case of hot fixes and other release critical activities. There would also be periodic production update cycles during the maintenance break schedule, where the development branch is verified to be stable by the release team and the new features that were not previously picked would be merged with the production branch. The new process and initial branching are illustrated in figure 3.

Although most of the Qt product repositories use a more elaborate release-based branching system as described on the “Branch Guidelines” Qt Wiki page (20), where every new product version will have a corresponding repository branch with its version number, this was seen unnecessary at this point with the releasing repository as there was already a tagging process in place where the repository state is tagged on new releases with the revision used to package the product. Therefore, this new process with the addition of a separate development branch was seen as sufficient also considering how the automation tool configurations were set up in the packaging Jenkins environment. The production Jenkins deployment would keep using the stable and tested production branch and the upcoming staging environment then default to the development branch with the latest features.

5.2 Artifact Exporter

For the staging releasing pipeline to work, some content for testing the packaging and delivery automation is necessary. For this purpose, the release team has a dedicated machine for getting the build artifacts from the CI system into the artifact storage server. The artifact exporter tool provides ways to either export the finished integrations automatically via a continuously running Linux systemd service (21) or to trigger a manual export job. As already discussed in the requirements section, it was decided that to keep the resource usage minimal and the content consistent with the production, a separate CI for the generation of the staging artifacts will not be used. Instead, the staging pipeline will share the same artifacts from the production integrations, and both instances of the artifact exporter services must be able to operate simultaneously connected on the same CI storage. The staging export will be done to a separate storage server with RSYNC tool which operates over SSH. See diagram in figure 4 below.

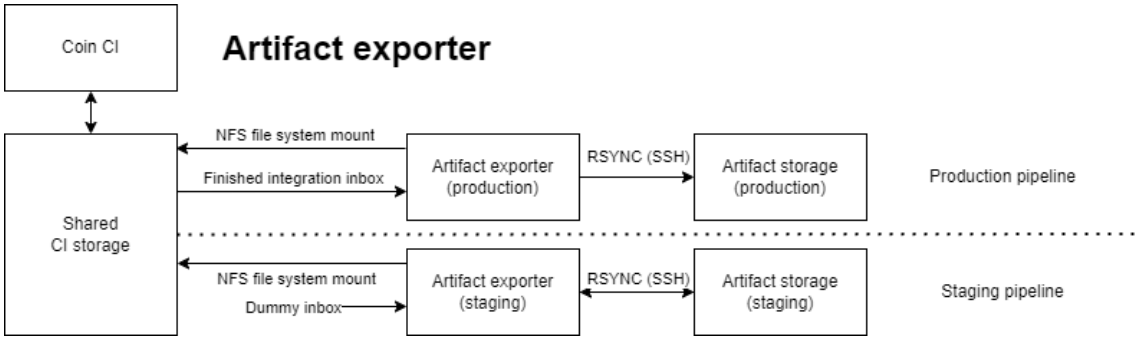

FIGURE 4. Artifact exporter pipelines sharing the mounted CI storage.

Mounting the artifact sources via Network File System (NFS) in two places does not in itself provide many challenges, but the automatic triggering of exports would, due to the implementation of the finished integration inbox, require the implementation of a task-sharing mechanism between the two exporter instances. After investigating the potential ways to implement this, it was decided that this was out of scope at this point. During the discussion, it was noted that the automatic or manual export functionality could easily be triggered manually when needed, and further resource savings could be seen by not running automatic exports as most of the exports would not be relevant for tests run in the staging pipeline.

The list of projects to automatically export artifacts from as well as the source and target storage server locations was specified in a single JSON configuration file in the version control. To better facilitate testing the automatic exports and getting the exported artifacts to the right place, the ability to specify the configuration file in the tool options was seen as necessary, so there could be a different configuration for the staging version of the exporter.

