BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

AUTOMATED TESTING ENVIRONMENT

PROSTŘEDÍ PRO AUTOMATIZOVANÉ TESTOVÁNÍ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **ALEKSANDR VEREVKIN**
AUTOR PRÁCE

SUPERVISOR **Mgr. KAMIL MALINKA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

148367

Institut:	Department of Intelligent Systems (UITS)
Student:	**Verevkin Aleksandr**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Automated Testing Environment**
Category:	Security
Academic year:	2022/23

Assignment:

1. Study the software testing methodology.
2. Learn methods and technologies to support automated testing and automated infrastructure deployment.
3. Learn about Apache Airflow, a workflow management platform for data engineering. Compare it to other available tools in the context of using them to create automated testing environments.
4. Design a framework for automated testbed deployment and test execution over a product that supports selected environment configuration parameters such as cluster version, other additional configurations, running services, test re-run and evaluation, and ability to test another product.
5. Implement and test the proposed solution.

Literature:
OWASP Web Security Testing Guide

Requirements for the semestral defence:
Items 1 to 4

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Malinka Kamil, Mgr., Ph.D.**
Consultant:	Jaroš Miroslav (RedHat)
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	10.5.2023
Approval date:	3.11.2022

Abstract

This work deals with the task of setting up and deploying an environment for software testing. The main objective is to simplify and automate this process. The chosen problem is intended to be solved with the help of existing tools for workflow automation. Specifically, with the Apache Airflow, a workflow management platform for data engineering pipelines. The contribution of this work is in the study of existing workflow automation tools and the creation of a framework for automatic deployment of infrastructure.

Abstrakt

Tato práce se zabývá úkolem nastavení prostředí pro testování softwaru. Hlavním cílem je tento proces zjednodušit a zautomatizovat. Zvolený problém je určen k řešení pomocí stávajících nástrojů pro automatizaci workflow. Konkrétně, za pomocí Apache Airflow, platformou pro správu pracovních toků pro datové inženýrské pipeliny. Přínos této práce je ve studiu existujících nástrojů pro automatizaci pracovních postupů a vytvoření frameworku pro automatické nasazování infrastruktury.

Keywords

Automated software testing, testing environment, automation, automation tools, infrastructure, workflow, pipeline, framework, Apache Airflow, DAG.

Klíčová slova

Automatizované testování softwaru, testovací prostředí, automatizace, automatizační nástroje, infrastruktura, workflow, pipeline, framework, Apache Airflow, DAG.

Reference

VEREVKIN, Aleksandr. *Automated Testing Environment.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Mgr. Kamil Malinka, Ph.D.

Rozšířený abstrakt

Hlavním cílem této práce je automatizovat akce, které softwaroví testeři ručně dělají ze dne na den. Konkrétně nastavení požadovaného prostředí pro testování produktu.

Tuto práci iniciovala softwarová společnost *Red Hat* a konkrétně tým zajištění kvality produktu *3scale API Management*. Pro každou z kombinací pro testované prostředí, musí tento tým přeinstalovat a překonfigurovat existující prostředí. Tento proces komplikují omezené zdroje dostupné pro instalaci. Kombinaci prostředí zahrnují verze clusteru, databázi, další konfiguraci, externí služby a varianty samotného produktu pro testování.

Pro výběr správného nástroju pro tvorbu automatického frameworku, byl proveden průzkum existujících přístupů a nástrojů pro automatizaci workflow. Během průzkumu byly nastudované nástroje jako Tekton, GitHub Actions, GitLab CI/CD, ArgoCD a Luigi. V důsledku, byl vybrán ten nejvhodnější pro stávající problém. Zvoleným nástrojem stál Apache Airflow, platforma pro správu pracovních toků pro datové inženýrské pipeliny.

Apache Airflow byl vybrán kvůli svému zaměření na vytvoření závislostí jeho úkolů pomocí konceptu DAG. DAG je Orientovaný Acyklický Graf, takový graf nemá cyklické závislosti a je jednosměrný. Tento koncept pomáhá vytvořit workflow takovým způsobem, že definované úlohy lze přeskočit v závislosti na přijatých dynamických informacích. Úlohy v Airflow jsou navíc sestavované pomocí programovacího jazyka Python. Python je jedním z nejpopulárnějších jazyků a jeho syntax je považovaný za snadno srozumitelný i pro člověka, který se nikdy nezabýval programováním. Kromě toho, Airflow má několik dalších výhod, které mohou výrazně pomoci při vytváření požadovaného frameworku. Mezi tyto výhody patří flexibilní plánování úloh Airflow, informativní grafické rozhraní a nativní integrace s velkým počtem externích služeb.

Za pomocí zvoleného nástroje byl navržen a implementován framework pro automatizaci nasazení infrastruktury pro testování. Framework reprezentuje pipelinu se skripty a nástroji k vytvoření požadovaného prostředí. Části této pipeliny se nazývají moduly a lze je flexibilně konfigurovat. Dva požadované moduly pro využití plné funkčnosti frameworku jsou modul pro instalaci clusteru a modul pro nasazení produktu do existujícího clusteru.

Všechny moduly a další nastavení lze konfigurovat pomocí jednoho souboru. Tento přístup tváří jediné rozhraní pro všechny možné konfigurace workflow.

Jednou z funkcí frameworku je správa dostupných zdrojů. Pokud framework zjistí, že požadovaná konfigurace clusteru nebo produktu již existuje, bude část pracovního postupu přeskočena, čímž se ušetří čas instalace a prostředky úložiště, které lze použít pro další, paralelní testování.

Framework je také schopen vypočítat zdroje potřebné k instalaci clusteru a upozornit uživatele, pokud instalace překročí nastavený limit.

Na konci požadovaného pracovního postupu je uživatel schopen spustit testy na nainstalovaném nebo nalezeném prostředí. Framework také obsahuje flexibilní konfiguraci pro testování. Jednou z těchto konfigurací je zaslání zprávy a výsledků testů po jejich provedení na nastavenou e-mailovou adresu.

Řešení bylo důkladně otestováno týmem 3scale a všechny zjištěné chyby byly opraveny. Testování probíhalo spuštěním frameworku v různých kombinacích konfigurací. Byla testována flexibilita frameworku a jeho možné přizpůsobení jiným produktům a workflow.

Výsledkem této diplomové práci bylo vytvoření automatického frameworku pro instalaci infrastruktury pro testovací proces. Tento framework se bude vyvíjet a rozšiřovat ještě po dlouhou dobu a poskytovat týmu 3scale a dalším týmům zajištění kvality snadný způsob automatizace jejich pracovních postupů.

Automated Testing Environment

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mgr. Kamil Malinka, Ph.D. The supplementary information was provided by Miroslav Jaroš. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Aleksandr Verevkin
May 10, 2023

Acknowledgements

I would like to thank Mgr. Kamil Malinka, Ph.D. for his assistance in this work creation. Thanks to Miroslav Jaroš, who was my external consultant from the company Red Hat. Also, my thanks belong to my team supervisor Bc. Filip Čáp for his detailed analysis of the problem and the suggestions on how to solve it.

Contents

Chapter 1

Introduction

Software testing is an important part of the development life cycle of every application. The purpose of software testing is to make sure that the product not only was correctly programmed but also that a functioning product has been created.

With the development of software engineering, a more systematic approach to the testing of products has begun. Methodologies, guidelines, and recommendations were created, testing tools began to be developed. Since then, testing has seen progressive development — currently, there are many testing tools and frameworks that offer advanced features for the regular launch and execution of tests.

Despite that, many software companies continue to rely on manual testing. Because of this, employing a designated Quality Assurance team is a standard procedure for software organizations. This team is in charge of eliminating or at the very least reducing bugs in the software products by continuously testing new product builds and reporting back to the engineers about the results.

A standard part of their work is setting up a testing environment, the environment in which tests will be run and evaluated. The team will manually install the system, set up configurations and additional services for the execution of the following tests. After test evaluation, it is required to clean up this environment in favor of the next one, since storage resources are not infinite and must be continuously iterated. What if the team has dozens of testing configurations to test? What if there are hundreds of them?

This necessary process is unreliable as people tend to make unintentional errors or, for example, may skip the important step. And it also takes valuable time that could be spent on more sensitive and important tasks. Automation can greatly improve the overall efficiency of the Quality Assurance team. This work intends to solve this problem.

As a result of this thesis, will be created a collection of automatic scripts, upon execution of which, the environment will be automatically installed, set up, and configured, depending on the parameters specified at startup, so all that remains is to run the tests and evaluate the results.

The pipeline automation industry offers a wide variety of workflow tools. It might be challenging to select the right one for your needs. I've found that there aren't many in-depth resources out there that compare and analyze these instruments, making it harder for developers to decide which ones are ideal for their projects. After going through this work, readers should be able to tell which tools are suitable for their automated environment needs.

In the chapter 2 the problem that this thesis attempts to solve will be analyzed. After that, chapter 3 and chapter 4 will go over what exactly software testing is and how to

simplify and automate this necessary process. The focus of chapter 5 is on Apache Airflow, a workflow management platform for data engineering pipelines, and its features. In the next chapter, a solution for the existing problem will be proposed and, consequently, implemented in chapter 7.

Chapter 2

Problem analysis

This work is aimed at automating the tasks that Quality Assurance teams must carry out from day to day in order to conduct tests on company products. These tasks include setting up and configuring a suitable testing environment, installing and configuring the object for testing, appropriately running tests, and presenting the results. The purpose of this work is not the implementation of tools for individual components set up, but the creation of the connection between these tools and making from this an automatic framework, where these tools can be sequentially carried out and where the information from single components can be automatically transferred to others.

Successful testing of a product requires an existing cluster with all the necessary resources on it. In addition to the cluster and its configuration, these resources include external services and the tested product itself.

At the moment, a whole workflow of utility execution for the installation and configuration of the desired environment is executed manually. This means that developer time and free storage resources must be allocated for each of the testing processes.