The artifact exporter service script is written in Python 3. For arguments passed to it from command line it utilizes the “argparse” library (22). The parser was modified to take an additional launch parameter “--service-config” (figure 5) to specify the configuration preset when the systemd service was launched (figure 6). When the service is started with this parameter, instead of using the production configuration, it will try to find the corresponding JSON configuration file from the directory and error out if no such configuration exists. The ability to filter automatic exports by projects could then be tested in the staging pipeline without modifying the configuration for the production exporter service. Also, when the manual export is triggered from the command line, additional configuration options, such as the correct export destination addresses could be set for exports with the staging counterparts which were documented to the intranet pages.

```
release_service.py > ...
    parser = argparse.ArgumentParser(...)
    ...
    parser.add_argument(
        "--service-config", dest="service_config_name", type=str, default="",
        help="Specify a custom configuration preset for the release export service, e.g. 'staging'."
    )
    ...
```

FIGURE 5: The Python 3 argparse argument definition for --service-config.

```
release-service.service
1   [Unit]
2   Description=Qt - Release Service
3
4   [Service]
5   …
6   ExecStart=/home/%u/.local/bin/pipenv run src/release_service.py --service-config=staging
7
8   [Install]
9   WantedBy=default.target
10
```

FIGURE 6: The Artifact Exporter tool systemd service file, with the added command line argument highlighted for the staging environment.

In addition to this artifact exporter tool script change, a new configuration was created in the version control for the staging environment with the planned server addresses and initial set of automated export definitions. The Ansible roles configuration was also modified to take the new launch arguments into account when installing and activating the exporter services in both the production and staging instances.

5.3 Jenkins CI/CD Deploy Configuration

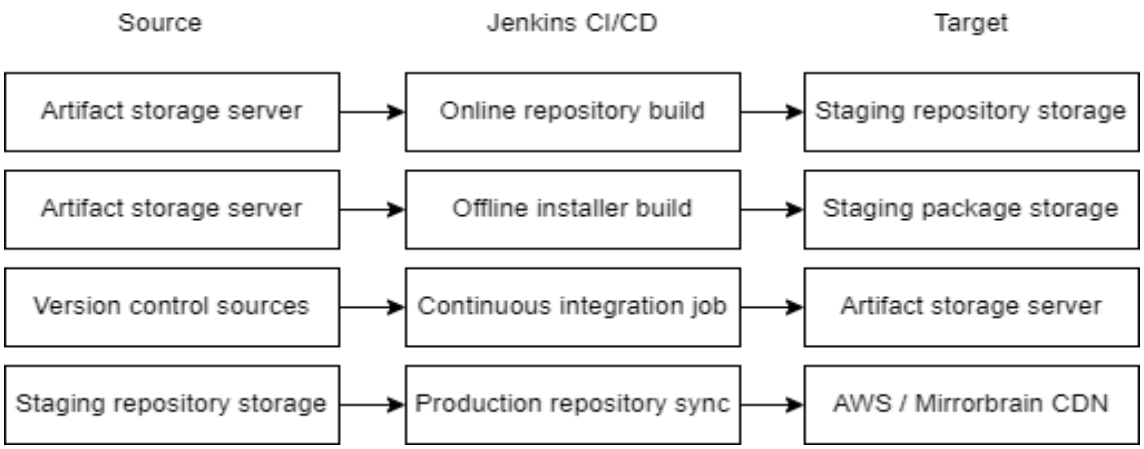

FIGURE 7. Common Jenkins integration, packaging, and delivery job flows.

The packaging and distribution stage of the pipeline uses Jenkins for automation. The different types of tasks run on Jenkins can be seen from figure 7. All these main pipeline flows must be tested in the staging environment, which meant that a separate staging instance would be deployed for the staging environment. It became apparent that Ansible had the configurations required for doing a plain Jenkins installation with some of the required plugins, but for example, the user matrix authentication set up was missing and was previously deployed manually in the production instance.

To resolve the need for user authentication, Jenkins must be connected to the company's Active Directory (AD) (23). Using the generated default credentials for the admin Jenkins user, it is possible to configure LDAP authentication and set the required permissions for the users in compliance with the company's security policies. The "need to know" principle applies where the employees are given only the necessary access to do their assigned tasks. In addition to the plugins, a simple Groovy script was created to be run as part of the Ansible roles to automatically configure the LDAP options via the Jenkins API if they were missing from the machine. There was no existing Ansible plugin found to achieve this, so the Jenkins script plugin was the easiest option.

Another thing missing from Ansible was the deployment of Jenkins jobs. The pipeline jobs were manually deployed to the production instance using a custom deploy script written in Python 3 that interacts with the Jenkins REST Application Programming Interface (API) (24). INI configurations and XML job templates used to generate the final Jenkins job XML configuration are stored in the packaging repository to allow for deployments from the version control system. The missing configurations that had job deployments in the current production Jenkins that were not present in version control were created.

Considering that even if running this job deploy script was added to Ansible in its current state, it would require some work as it was limited to deploying jobs one or few at the time from a single configuration file or a template. Furthermore, the server addresses for the Jenkins instance and repository branch names were hard-coded and scattered around the deploy configurations. For multi-deployment scenario, those should be configurable from a single place. Also, it is expected that there will be some jobs that are only relevant to the staging environment for testing purposes or vice versa, so that there must be a mechanism to decide which jobs belong to which Jenkins instance.

To satisfy the above requirements in addition to that the pipeline jobs shall be easily deployable in an automated manner, it was decided that a new top-level configuration file must be added for each Jenkins deployment which specifies all the jobs expected to be deployed for that instance. This will allow the deployment of all jobs quickly at once when needed, such as during a maintenance break, and it can be added to Ansible for automated deployment of jobs when running other updates. The configuration file will then also specify which repository and branch combinations to add to every Jenkins job and some other options such as the target Jenkins instance address URL. The YAML file format (25) was chosen for this purpose due to its simple human-readable syntax. See figure 8 for the initial example configuration of the top-level configuration file.

deploy_sample.yaml

```yaml
# Sample top-level deploy configuration for packaging Jenkins instance at localhost
jenkins_url: http://localhost:8080/  # Jenkins instance URL
disable_rta: true  # Whether to disable RTA triggers
disable_timer: true  # Whether to disable automated job timers
repositories: repository#branch#clonepath  # List of repositories to add to every job
deploy_jobs:  # List of deployable jobs for this instance
  qt-packaging-all.ini:  # Deploy all jobs from this INI configuration
    - "*"
  qt-packaging-including.ini:  # Deploy some jobs from this INI configuration
    - "Included_job_name_1"
    - "Included_job_name_2"
  qt-packaging-all-except.ini:  # Deploy all except some jobs from this INI configuration
    - "*"
    - "-Excluded_job_name1"

```

FIGURE 8: Top-level Jenkins configuration file example.

With the addition of the new top-level configuration files for both production and staging deployments, Jenkins URLs, duplicate repository definitions, and branch definitions were cleaned up from the configuration INI files and the original deploy script adapted to ask for this information via the CLI options. A new batch deploy script was added which parses the given top-level configuration file with the PyYAML library for Python and loops over the included jobs applying the options specified to the XML templates and deploys the updated job definitions via the Jenkins API.

Furthermore, new options were added to allow disabling of all specified RTA and timer triggers, which will be useful for the non-production deployments as per the requirements. Deploy script and INI configurations were also modified to support adding Jenkins jobs automatically to the specified views, which was not previously supported. Jenkins Access tokens are not shared between the production and staging deployments to allow for more isolation preventing accidental cross-

deployments and are not stored in the version control but are kept separately in a more secure place to follow the company's security policies.