One of the required features of an implemented framework is resource management. For example, if you need to test a product on a specific version of the cluster and an available for use cluster with the desired version and the installed product on it already exists, then the framework will need to skip the installation of already existing resources and immediately jump to the test execution. This feature will automate cloud space management, reducing the time and effort required for testing.

2.1 Focus product

Red Hat is an American software company that provides open-source software for enterprises.[1] The company specializes in providing solutions to software challenges faced by companies around the world.

One of the products, developed and maintained by *Red Hat*, is a ***3scale API management***.[2] API management is an essential component of a successful software company, as it helps deliver secure, scalable, and monetizable APIs to your customers and partners. It also helps to provide integration between internal services, by bringing a single, manageable interface for businesses and developers.

[1] https://www.redhat.com.
[2] https://www.3scale.net.

Specifically, in this work, attention will mainly focus on the *3scale API management* and the tests that a special Quality Assurance team should conduct on this tool daily. Despite the focus of work on one product, the original framework should be able to be applied to any other product testing environment with minimal changes and thus save valuable time for developers on a daily basis.

2.2 Testing environment

At the beginning of the existing testing pipeline, the open-source tool **Osia** is used to install the cluster. This tool provides a simplified command line interface for the setup of cluster installation script, transferring the creation of an installation template into parsing parameters to the utility input.[3] Besides that, Osia also makes some availability monitoring of DNS and cloud providers during the installation. These providers, at the moment, can be *Route 53* and *NSUpdate*, and *OpenStack* and *AWS* accordingly.

Osia requires a repository with resources that will contain the metadata of already installed clusters, created by the cluster-installation script and partly by Osia itself, and additional resources for the installation, such as internal certificates, allocated domains, and cloud configurations [11]. This tool is used by the *3scale* team and at least a few other Quality Engineering teams to simplify the management of their clusters.

Our clusters resource repository is named **cluster-management**. This name will be later used to refer to this repository. The cluster-management repository and Osia together represent the root of the cluster setup pipeline.

The **free-deployer** CLI tool is used to install the product. Free-deployer is a tool created and used by the *3scale* team to manage product deployment configuration on a cluster. This tool uses YAML configuration files to create templates for installation. These files may overlay and be inherited from each other, creating a flexible environment for installation configurations. Just like Osia, free-deployer expects the user to create a separate repository with these configuration files. In this repository, all the necessary resources can be hidden from unwanted eyes. This tool is in the last stage of its development and intends to become open-source in the near future.

The storage repository for *3scale* configurations is called **free-deployer-workplace** and further will be referred to as an element for installing and configuring the product on a cluster.

Osia does not provide an installed cluster configuration. Configuration includes such necessary things as creating user accounts and granting them the appropriate permissions, adding TLS certificates, setting up workload monitoring and automatic scaling of pods on the cluster, and adding the required *ImageContentSourcePolicy*[4]. To do this, the *3scale* team uses the scripts they created that are located in the repository called **cluster-setup-tools**. Scripts require their execution immediately after clean cluster installation.

As well as the script for cluster configuration, *3scale* has utilities for installing external services for the product on the cluster. These services must be installed prior to the actual product installation so that the product can detect them on the cluster automatically. Services include *MinIO* and *Redis* storage, a *RedHat SSO* provider, and some tools to make the testsuite work properly, such as *Mockserver* and *Httpbin*. These services are applied

[3] https://redhat-cop.github.io/osia/usage/install.html

[4] https://docs.openshift.com/container-platform/4.12/rest_api/operator_apis/imagecontentsourcepolicy-operator-openshift-io-v1alpha1.html

using the Kubernetes manifests and the scripts, that are residing under the **testsuite-tools** repository.

Scripts for setting up clusters and additional services are not required for a simplified installation of the product on a cluster and can be replaced with your own.

Tests are executed using an according job in *Jenkins*. Jenkins is an automation server, which can be used for the automation of several tasks, including building, deploying, and testing software [5]. In the testing job, for standard execution, it is enough to provide the API URL of the used cluster and the name of the project in which the target product is located. The execution of tests is the final part of the pipeline and is carried out after the successful installation and configuration of all the necessary environment elements.

Chapter 3

Software testing

Testing is an important part of every software development life cycle. It can be defined as the process of executing a program with the intent of finding errors [12]. Testing ensures that the software not only does what is defined in the functional specifications but also doesn't do anything that is not mentioned in any of them.

The first and most significant task of testing is not bug detection but bug prevention. A prevented bug is better than a corrected bug, because there is no time spent on searching and code rewriting [8]. Moreover, there is no need for test re-execution to check if the bug was properly corrected. To accomplish that, tests should be properly designed.

Test design is the technique of "how" testing should be done. It touches on such testing topics as what parts of the implementation should be tested, how comprehensive the tests should be, the initial data for the testing, and many others. The question of design should be raised before the implementation of the tests itself. Well-thought-out test design may save hundreds of hours of debugging and troubleshooting. And, of course, it is a part of the final product on one level with implementation.

In this chapter, I will closely look at the reasons to test the software and what challenges could come with it. I will analyze the best practices and how testing could be automated to not get tired of doing it on a permanent basis.

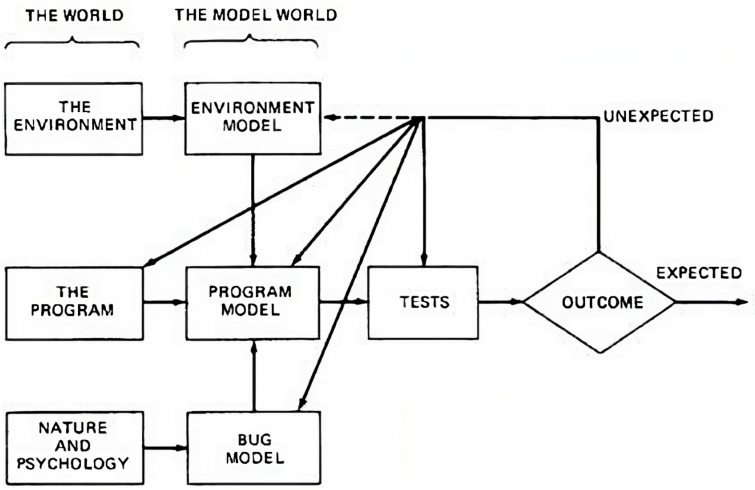

Figure 3.1: A Model of Testing [8].

3.1 Software testing motivation

Why testing software is important? Can't we just trust the engineers on their work and straightforward release the product? The cost of fixing an unintended bug in each developing phase can increase logarithmically [8]. In a small software company it could be a company-ending mistake and in some cases could even cost somebody a life. Figure 3.2 shows how the cost of the bug fixing can grow over developing time.

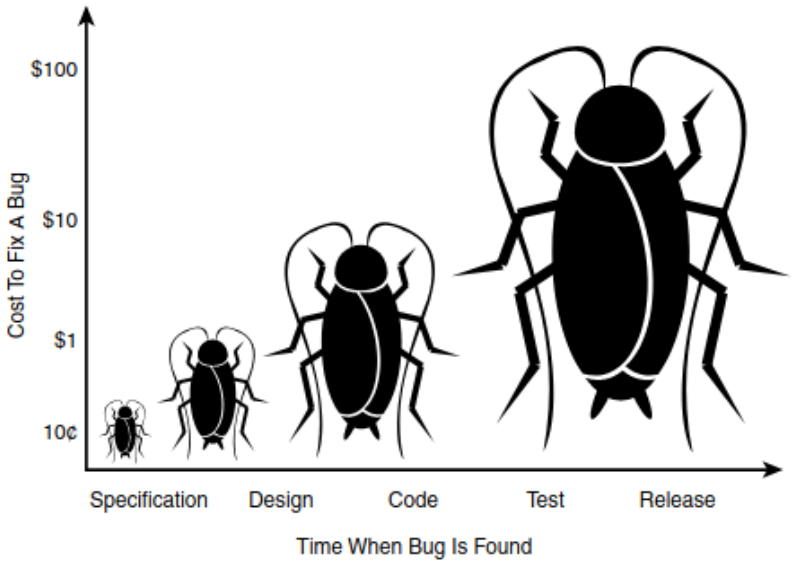

Figure 3.2: Bug fixing time-cost relationship [8].

Testing of software largely contributes to the improvement of software quality and can be measured by the amount of found defects. But software quality not only comprises the elimination of code errors, that has been found during the testing, but also, things such as maintainability and usability of the software product [14]. According to the *ISO/IEC Standard 25010 [ISO 25010]*, the following factors also belong to software quality (figure 3.3), therefore should be tested [1]:

- **Functionality** – The software's capability to offer functionalities that fulfill stated and implied demands when utilized under specified configurations.
- **Reliability** – The software's capability to keep up a certain level of performance when utilized under specified configurations.
- **Usability** – The software's capability to be understood, learned, and used, when utilized under certain configurations.
- **Efficiency** – The software's capability to offer acceptable performance in relation to the number of resources used, when utilized under certain configurations
- **Maintainability** – The software's capability to be modified. Corrections, upgrades, or adaptation of the product to changes in the environment are examples of modifications.

- **Portability** – The software's capability to be moved from one environment to another.

Appropriate tests that confirm compliance with these standards are essential.