5.4 Packaging Job Configuration

Next, it was time to adjust the packaging configurations and build scripts to allow for use in multiple deployments. Previously hard-coded server address values for artifact storage servers and distribution endpoints were replaced by aliases, which would be then configurable from a single place to point to the correct locations. They are applied at the start of the Jenkins job via a packaging bootstrap script that is aware whether the Jenkins agent is connected to a production, or a staging instance based on the "JENKINS_URL" environment value that Jenkins sets automatically (26). Therefore, even agents from the same virtual machine pool can be somewhat isolated from each other if the correct configuration is applied. Separate access keys and secrets were also created for use in the staging environment so that there is less chance of mix-up between the environments. (R6)

5.5 Delivery Job Configuration

Automated and manual delivery job configurations needed adjustments not to update production access control lists automatically during a sync job. As per the requirements, the distribution buckets hosted on AWS S3 service and storage servers deployed by the CI team would be made available with different names only internally so no separate access control would be required to access them other than that. There were hardcoded S3 bucket names that needed to be adjusted to be different for the staging environment, as well as new credentials needed to be stored for accessing those separately from the production environment.

5.6 Deployment

This subsection will go over the details on how the deployment of the staging environment machines was done to bring up the staging environment itself. The same CI infrastructure pool from OpenNebula already in use with the production environment was used to provide and provision the virtual machines to best replicate the pipeline conditions between the environments as per requirement R1.

The first step in the deployment process was the provisioning of virtualized hardware and the operating systems along with Ansible for the required test environment machines. Hardware with resources comparable to the existing production machines was chosen, although downscaled for now to accommodate the expected load for the test pipeline. The capabilities would then be scaled up in the future if deemed necessary, but it is good to start small to minimize the resource waste. A Jira ticket with the planned server specifications was assigned to the teams responsible for CI so that the resources could be reserved for the configuration of the systems.

In addition to the hardware and the deployment of operating system templates, the CI team configured the private networking in cooperation with the IT. A network with a different subnet, domains, and access keys was created for the test environment servers which were practically isolated from those used in the production deployment to avoid any potential damage from misconfigured server addresses in the pipelines. The security aspects such as firewall configurations were also set up according to the company policies, and the list of needed services such as the Jenkins, HTTP server (Apache, Nginx) proxies to web user interfaces (UIs), and Secure Shell (SSH) server software that will be running on the machines was requested to be added to the allowed list for the data communication inside the pipeline environment to be able to take place.

6 RESULTS AND VERIFICATION

After the staging environment machines were initially deployed, it was time to verify that all the services in the pipeline were accessible and functional. Initially there were some unexpected problems in general with the network and firewall configuration that delayed some functionality verifications but those were resolved with the IT team. In this section, the achievements regarding the requirements set before the implementation will be also evaluated. For example, considering that there was not much manual work required to bring up the staging environment after the missing configurations in Ansible and the Jenkins job deploy functionality were implemented, that was enough to meet the R2 requirement of automating future deployments.

6.1 Artifact Exporting Functionality

To begin at the start of the staging pipeline, artifact exporting functionality was tested first. There were some initial issues with getting the CI storage mounted as it was in a different network, but after a second network interface was connected to the staging exporter machine the exports could be tested. This is not ideal in terms of isolation, as the exporter machine will have access to the production CI network, but it was the most reasonable option considering that a separate CI would use up too many unnecessary resources without any real benefit, as decided with R3. Therefore, the R6 isolation requirement was determined to be fulfilled at this stage.

Other than some minor issues with configurations, the full export of Qt framework was tested to the staging artifact storage server and the added staging configuration preset appeared to function as expected. The exporter tool storage drive for temporary files also had to be expanded from the initially planned specifications, as a single export was able to take up the full space reserved for the data partition. Once the export testing was completed successfully, there appeared to be no differences between the exported artifacts from production and staging, so with R1 in mind the operation is identical in both environments. With the export service running, it was also verified that there were no automatic exports launched with the dummy inbox configuration on the staging environment as per the requirement R4.

6.2 Jenkins, VM Cloning, Pipeline Jobs

After the Jenkins deployment, the dashboard was accessible through the web interface. Some missing plugins noticed from the configuration panel were added to the Ansible configurations, and the batch deploy script was used to deploy the staging jobs from configurations. The next step was to ensure that the virtual machine cloning functionality was set up to be able to run jobs on agent nodes other than the built-in node (the Jenkins host machine).