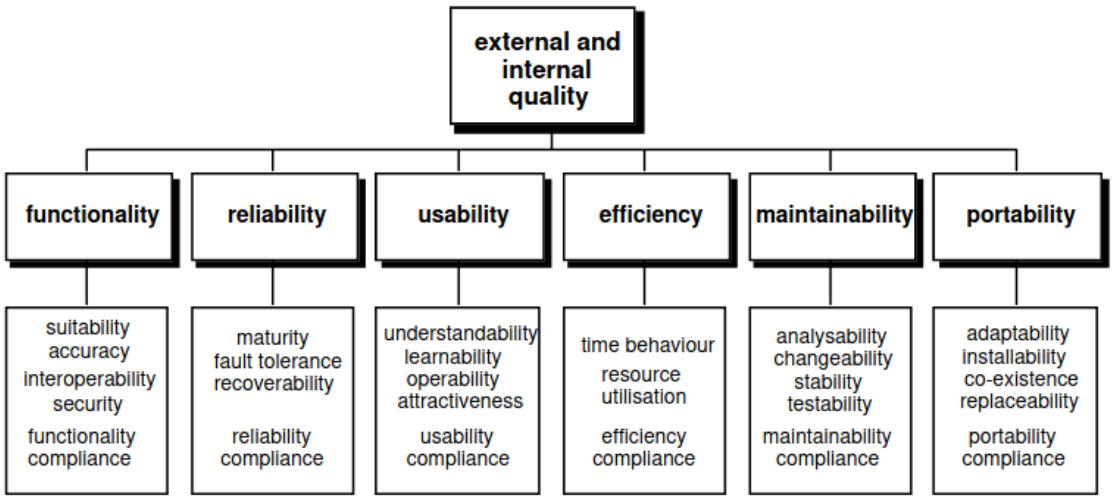

Figure 3.3: Quality model for external and internal quality [1].

Testing is the key to a product's success. It is the necessary procedure, that is performed in any company and on any type of software. Of course, tests can't help to get rid of all of the errors in the software. To accomplish this, we would need to test the program execution with every possible input, in any possible way – which is impossible based on the infinite number of test cases. The goal of software testing is to minimize the number of bugs in the output product as soon as possible so that new bugs cannot be created on existing ones and so that quality software can be made.

Nowadays there are a lot of well-known approaches and techniques to software testing. Everything has already been studied and examined, so the only thing left for us is to look at existing methods and choose what will be more effective in our case. The well-chosen testing techniques can save millions for the company.

3.2 Testing automation

Testing process automation is an important part of software testing. You don't want to spend your time on repetitive tasks. Automation refers to the use of automated frameworks and tools to set up, run tests and automatically evaluate the results. It involves creating scripts and utilities to simulate user actions and comparing the outcome with the expected one. Testing automation increases productivity in software development by reducing manual steps in the necessary process.

There are several advantages of test automation. The most significant of them are [13]:

- **Speed**. With the fast pace of the new product builds and agile methods of testing, there is no time to manually go through each application functionality to check if everything is working correctly. It will be much easier and less time-consuming to just start the automation job and evaluate the testing results.

- **Efficiency**. If those simple and repetitive tasks will be automated, then there are more important tasks to spend your time on. Like improving testsuite with new test cases or reporting already found bugs to the engineering team.
- **Accuracy and Relentlessness**. Tests can run outside the normal working hours – during the night or even during the weekends, in case of more extensive testsuite. And they will still be more accurate than a tester, whose attention intends to wane and mistakes to grow over time.

3.3 Testing environment

In the context of software engineering, the term *program's environment* is the software and hardware required to make it run [8]. Respectively, *testing environment* is the computerware that is required to run the tests on the application, including all the additional programs, services, and utilities that are used in the program which intended to be tested or are required to run the tests themselves.

Testing environment refers to the infrastructure in which the tests need to be performed and its configuration. Correct deployment of the testing environment is critical to accurate and efficient results.

Several factors affect the complexity of the environment setup. These include hardware or its virtualization, various databases and external services used, networking setup, and additional software dependencies. Added to this is an effort to avoid misconfigurations, version conflicts, and simple human errors. This deployment, when done manually, can be a serious challenge for Quality Assurance teams, wasting precious time and reducing their effectiveness.

In order to optimize the necessary process of installing the environment for tests to run on, teams incline to automate it.

For automation, usually, teams create their scripts and additional utilities for specific cases. In order to cover a larger configuration coverage and connect several required steps, it is possible to use tools, specifically created for this process.

Chapter 4

Tools for workflow automation

There are a lot of existing tools for the automation of environment deployment and new ones come out almost every day. Each with its own strengths and weaknesses. Despite this, all of them can be divided into several sections. In most cases, automation will require some combination of these tools [7].

Name	Originated at[a]	Workflows defined in	Written in	Scheduling	Backfilling	User interface[b]	Installation platform	Horizontally scalable
Airflow	Airbnb	Python	Python	Yes	Yes	Yes	Anywhere	Yes
Argo	Applatix	YAML	Go	Third party[c]		Yes	Kubernetes	Yes
Azkaban	LinkedIn	YAML	Java	Yes	No	Yes	Anywhere	
Conductor	Netflix	JSON	Java	No		Yes	Anywhere	Yes
Luigi	Spotify	Python	Python	No	Yes	Yes	Anywhere	Yes
Make		Custom DSL	C	No	No	No	Anywhere	No
Metaflow	Netflix	Python	Python	No		No	Anywhere	Yes
Nifi	NSA	UI	Java	Yes	No	Yes	Anywhere	Yes
Oozie		XML	Java	Yes	Yes	Yes	Hadoop	Yes

Figure 4.1: Overview of several well-known workflow managers and their key characteristics [10].

The goal of this chapter is to review the most popular of tools for the automatic environment deployment and to tell what situation each tool is suitable for.

4.1 Argo CD

Argo CD is an open-source, community-developed Continuous Delivery tool, used for the automation of applications management on Kubernetes clusters. Argo CD follows a declarative GitOps approach for the maintaining of desired application state, which means that it's using a Git repository as the only source of truth, and every change in it is instantly propagated to the Kubernetes cluster.

Typical Argo CD workflow starts with pushing configuration changes into the dedicated git repository. That triggers Argo CD webhook and starts synchronization process of cluster state and the new configuration. Kubernetes manifests are generated based on the new desired state of the application. If there are no differences between the new configuration and the current state of the deployment, then Argo CD will do nothing and will end the synchronization. If there are differences between states, new manifests are applied to the cluster. The process of cluster synchronization can be monitored in the graphical interface or by using Argo CD's API.

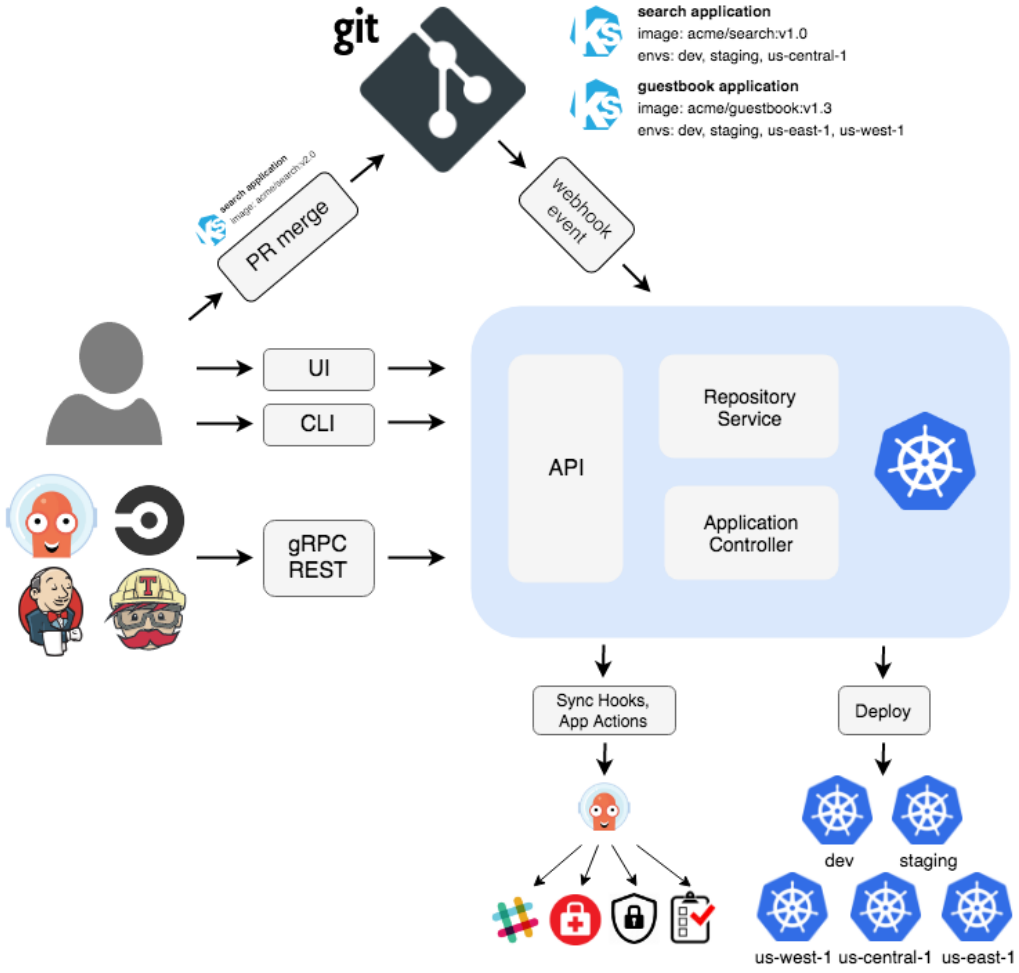

Figure 4.2: ArgoCD architecture [3].

When compared to other tools for automating workflows, such as Jenkins, Tekton, and Azkaban, Argo CD has the following advantages:

- **Scalability**. Argo CD can support multiple clusters and application handling at the same time. Argo CD also provides multi-tenant cluster architecture which allows for more granular access to the integration and authorization.

- **Declarative approach**. As stated previously, Argo CD uses a declarative GitOps approach, which can provide application versioning, instant deployment, changes withdrawal and review.
- **Extensibility**. Despite the Kubernetes-only deployment, Argo CD also can be extended with custom user plugins and integrated with other CI/CD tools like Jenkins and GitHub actions.
- **Visualization**. Dedicated UI is distinguished by its user-friendly approach to pipeline monitoring. Which makes it very easy to handle deployments, access control, and integration with other tools.
- **Kubernetes-native**. Argo CD has native integration with Kubernetes-based applications.