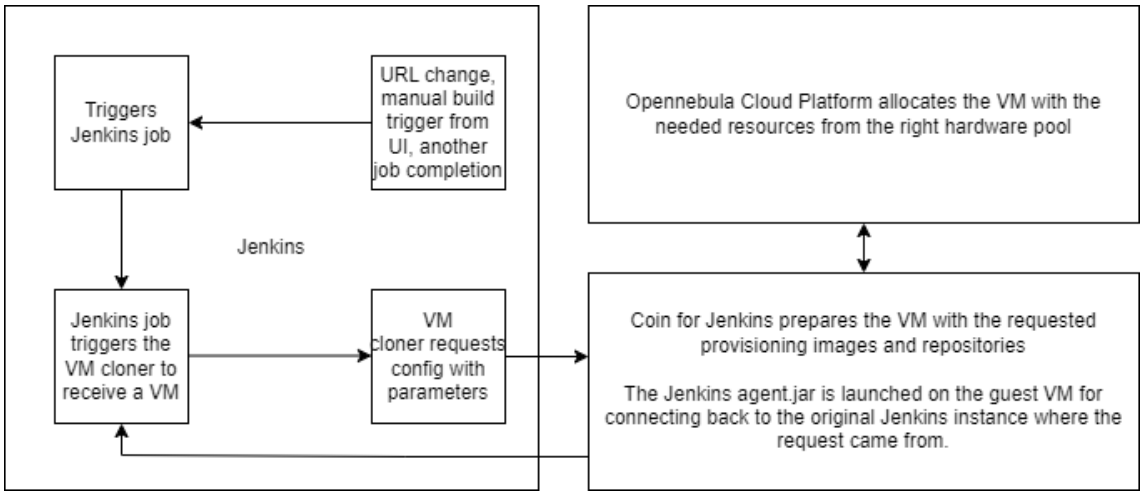

FIGURE 9. Overview of the Jenkins executors set up.

For this purpose, there is a custom implementation in place to integrate with the company's in-house built CI system Coin as shown in figure 9 above. When a job is launched on Jenkins, it will trigger another job which contacts a Coin instance (specifically deployed for Jenkins use) with the integration request containing information such as the desired machine configuration and the required additional Git repositories to be cloned from the version control, and then once the guest VM provisioning and allocation from OpenNebula cloud platform (27) is ready the Jenkins agent will be launched and connected to run the job.

Once the basic features of Jenkins were functional, more in-depth testing was done with the new configurations to ensure that all the continuous integration build jobs in Jenkins, installer and packaging jobs, and continuous delivery publishing tasks were able to be successfully run in the staging Jenkins environment without affecting the production deployment. This consisted of running test integrations for products that used Jenkins as CI such as Qt Creator and various repository and dummy production sync tasks for the Qt Framework and other products. With the main

packaging and delivery flows tested, it was verified that the staging Jenkins was able to operate independently from production (R6) with the same features functional (R1).

6.3 Production update

Once the staging pipeline functionality was tested, the development branch changes were scheduled to be merged back to the production repository. The date was chosen to be the next maintenance break when the production machines would be also updated with the latest Ansible changes. Despite a problem with the environment clean up between the Jenkins jobs ran in different deployments everything went rather smoothly and there were no big interruptions to the releases. Thus, the R2 requirement of automatic deployments were further verified with the updated production deployment in addition to the initial staging deployment done before that.

6.4 Getting Feedback

Although the process changes were announced via email, there was still some confusion about the new staging environment and processes with the development branch strategy. After a while of monitoring the situation, most of the team members got somewhat used to the new workflow and there were some valuable suggestions presented on how to enhance the proposed process as part of future process improvement work. For example, the use of release branching for the repositories was suggested, and there were multiple comments about the change cherry-picking process.

One of the highlighted things in the feedback was also that there should be more documentation specifically related to the usage of the staging environment and the production update process. In terms of updating existing documentation to include the staging environment details, however, the documentation requirement R5 would be sufficient, so it should be considered partially completed. Keeping this in mind, more work was required to improve the documentation pages on the intranet and communicate the new changes to all the team members. Future tasks were planned for this in the team backlog.