Argo CD disadvantages:

- **Complexity**. For users who are not familiar with Kubernetes and GitOps, it can be difficult to set up and configure the workflow.
- **Limitations**. Designed only for Kubernetes-based deployments. Argo CD doesn't support other types of applications.

4.2 GitHub and GitLab

GitHub and *GitLab* are popular code hosting and version control platforms. Not many know that these two are also offering their Continuous Integration/Continuous Deployment services. These services are called *GitHub Actions* and *GitLab CI/CD* respectively. Enabling developers to design their own workflow pipelines by configuring them in YAML format, also supporting separate pipelines for various deployments.

Usually, workflows are set to trigger on any interaction with the repository such as pull request, commit, or new release. But also can be triggered manually or on the schedule. Both services provide retry functionality on task failure and workflow results reporting.

For the creation of a typical GitHub Actions workflow, you will need to create a job file in the `.github/workflows` subdirectory. The job consists of one or more predefined steps. Jobs can run in parallel by default and sequentially by defining job prerequisites. Developers can use pre-build actions from GitHub Marketplace, such as image building or report generation.

Example of the standard GitHub Actions workflow. Workflow-defining jobs for setting up and running commit acceptance tests on every new code pull request to the repository's main branch.

```
name: acceptance
on:
  pull_request:
    branches: [ main ]
jobs:
  check:
    runs-on: ubuntu-latest
    steps:
```

```
      - uses: actions/checkout@v3
      - name: install pipenv
        run: pip install pipenv
      - name: setup pipenv
        uses: actions/setup-python@v4
        with:
          python-version: "3.11"
      - name: install dependencies
        run: pipenv install --dev
      - name: run acceptance check
        run: make commit-acceptance
```

For the GitLab workflow setup, serves single `.gitlab-ci.yaml` file, located in the repository root. This file defines the jobs to be run and the conditions under which they should be run. Jobs also consist of one or multiple tasks – scripts. In addition, you can define external dependencies for your jobs called services. Services can be a database or a server for cache.

Same commit acceptance workflow using the GitLab CI:

```
stages:
  - build
  - test

variables:
  # variable with python version for pipenv
  PIPENV_PYTHON_VERSION: "3.11"

before_script:
  # install pipenv
  - apt-get update && apt-get install -y python3-pip
  - pip3 install pipenv

build:
  stage: build
  script:
    # install dependencies
    - pipenv install --dev

test:
  stage: test
  script:
    # run acceptance check
    - make commit-acceptance

# run only on merge requests to the main branch
only:
  - merge_requests
  - main
```

Both GitHub and GitLab workflow services offer robust and expandable functionality. But they do have some disadvantages compared to other workflow tools. One of them is the limitations of customization. For example, a limited caching feature which allows storing only build artifacts and dependencies. Other tools often provide more advanced features such as Kubernetes support for deploying containerized applications or multi-stage conditional logic.

While both have free tiers for their services, the cost can quickly add up if you planing to fully move your workflow on one of these tools. Compared to other open-source projects such as Jenkins or Apache Airflow whose instances can be run on personal machines.

4.3 Tekton

Another interesting and relatively new Kubernetes-oriented continuous integration and delivery tool is *Tekton*. Tekton follows a declarative approach to workflow definition by providing reusable blocks to build, deploy and test your applications in a consistent way

After installing Tekton on the cluster, which can be done by using Kubernetes operators, YAML manifests or Helm charts, you can create a pipeline with your workflow. A pipeline can consist of various components such as tasks, triggers, and workspaces. Tasks for Tekton are defined in YAML or JSON format, making them easy to create even for non-programmers.

In the listing 4.1 is manifest with the simple bash task. The defined task will great a user using a parameterized username value:

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
name: hello
spec:
params:
- name: username
    type: string
steps:
    - name: hello
    image: ubuntu
    script: |
        #!/bin/bash
        echo "Hello $(params.username)!"
```

Listing 4.1: Simple Tekton bash task [6]

The pipeline can be executed by using the Tekton command-line interface, web-based UI, or by triggering one of the Tekton webhooks. Execution will create a *PipelineRun* Kubernetes object, which can be inspected for initial parameters and current run status.

Tekton provides a variety of integrations with existing CI/CD services such as Jenkins or GitHub, which helps to execute your pipelines within already existing workflows. Tekton follows modern techniques by using containerized workloads and supporting micro-services architecture, making it easy to deploy and test applications in a cloud-based environment.

Tekton is a community-driven project. While being a powerful workflow management system, Tekton has a few downsides. Tekton provides ready-to-use building blocks for

pipeline creation, but their functionality may not be enough for assembling complex workflow with bunches of dependencies to resolve and control. Tekton web-based interface cannot be compared with interfaces of some of the existing tools, making it harder to understand on the initial setup 4.3.

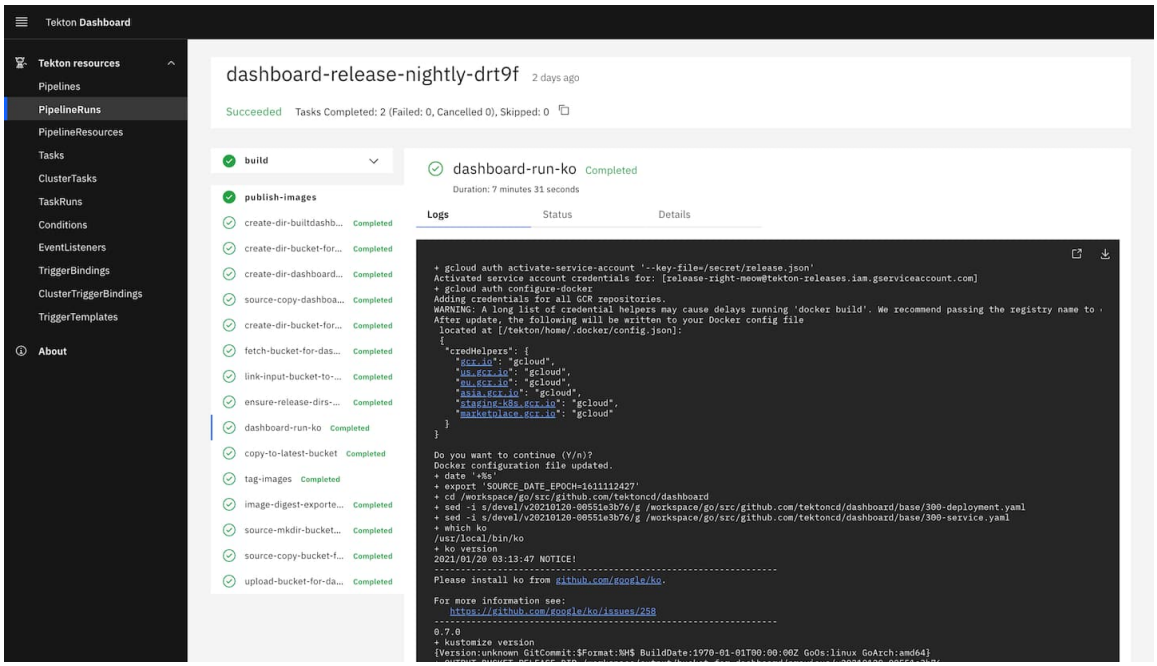

Figure 4.3: Tekton user interface [6].

4.4 Luigi

Luigi is an open-source framework for Python, developed and released by Spotify. Luigi was created to help build and process complex data pipelines. This workflow management system enables dependency resolution between job batches, scheduling, and task retries on failure.

From the main benefits of this tool can be highlighted:

- **Scalability**. Luigi was designed to handle large amounts of transferred data.
- **Complexity**. An object-oriented interface makes it easy to create and manage complex workflows. Luigi allows writing custom tasks, making it a highly extensible pipeline tool.
- **Dependencies scheduler**. Luigi's scheduler makes sure that all task dependencies are resolved in the correct order.
- **User interface**. Luigi's web-based UI allows monitoring of the progress of each task.

Luigi's tasks are defined in Python code. A bunch of dependent tasks composes a pipeline. Tasks may have dependencies on each other, by which letter the scheduler determines in what order they should be executed. Output data, generated from the executed task, can be passed as input to the next, downstream tasks. If the task fails, Luigi will

automatically retry failed task up to a declared number of times, if the task still will be failing – the pipeline raises an error. Luigi pipelines can be managed by the dedicated web-based user interface (UI) or command line interface (CLI).

As Spotify Luigi is a relatively small project compared to the others in the same field, it has some considerable disadvantages:

- **Limited support**. Despite the open-source nature of the Luigi, it's not a first choice for many users. Which can sometimes make it difficult to find a solution to your problem.
- **Real-time processing**. Luigi is not suitable for real-time task processing as it is mainly dedicated to working with large amounts of data.

Chapter 5

Apache Airflow

As for the definition from the official Apache Airflow documentation [2] – *"Apache Airflow is an open-source platform for developing, scheduling, and monitoring batch-oriented workflows. Airflow's extensible Python framework enables you to build workflows connecting with virtually any technology. A web interface helps manage the state of your workflows. Airflow is deployable in many ways, varying from a single process on your laptop to a distributed setup to support even the biggest workflows."*

Apache Airflow is an open-source platform that is used for scheduling, building, and monitoring data pipelines. Directed Acyclic Graph (DAG) is the core concept of Airflow. DAG is a collection of dependent on each other tasks. DAGs are defined programmatically, using Python programming language.

5.1 Directed Acyclic Graph (DAG) and tasks

DAG may be described as a collection of nodes connected by directed edges that only go in one direction and do not produce any cycles. An example of the simple Airflow DAG can be seen in figure 5.1.

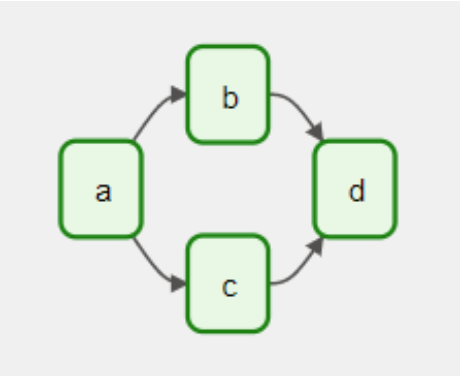

Figure 5.1: Simple DAG example [2]. DAG defines 4 tasks – tasks b and c are dependent on task a. Task d is dependent on tasks b and c execution.

In code, DAGs can be defined using a Python context manager, a DAG decorator, or a simple constructor. Within a defined DAG are tasks and their relationships.

Airflow DAG is composed of smaller jobs – tasks. These tasks represent the code that needs to be executed. Tasks are executed as they are scheduled.

Each task has its own requirements in order for it to start executing. Triggers can be the successful execution of all upstream tasks, at least one successful previous task, or, for example, it can be executed independently on the result of the rest when its turn comes.

Task dependencies can be defined using the Python bitshift operators » and «. These operators represent upstream and downstream dependencies, respectively. Also, these dependencies can be set using the `upstream` and `downstream` task methods, which take the dependent task as the only argument.

Besides the common, custom task, Airflow provides some built-in tasks such as Python, Bash, or Email. These tasks are called operators. Operators simplify the creation of tasks, reducing their definition to passing corresponding arguments to the operator. For example, for a Python operator, it is enough to provide it with a function that will have to be executed when the time comes for this task. The bash operator requires to pass the needed shell command. A simple example of the bash task definition in code can be found in the listing 5.1.