7 DISCUSSION

In conclusion, the implementation of the staging pipeline has marked a first step toward the pursuit of a more streamlined and reliable delivery automation development process in the release team. The ability to run builds separately from production within this isolated pipeline has already allowed the development and testing efforts to continue in the development branch, although it will still take some time for every member of the team to adopt the staging environment and the new branch workflow in their own tasks. I will be looking forward to continuing to work with the team getting everyone to the same page on the current state of the delivery automation pipeline deployments and to further improving the processes outside of this work.

Even after an extended period getting familiar, developing, and working with the individual tools in the delivery automation pipeline during the previous work I have completed for the company, there was still significant value gained from this exercise for the purposes of personal development and growth. The pipeline implementation process allowed me to deepen my knowledge about the releasing system in general, seeing how everything glues together and interacts in the bigger picture. It provided more elaborate insights in various individual components, too, that I was not necessarily aware of before looking into it further in the context of this work. This comprehensive understanding has enhanced my future abilities significantly in terms of troubleshooting the pipeline for problems and optimizing the workflow in the future.

The exercise was also helpful for the purpose of exploring the gaps in the current automated deployment tools and configurations, as it was clearly seen which parts of the system were not fully deployed by the Ansible roles and other scripts, since the automatic deployments were not always used from the start but developed afterwards alongside the growth of the company. Therefore, it was possible to improve on this aspect by implementing the missing blocks from the automation configurations. Having almost no prior experience with Ansible, this made it possible to also develop my skills in that area and familiarize myself with the Ansible role design principles which were not obvious from the start without reading up on the documentation and getting feedback from the other team members.

The most challenging aspects were that the existing internal documentation of deployments and who has access to what systems were not always easily accessible in the company intranet but

had to be assumed from the current production environment or by interviewing colleagues. Relying on outdated details, missing information, or inaccuracies with individual recollections made it more difficult to proceed as quickly as was envisioned. In addition, most of the work was completed during the holiday season, which meant that key personnel were not always available when needed, leaving the team with the resources we had to figure things out within the planned timeline.

That also unfortunately meant that the reviewing of changes was delayed by some time until the right people could give feedback on the implementation and further changes. With the tight release schedule, it took some time to finally implement the repository process changes and the execution of the initial production update strategy, so that it could be completed smoothly without many interruptions. Ultimately, the work and the results from it were spread over a much longer period than initially planned, although it was expected there would be some setbacks and challenges.

In essence, most of the target requirements set out before the implementation were achieved during the thesis work. However, considering the current state of the pipelines after the deployment, there were still areas that require further work. Despite the focus on manual pipeline testing use cases in the requirements of the initial deployment, defining a set of automated test jobs to be run in the staging environment could be one of those future improvements. This would allow the pipeline to continuously test and deploy the new changes to the delivery automation tooling as part of staging a change in the version control.

Another aspect to pay more attention to in the future is to add more isolation between the production and staging environments, mainly as the virtual machine pool is still currently shared, which could allow for leakage between the environments given an erroneous configuration value since both networks are accessible from the agent. The thing to learn from introducing isolation, however, is that it can distance the staging environment from the production, making it so that the configurations might not stay consistent with both. Thus, it is important to balance in between the isolation and the replicability of the test deployments.

Furthermore, it was noticed that the monitoring capabilities in the staging area are still very limited compared to the production environment. The Grafana dashboards providing overview of the production environment machines are not yet available for the staging machines although the alerts seem to be functional, although the Jira tasks were created to address this in the future. The same goes for the documentation related to the usage of the different pipelines in place, which needs to

be improved over time so that there is less training time needed to be spent when instructions could be easily available in the intranet wiki pages for quick reference.