```
from airflow.operators.bash import BashOperator

bash_task = BashOperator(
    task_id="update-submodules",
    bash_command="git submodule update --init --recursive --remote",
    cwd="/home/user/framework/"
)
```

Listing 5.1: Bash task for the update of git submodules.

5.2 XComs

XCom is a way to pass data between tasks in Apache Airflow without creating your own complex mechanisms and explicit dependencies for passing dynamic parameters.

Data types, which can be transferred through the xcom, range from simple ones such as integers or strings to more complicated objects and data structures. This allows developers to create complex and flexible workflows by allowing a task to share its output with multiple downstream tasks during pipeline runtime.

Xcoms are stored and identified in the database with a key, task, and dag, where they have been pushed. Implicitly, each operator stores its result under the identifier `return_value`. Additional data can be explicitly pushed and pulled from storage using the `xcom_push` and `xcom_pull` task instance methods. An example of pushing the namespace name in the task storage using the xcom:

```
task_instance.xcom_push(key="cluster_namespace", value="testing-project")
```

5.3 Scheduling

Scheduling is one of the biggest Apache Airflow features. The *Scheduler* helps to determine when and what task should be executed.

There are a few options for setting scheduling mechanisms on your DAGs and tasks. The most popular of them is fixed or interval-based scheduling, which allows you to set up

a workflow execution plan in fixed intervals. These intervals could be, for example, every week, day, or every hour. Another popular configuration is the dependency of a task on one of the external workflows, such as the success of the upstream job or the presence of a specific data entry in the storage.

In order to configure scheduling for your workflow, it is enough to use one of the following definitions:

- Task should be executed only once, at the beginning of 10th July:

```
from datetime import datetime
from airflow.operators.python_operator import PythonOperator

task1 = PythonOperator(
    task_id="first-task",
    dag=dag,
    python_callable=task_run,
    schedule_interval=datetime(2023, 7, 10),
)
```

- Task should be executed every 5 minutes:

```
from datetime import timedelta
from airflow.operators.python_operator import PythonOperator

task2 = PythonOperator(
    task_id="second-task",
    dag=dag,
    python_callable=task_run,
    schedule_interval=timedelta(minutes=5),
)
```

- Task should run daily:

```
from airflow.operators.python_operator import PythonOperator

task3 = PythonOperator(
    task_id="third-task",
    dag=dag,
    python_callable=task_run,
    schedule_interval="@daily",
)
```

In addition, scheduling also can be configured using the *cron* format, a commonly used Unix format for time intervals definition. For example, `"0 18 * * 0-5"` configuration would mean that the task should be executed every weekday at 18pm. Support of *cron* definitions allowing users to set flexible time intervals according to their needs.

5.4 Airflow management

Communication with Airflow functions is done through the Airflow command line utility (CLI) or the graphical user interface (GUI). The web client, for many users, is often the

main interface for working with DAGs. The web interface helps you efficiently manage your DAGs by providing detailed information about runtime, task results, and available resources.

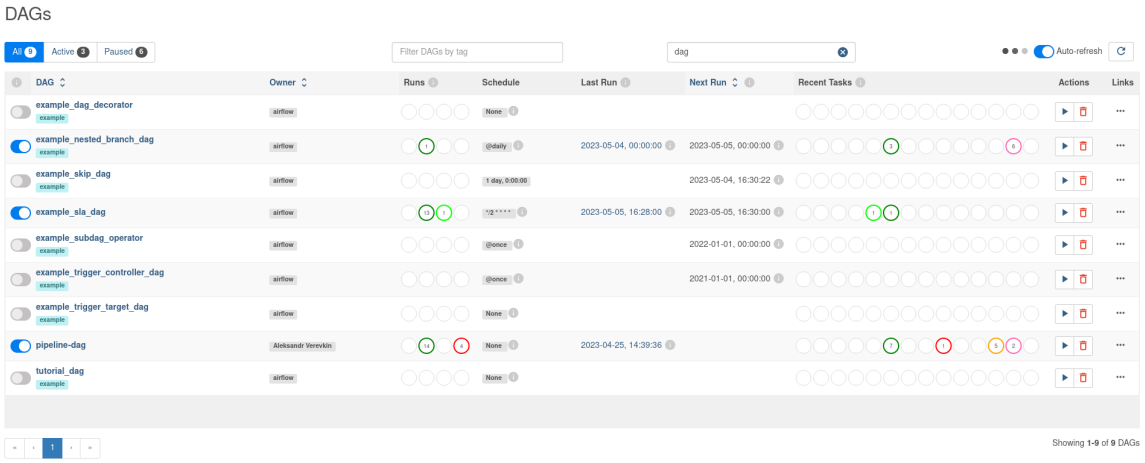

Figure 5.2: DAGs view.

Despite all the advantages of working with Airflow UI, the web client is a pure *Quality of life* addition and is not required for comfortable workflow management.

With the Airflow command line utility, you can manage and monitor Airflow from your favorite terminal. The commands like `dags list`, `dags show`, `dags pause`, and `dags trigger` allow you to interact with DAGs. The `db check`, `db init` and `db shell` commands give you access to the attached databases. The `connections add`, `connections list`, and `connections get` commands exist for the management of external services connections.

5.5 Integration with other services

Apache Airflow provides the possibility to integrate your workflow with other, external services. These services can be databases, cloud storage, or other workflow tools. Mostly, connections to these services are available through the Airflow operators.

Airflow provides native operator implementations for a bunch of external services. However, it is possible to create a custom operator for a service that does not yet have native support.

In order to set up an operator to work with an external service, you must provide him with information about the connection. These connections are usually stored in the Airflow database or can be dynamically set up using environment variables. Each connection has a unique ID and can be used by referencing it in the operator configuration. Here are some examples of external services that Airflow can communicate with:

- **MySQL**: to send a query to execute to a MySQL database, it is enough to use the built-in MySQL operator with a configured connection and provide it with the desired command.
- **Slack**: in order to use the Slack operator, you need to provide him with a Slack connection ID, a channel, and a message to send.

- **Jenkins**: to start the Jenkins job from the Airflow workflow, it is enough to provide the Jenkins operator with the Jenkins connection ID with the credentials of the authorized user in it.

5.6 Airflow modules

The core functionality of Airflow can be significantly expanded, providing integration with many known tools simply by installing the corresponding package – provider. These packages typically add new operators, connections, and other new functionalities.

One of the fundamental modules are packages with the `apache-airflow-providers-` prefix. These modules expand Airflow with a large number of integrations with external services. Among them, it is possible to find integrations with services like Discord, Facebook, GitHub, or Telegram.

Airflow extras are another popular package type. Extras add new operators, authentication methods, and other useful features. Extras include asynchronous operators, Kerberos authentication, Kubernetes executor, and many more.

Furthermore, it is possible for users to create their own custom packages with integrations for services that are not yet supported. These modules are usually hosted in the user's git repositories and can be installed just like other Python packages.

5.7 Choice explanation

After researching the available tools for automating workflow, it was decided to choose Apache Airflow. The Directed Acyclic Graph (DAG) concept, that Airflow is based on, satisfies the requirements of the framework to be able to skip parts of the pipeline and still come to the same end.

Flexible configuration of task dependencies and scheduling of their execution allows the creation of a dynamically changing workflow without rewriting the source code of the framework.

The Airflow web server provides a graphical view of the ongoing process. This allows the monitoring of the current pipeline state without accessing the command line terminal.

Chapter 6

Purposed solution

In appendix A, you can see the diagram of the proposed framework. In the beginning, the program must receive all the scripts and utilities that must be executed. These modules should have a clear order of execution. The framework should be as versatile as possible. Having 2 mandatory modules, it is needed to make configuration as flexible as possible in order to be able to extend the framework for other products and workflows.

Next, we need to configure the framework execution. Ideally, these settings should be in one or two places and there would be no need to jump between dozens of configuration files. The configuration will include such things as restrictions on the cloud resources usage, the names of modules with resources for the cluster and product management, connection information for the Jenkins server, the name of the Jenkins job with tests to be executed, the email to which will be sent a report with the results, and the names of module tasks in the Airflow. The majority of configuration variables must be optional and have default values.

At the beginning of the execution, an inspection of existing clusters will be performed to determine if one of the clusters satisfies the needs and can be used for testing. If the cluster under inspection satisfies the requested configuration, the products on that cluster are also inspected. The obtained information is sent to the element with the product installation, which then, according to this information, will decide if it is necessary to install the product or if the desired product configuration is already present on the cluster.

To determine the required cloud resources for the cluster, the Python API for each cloud service must be used. The connection information for each of them can be acquired from the cluster-management repository and used for the required calculations.

The cluster-management repository will also be used to manage resources on the cluster, as it stores credentials for the installed ones.

Modules that are not repositories with Osia or free-deployer packages, should be flexible and available for placement in any part of the pipeline, whether at the very beginning, end, or somewhere in the middle. In our case, these are modules with cluster configuration and installation of external services. These modules will be placed right after the cluster installation and before the product deployment.

At the end of the specified workflow, depending on the configuration, the tests will be executed. The Jenkins job will need to be provided with the API URL of the installed or already existing cluster. Along with this, Jenkins needs to be provided with a namespace with an installed or, again, already existing product. All of this information will be determined during the framework runtime and parsed dynamically to the task with tests.

Chapter 7

Implementation details

7.1 Run configurations

For the individual run settings, it was decided to use *Dynaconf* – a dynamic configuration management library for Python. Dynaconf allows flexible configuration definition which can be either from one of the supported format files, such as YAML, INI, or TOML, or by setting up environment variables for each run [4]. Later, these configurations can be easily accessed by using simple syntax like `settings.MY_SETTING` or `settings["MY_SETTING"]` instead of writing hundreds of code lines for configuration parsing and reading.

Validators help define which settings should present, allowed settings values and types. Preventing bugs and making code more robust. There is a possibility to create a custom Validator, which can check for the more complex configuration conditions and mandatory values.

Dynaconf was also chosen for its support to set configurations for different environments and easily switch between them, overriding entire settings without changing them manually. The default environment called `default`. Further, this environment will be used for demonstration purposes.

Effective configurations are parsed and stored in the `pipeline.config` project module, listing 7.1, and can can be accessed by importing a `settings` variable from it.

```
from dynaconf import Validator
from dynaconf.base import LazySettings
from dynaconf.constants import DEFAULT_SETTINGS_FILES

settings = LazySettings(
    environments=True,
    lowercase_read=True,
    load_dotenv=True,
    default_settings_paths=DEFAULT_SETTINGS_FILES,
    validators=[
        Validator("modules", must_exist=True)
    ],
    validate_only=["modules"],
)
```

Listing 7.1: `pipeline.config` module with settings configuration

Dynaconf configurations on the listing 7.1 allow different settings environments, a list of predefined settings file extensions, from where configurations can be fetched, and validation for core pipeline setting – pipeline modules. Also, was enabled overlaying of these settings if the `.env` file is present in the configurations directory. It is a good practice to store there settings, which may be changed in the feature, such as environment variables prefix or custom Dynaconf loaders. Project `.env` file can be inspected in the listing 7.2.

```
ENVVAR_PREFIX_FOR_DYNACONF="_3SCALE_PIPELINE"
MERGE_ENABLED_FOR_DYNACONF="false"
LOADERS_FOR_DYNACONF=false
```

Listing 7.2: Project `.env` configurations.

7.1.1 Settings files

When somebody new will be accessing the implemented framework, they will see two configuration files in the `config` subdirectory. These two files are `settings.yaml` and `settings.yaml.tpl`.

File with `.tpl` extension contains a fully configured example of the pipeline setup, with the documented settings. This file serves as a documentation template, making it easy to understand the concept of settings and, if needed, find the required setting entry.

Other settings file, `settings.yaml`, contains a minimalistic framework configuration that will be parsed on the execution. You can change effective configuration right in this file, however, it is a convention to copy his content into the new `settings.yaml.local` file.

The `settings.yaml.local` configuration file, if present, will completely override the `settings.yaml`, making this functionality most suitable for defining your local configuration. An example of the configuration file can be found in the appendix B.

7.1.2 Settings structure

The three main entries in the configuration file are **constraints**, **tests**, and **modules**. *Constraints* providing interface for the framework-specific configurations such as limit for products on the cluster, limit for existing clusters with a specific cloud provider, or whether cluster should always be installed on the execution, skipping by that one of the core framework functionalities – cluster resources management, section 7.3.

Tests settings entry, according to its name, refers to the test configurations that are contained inside. Test configurations include dynamic settings such as the Jenkins job that should be run, the email where the test results will be sent, or of course, if the tests should be run at the end of the pipeline at all.

Lastly, the most important pipeline configuration resides under the *modules* entry. This key contains a list of the framework modules that should be executed. Modules are scripts or utilities that have an entry point and can be executed either by using Bash or Python. Modules should be added to the *modules* subdirectory and name of the module in the configuration should match with one of the modules in this directory. It is a general recommendation to add each utility repository like a Git submodule, as they will be synchronized with the code repository at the beginning of the workflow. Module order matters. For the full framework functionality utilization, it is needed to have cluster-management and free-deployer-workplace modules defined, with the cluster-management module residing before the product workplace module. However, it is not required. Combinations

like only cluster installation, product deployment, or tests run are also supported and can be executed without any additional code twitching, making it possible to define personal workflow in the same environment. The default name for the cluster management module is "cluster-management", for the product module – "free-deployer-workplace". Default names can be changed by overriding them in `constraints.cluster_module` and `constraints.product_workplace_module` constraints accordingly.

7.1.3 Settings parsing

Parsing of *constraints* and *tests* settings takes place in the `utils` module. Values are fetched by using the Python utility package – *Weakget*. Weakget provides a wrapper around `.get` and `getattr`, returning a default value stated after the `%` operator if any of the looked-up keys don't exist [15].

There is a possibility in the feature to move this parsing into separate functions, which will read the given configuration and substitute missing values with the default or automatically fetch them from the cluster parameters. But right now, everything that should be done for getting the required parameter is to import the `pipeline.utils` module and use a predefined variable, making it accessible to get these values across all of the code. Listing 7.3 shows how some of the settings are parsed programmatically.

```
from weakget import weakget

from pipeline.config import settings

# constraints limits
LIMITS = weakget(settings)["constraints"]["limits"] % {}

ALWAYS_INSTALL_CLUSTER = weakget(LIMITS)["always_install_cluster"] % False
MAX_AWS_CLUSTERS = weakget(LIMITS)["max_aws_clusters"] % 3
MAX_OS_LOAD_PERCENTAGE = weakget(LIMITS)["max_os_load_percentage"] % 90
MAX_PRODUCTS_ON_CLUSTER = weakget(LIMITS)["max_products_on_cluster"] % 10
```

Listing 7.3: Settings parsing example.

Some of the variables in the `pipeline.utils` module cannot be changed dynamically. Task names and id's are one of those values. Sometimes task id need to be accessible from the later tasks, good example can be cluster context, which contains cluster's API URL and project where the product should be installed. This context is pulled by the product management task to determine where it should reside. Listing 7.4 shows the definition of names for some of this tasks.

```
# names
MAIN_PIPELINE_NAME = "main-pipeline-group"
TEST_GROUP_NAME = "tests-group"

CLUSTER_MANAGEMENT_START_NAME = "manage-cluster-start"
CLUSTER_MANAGEMENT_END_NAME = "manage-cluster-end"

PRODUCT_MANAGEMENT_START_NAME = "manage-product-start"
PRODUCT_MANAGEMENT_END_NAME = "manage-product-end"
```

Listing 7.4: Tasks names definition example.

7.2 Tasks processing

The full pipeline is built from one main DAG. At the beginning of the pipeline execution, there is always a submodule initialization task. This task makes sure that git submodules are synchronized with the current state of their branches. Executed command:

```
$ git submodule update --init --recursive --remote --merge
```

After synchronization, modules are read in the sequence of their occurrence in configuration. Finally, after all modules are read and saved into the workflow, tests task is appended.

```
@task_group(group_id=utils.MAIN_PIPELINE_NAME)
def main_pipeline():
    """Main pipeline task group"""
    from pipeline.tasks.tasks.tasks import task_from_module
    from pipeline.tasks.tasks.tests import build_tests_task
    from pipeline.utils.utils import connect_tasks