REFERENCES

1. The Qt Company 2024. About the Qt Group – Code Less & Create More. Search date 25.2.2024. https://www.qt.io/group.
2. Thales Group 2021. Software Delivery Explained. Search date 13.5.2024. https://cpl.thalesgroup.com/software-monetization/software-delivery-explained.
3. Lachaume, Arnaud (Keypup) 2023. From Chaos to Consistency: How to Streamline Software Delivery for Seamless Results. Search date 13.5.2024. https://www.keypup.io/blog/mastering-software-delivery-strategy.
4. Google Cloud DevOps Research and Assessment (DORA) 2021. Accelerate State of DevOps Report. Search date 13.5.2024. https://services.google.com/fh/files/misc/state-of-devops-2021.pdf.
5. The Git Community 2024. Git. Search date 13.5.2024. https://git-scm.com/.
6. Qt Wiki 2024. Qt Contribution Guidelines. Search date 13.5.2024. https://wiki.qt.io/Qt_Contribution_Guidelines.
7. Google Open Source Projects 2024. Gerrit. Search date 13.5.2024. https://opensource.google/projects/gerrit.
8. The Qt Project 2024. Gerrit Code Review Dashboard (QtFork). Search date 13.5.2024. https://codereview.qt-project.org/.
9. Jenkins 2024. Jenkins User Documentation. Search date 13.5.2024. https://www.jenkins.io/doc/.
10. The Qt Company 2022. Introduction — Coin - Qt Continuous Integration 1.0.0 documentation. Search date 13.5.2024. https://testresults.qt.io/coin/doc/introduction.html.
11. Shahin, Mojtaba & Ali Babara, Muhammad & Zhu, Liming 2017. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. Search date 25.2.2024. https://arxiv.org/ftp/arxiv/papers/1703/1703.07019.pdf.
12. Disruptive Library Technology Jester 2006. Traditional Development/Integration/Staging/Production Practice for Software Development. Search date 25.2.2024. https://dltj.org/article/software-development-practice/.
13. Humble, Jez & Read, Chris & North, Dan 2011. The Deployment Production Line. Search date 25.2.2024. https://continuousdelivery.com/wp-content/uploads/2011/04/deployment_production_line.pdf.

14. Ansible 2024. How it works. Search date 25.2.2024. https://www.ansible.com/overview/how-ansible-works.
15. Humble, Jez & Farley, David 2010. Reliable Software Releases Through Build, Test And Deployment Automation. ISBN 978-0-321-60191-9.
16. Forsgren, Nicole & Humble, Jez & Kim, Gene 2018. Accelerate: The Science of Lean Software and DevOps. ISBN 978-194278379.
17. Grafana Labs 2024. Get started with Grafana Open Source – Grafana Documentation. Search date 13.5.2024. https://grafana.com/docs/grafana/latest/getting-started/.
18. Narwhal Technologies Inc 2022. Monorepo explained. Search date 13.5.2024. https://monorepo.tools/.
19. Atlassian 2024. Jira | Issue & Project Tracking Software | Atlassian. Search date 13.5.2024. https://www.atlassian.com/software/jira.
20. Qt Wiki 2024. Branch guidelines. Search date: 13.5.2024. https://wiki.qt.io/Branch_Guidelines.
21. Systemd 2023. System and Service Manager. Search date: 13.5.2024. https://systemd.io/.
22. Python Software Foundation 2024. argparse — Parser for command-line options, arguments and sub-commands. Search date: 13.5.2024. https://docs.python.org/3/library/argparse.html.
23. Jenkins 2024. Jenkins LDAP Plugin. Search date: 13.5.2024. https://plugins.jenkins.io/ldap/.
24. Jenkins 2024. Remote Access API. Search date: 13.5.2024. https://www.jenkins.io/doc/book/using/remote-access-api/.
25. The Official YAML Web Site. Search date: 13.5.2024. https://yaml.org/.
26. Jenkins 2024. Using environment variables. Search date 13.5.2024. https://www.jenkins.io/doc/book/pipeline/jenkinsfile/#using-environment-variables.
27. OpenNebula Systems 2023. OpenNebula – The Open Source Cloud & Edge Computing Platform. Search date 13.5.2024. https://opennebula.io/.