    pipeline_tasks = []
    # create tasks from existing modules
    for module in settings.modules:
        module_tg = task_from_module(module)
        pipeline_tasks.append(module_tg())
    # set tests tasks
    tests_group = build_tests_task()
    pipeline_tasks.append(tests_group())
    # create single tasks pipeline
    connect_tasks(pipeline_tasks)
```

Listing 7.5: Main task group.

The utility `connect_tasks` was made to provide a linear dependencies connection to the given tasks. Tasks in the list are connected using task instance method `set_downstream`, which is making the used task a downstream dependency of the other task.

Inside the module reading loop, there is a task management entry-point function – `task_from_module`. The function takes a single argument – module, which should be converted to a pipeline task. Inside the function each task is separated from others by its own group, making it easy to monitor and manage them in the web interface. Full workflow can be seen in the figure 7.1.

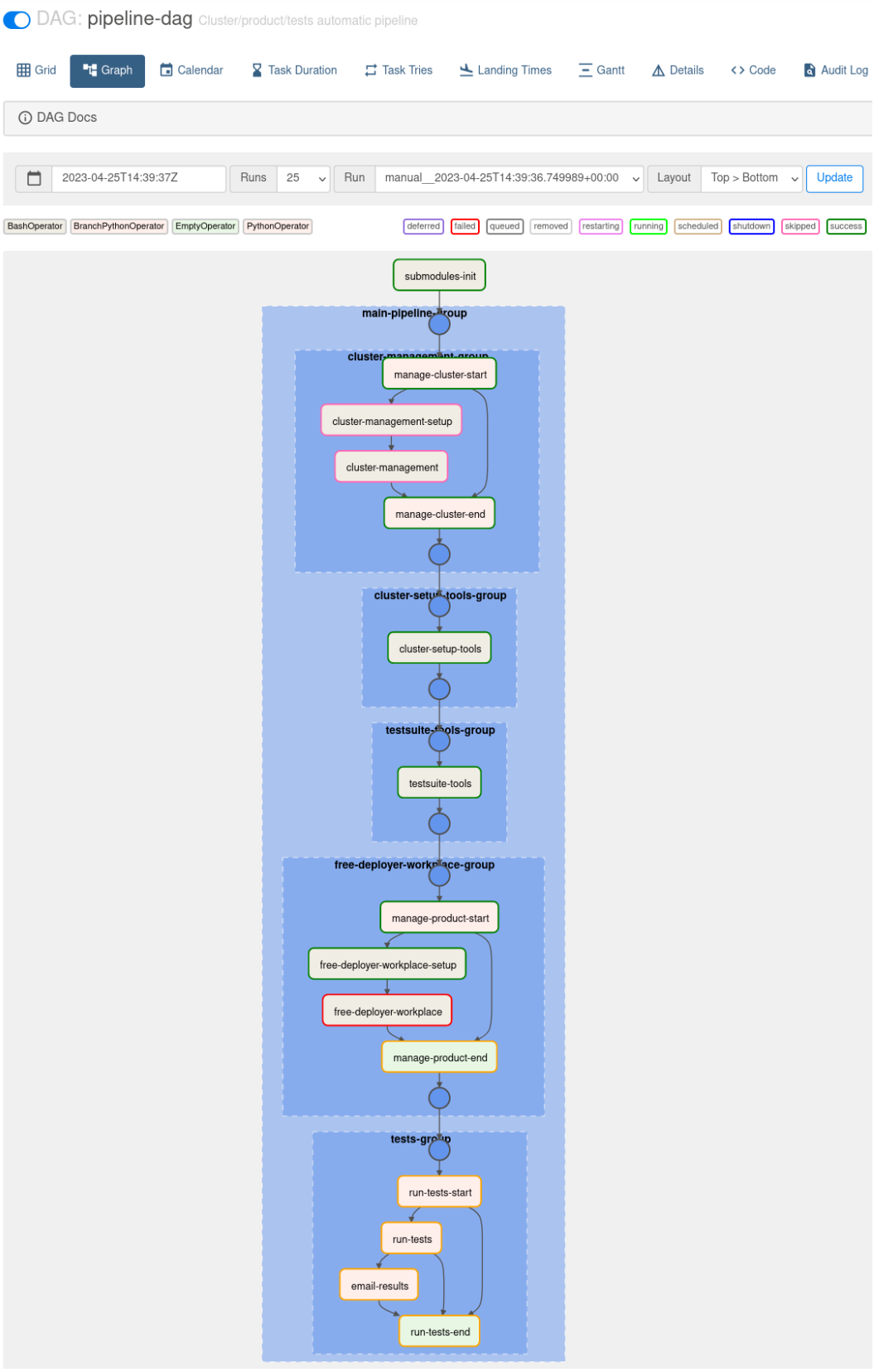

Figure 7.1: Final tasks pipeline in the Apache Airflow UI.

Each configuration module, except the cluster and product modules, which are resolved separately, is resolved according to its operation – Bash or Python.

7.3 Cluster resources management

The cluster management task group consists of 4 tasks. Where 2 of them are generic install environment and install cluster tasks. At the begging of the cluster installation, a task, with the default name *manage-product-start*, is executed. This task will search through the clusters saved in the cluster-management repository and will decide if it is needed to install the new cluster.

Cluster inspections are based on two, important for the testing, cluster settings. One of these settings is the cluster version, or, in other words, the cluster's current channel from where it is getting its updates. Cluster's channel is fetched directly from the cluster, using *clusterversion* Openshift resource and cluster's kubeconfig, which is saved in the cluster's installation directory under the *$INSTALLATION_DIR/auth/kubeconfig* path after the installation.

Another important setting for the testing on the cluster, which used during the cluster's installation, is the cloud provider. The provider can be easily acquired from the `metadata.json` file. This file is placed in the installation directory by the cluster installation script and is required for the cluster's removal. Currently, the only supported cloud providers are *OpenStack* and *S3* by *AWS*.

Configurations of installed cluster versions and clouds are compared, searching for the suitable cluster. Fitting cluster context is sent further for the product-based investigations, which should be configured and provided by the pipeline user. The output of this step is the cluster's API URL and the namespace where the testing product resides. Both of these entries can be empty according to the management outcome. These values are used in the cluster management task group end step, where the context for the next tasks is set. If none of the clusters are suitable for testing purposes, the desired cloud provider must be checked for the sufficiency of the resources for the new cluster installation.

AWS S3 pricing is *pay-as-you-go* type, which is based on the used resources each month. This type of pricing creates a need to be frugal with existing instances. Requiring AWS resources not to drain the money from the budget, using S3 instances only during the testing phase and immediately deleting them after. Clusters are checked only on the number of present clusters with the AWS cloud provider. The limit on the instances is 3 simultaneously deployed clusters and can be changed by accessing the `max_aws_clusters` value in the pipeline settings.

On the other hand, OpenStack resources require more precise inspections. OpenStack resources quota is separated into 3 parts – storage, networking and compute resources. Storage quota is based on the amount of computer space, allocated for the instances in the cloud. Networking quota following network capabilities, such as virtual servers, load balancers, and amount of the used network bandwidth. And finally, most important for our purposes and able to be completely filled, is the computing quota. The computing quota is referring to the virtual machines (VMs) created in the cloud environment. Resource cost is typically based on the CPU and RAM allocated to each instance.

Required RAM for the OpenStack instance is counted based on the master and worker virtual machine's flavor and amount of their replicas. The flavor is the predefined virtual hardware configuration that specifies the number of resources, such as disk space, network bandwidth, and RAM, that will be allocated for the instance. OpenStack has some prede-

fined flavors, but usually, it is preferable for customers to create custom flavors with their own configurations. A virtual machine replica is the "copy" of an instance, which is created for the purposes of application high availability and allowing quick restore from the failure. Setting more running replicas is crucial for the avoidance of data loss and increase of the virtual instance uptime to the available maximum. Each replica requires a separate amount of resources according to its flavor.

The required amount of RAM for the allocation is counted based on the cloud configuration, which is specified in the cluster management module. RAM is calculated by the following formulas:

$$REQUIRED\_RAM = (RAM_M * REP_M) + (RAM_W * REP_W)$$
$$NEW\_RAM\_USAGE = \frac{USED\_RAM + REQUIRED\_RAM}{RAM\_LIMIT}$$

Where *RAM* is the amount of RAM required by chosen flavor and *REP* is the count of replicas for each type of machine (**M**aster or **W**orker). The amount of RAM usage after the cluster installation is divided by the overall RAM usage limit. The result is compared with the set limit in the pipeline configuration, which, by recommended default, is 90 percent. This limit is available for override by `max_os_load_percentage` option tuning in the limits settings.

OpenStack VMs flavors RAM and current compute quotas are acquired by the utilization of OpenStack API, and OpenStack Compute API – *Novaclient*, Python packages.

Whether the cluster was installed or installation was skipped due to the presence of a suitable cluster with the product, the finishing cluster management task is executed. This task sets the Openshift context for future steps. Deciding, according to the output of the resources management task, what cluster credentials to use on login and if it should create a new namespace for the future product installation.

If the existing cluster context was returned from the management task, this context will be used for the login to the existing cluster. If, however, no context was returned, it could only mean that the newly installed cluster should be used for the testing. In this case, the new cluster configuration is read, and his API URL and kubeadmin password is used for the cluster login.

All installed clusters information is stored in the cluster management directory, as it was stated in the chapter 2. For the pipeline framework needs, the framework will fetch clusters information and store it in the Python data structure, from where all this information can be comfortably iterated. This information consists of the cluster installation directory, Openshift context, kubeadmin password, cluster's API URL, version, and cloud provider. The stored information is used in many framework functions and updated if necessary.

7.4 Product management plugins

Management of product resources is carried out right after the suitable cluster was found. Every Openshift application has its own configuration architecture, making it impossible to create one, unified resources management workflow for every product. For the framework to be able to manage different product configurations, the concept of product plugins was conceived.

Product plugins are user-defined workflows for the management of application resources on the cluster. The product plugin should, by design, check for the existing products on

cluster, whose context will be provided as an argument to the plugin entry point. Plugins can utilize pipeline configuration settings like the maximum amount of products that can be present on the cluster simultaneously (`max_products_on_cluster` limit constraint) or rather product management step should be skipped entirely, making the product to be always installed.

One of the plugins was created for demonstration purposes. Module with the plugin is called `threescale.py` and resides in the `pipeline.tasks.product_plugins` package. The name of the module with plugin entry point should be configured in the pipeline settings `product_plugin` constraint, which has been set to the `threescale` value. This setup is required for the framework to find an appropriate module with an entry point function and allow users to easily switch between plugins for the testing of different products.

The entry point function for the plugin is called `manage_product_on_cluster` and is taking one positional argument – cluster context. Context allows one to access the cluster information within the plugin. The cluster can be accessed whether by using his installation directory in the cluster-management module or by accessing the Openshift context of the inspected cluster.

The core *3scale* application resource is the *APIManager*. *APIManager* Openshift resource instance stores all of the general deployment configurations. *3scale* deployments and their testing are mainly based on internal and external databases which are deployed together with the product.

Required and deployed *APIManager*'s are examined for the databases, used for the product setup. Both should be extracted from the configurations and compared with each other. Required databases can easily be extracted from the free-deployer-workplace *APIManager* step configurations. There, they are simply mapped under the according names and can be extracted without much effort.

Already deployed configurations require a different approach. External databases configuration can be found under the *externalComponents* section of the *APIManager* resource. Databases connections are mapped according to their names.[1] Listing 7.6 shows an example of configuration of external databases in the *APIManager* custom resource.

```
spec:
  externalComponents:
    backend:
      redis: true
    system:
      database: true
      redis: true
    zync:
      database: false
```

Listing 7.6: Configuration of external databases in the *APIManager*.

The external system database is one of the edge cases in our comparison. External system database can be either *MySQL* or *PostgreSQL*. Secret named *system-database* should be created on the cluster prior to deployment and should contain a URL with an external

[1] https://github.com/3scale/3scale-operator/blob/master/doc/apimanager-reference.md#externalcomponentsspec.

database connection. According to the documentation of the architecture, [2] the connection URL should have the following format:

```
<DBScheme>://<AdminUser>:<AdminPassword>@<DatabaseHost>/<DatabaseName>
```

From the beginning of the given URL, it is possible to extract the schema of the database. The schema can be either *mysql2* or *postgresql*, depending on the database used.

After the extraction of both configurations, databases are compared, deciding, if the deployment can be considered a suitable instance. Plugin defines an output, which should be followed for the proper framework execution. Three plugin outcome cases are defined:

- Cluster is suitable and has an appropriate namespace for tests to run – cluster's API URL, kubeadmin password, and namespace containing product should be returned.
- Cluster is suitable but doesn't have an appropriate namespace for tests (new project will be created) – should be returned cluster's API URL and kubeadmin password.
- Cluster is not suitable for the testing on products (cluster will be installed and the new project will be created) – the plugin should return nothing.

Plugins output helps cluster resources management task to decide which cluster and namespace to use for future interactions.

An example starting point for the custom plugin creation is defined in the `test_product` module. Presence of description of the creation of the plugin inside makes it a great starting point for the user.

7.5 Product testing

The last and optional pipeline job is the testing of the product. The testing job is made up of two tasks: execution of tests and the reporting of the test result. The latter depends on the success of the former. Whether tests should be run at the end of the workflow is determined by setting the `run_tests` variable in the configuration and the corresponding branching task in the pipeline.

The cluster and namespace contexts are pulled from the cluster-management job and are used for the configuration of the Jenkins build. The Jenkins Job is triggered using the Airflow *JenkinsJobTriggerOperator*, which is just the wrapper for the *python-jenkins* library. The Jenkins build additionally can receive user parameters by setting mapping of them under the `params` key in the configurations. Names of the parameters with cluster context and namespace are also can be configured.

The reporting of the test result is made through email. In order for the report to work, customers need to set an email, where they want to receive a report, and an email server, from where they will be getting the testing results, by providing the email address and the password for third-party apps. The report contains a reference link to the build and the most important information about it. Information includes build result, duration, artifacts, and the user who started this job. The results reporting is also an optional task and can be turned off by setting to `False` the `send_report` configuration variable.

[2] https://github.com/3scale/3scale-operator/blob/master/doc/apimanager-reference.md#system-database.

7.5.1 Connections

For the testing job to work properly it is necessary to set the Jenkins and SMTP servers configurations. In the Apache Airflow these connections are managed by the URIs of the servers. The URIs can contain a server host, port, credentials, and other required information about the server connection.

The Airflow provides two ways how to configure the connection to the external services. One of them is to set a connection statically, by saving it into the Airflow database. This can be acquired through the Airflow graphical or command line interfaces.

The other way how to set the connection is to set the environment variable with the connection URI. While for the Jenkins it is enough to only set the URI variable, the SMTP server requires to set the additional configuration variables such as server port or whether the SSL communication should be enabled. This method was chosen because of the ability to dynamically set the connection values at workflow runtime and not to override them in the database every time there is a new server.

Connections configuration takes place in the `pipeline.connections` module. The base connection interface class is called `BaseConnection` and is implemented using the *Singleton* design pattern. The Singleton pattern ensures that there is exactly one instance of the object and intercepts any requests to create a new one [9].

The Jenkins and SMTP server connections inherit the `BaseConnection` class and provide their own methods for the connection configurations where they setup the according environment variables. So later, it is enough to provide a connection id for each service and establish the connections with the desired servers.

Chapter 8

Testing and possible extensions

The framework was thoroughly tested by the *3scale Quality Engineering* team members. A fair amount of bugs were found and corrected. Some additional functionality has been added thanks to the provided review. Functionality such as the ability to run only a test module by providing cluster context and the name of the project in the configuration, or the ability to skip some parts of resource management if *always create cluster/namespace* flag has been set.

8.1 Testing process

Testing was performed by executing the pipeline under different cluster and product configurations. Were tested dozens of launches with different combinations. For example, an AWS cluster of version 4.10 and 3scale 2.12 with external databases, or a cluster of version 4.12, installed in OpenStack storage, and version 2.10 of 3scale with AWS database, on which the tests were sequentially run.

Was verified correct order of the executed modules. Cases of placing modules before and after installation of the cluster and product.

The work of configuration parameters was checked – their influence on the work of the pipeline. The ability to skip test execution or, alternatively, just execute tests on the existing environment.

The feature of monitoring available cluster resources was tested. Cases with different occupancy of the OpenStack storage and the number of clusters installed with the AWS S3 provider.

We also tested such edge cases where, for example, only was needed to install a cluster or just deploy the product on the existing cluster and run tests on it.

8.2 Monitoring

The workflow of the pipeline can be inspected in the Apache Airflow web server, making it easier to follow the execution and see what decisions have been made. The view of the result DAG can be seen in figure 8.1.

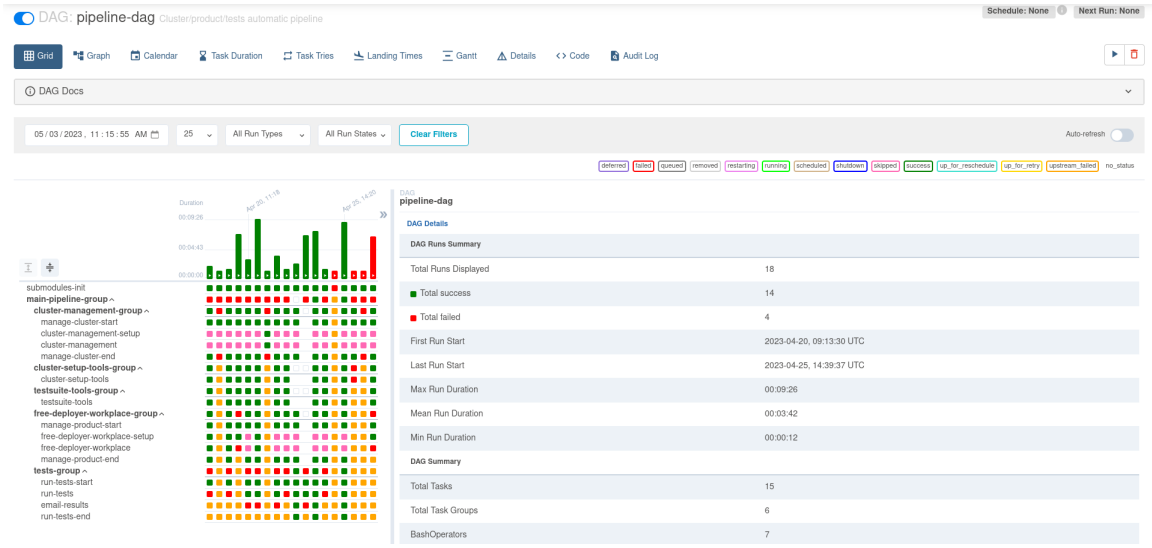

Figure 8.1: View of the pipeline DAG execution in the Airflow web server (Green – successful tasks, red – failed tasks, pink – skipped tasks, orange – skipped due to upstream failure tasks).

Each task's log can be examined for execution information. Information about framework execution is provided by the Airflow library and some of the important information logging were added additionally. In figure 8.2 can be seen a log from the resources management task. During this task, cluster *ocp410* was selected for the examination first. Because a flag was set to create a namespace always, despite the existing projects, product resources management was skipped and a suitable cluster for the product installation was found immediately.

Figure 8.2: Cluster resources management task log.

8.3 Future extensions

Some of the small features include the existence of the configuration entry with a list of the project names to avoid during the product resources management – do not include and check for this project's availability to run the tests on. These can be projects with products that have all needed configurations according to the product plugin, but somehow have the known issue with them. For example, projects with Openshift objects that were deleted by mistake or with pods that weren't reconciled and thus interfere with the correct application work.

Another essential feature will be support for services under the Kerberos authentication. Especially, the Jenkins instance, which is required to have a Kerberos ticket for the authentication. Due to the fact that Airflow has native support for Kerberos,[1] the implementation won't take much effort. The implementation involves setting up ticket generation in the airflow configuration. Then it is just enough to start generating tickets with the ticket renewer using the `airflow kerberos` shell command. Simple Kerberos configuration with the pre-generated service key tab can be seen on the listing 8.1.

```
[core]
security = kerberos

[kerberos]
keytab = /etc/airflow/airflow.keytab
reinit_frequency = 3600
principal = airflow
```

Listing 8.1: Simple Kerberos configuration in the Airflow.

On the other hand, one of the big additions that will be added in the near future is support for another cloud provider – *oVirt*.[2] oVirt is an open-source virtualization management platform, released and developed as a community project by *Red Hat*. oVirt requires a different approach in terms of resource management, by requiring to pay for hardware to run the virtualization environment. On this hardware, it is possible to allocate the required CPU, memory, and shared storage space.

[1] https://airflow.apache.org/docs/apache-airflow/stable/administration-and-deployment/security/kerberos.html.

[2] https://www.ovirt.org.

Chapter 9

Conclusion

The main goal of this thesis was to design and implement a framework for automated infrastructure deployment and test execution for the product in a cloud environment. To do this, it was necessary to study the existing platforms for automatic pipelines and workflows. On the basis of the research, it was necessary to choose a tool that would mostly satisfy existing requirements and use it for the implementation of the framework.

In the theoretical part, the topic of software testing was explored and why this is an important part of development was analyzed. In addition, the topic of testing environments and ways to configure them was broken up.

After getting acquainted with the problem that this work is trying to solve, the most popular tools for setting up the environment were investigated and, as a result, was chosen an open-source tool – Apache Airflow, which allows one to create data pipelines and evaluation workflows.

With the help of the chosen platform, it was possible to create the implementation of the framework for automated testbed deployment and consecutive activation of the tests. The framework uses a modular architecture for the possibility to attach custom scripts and provides resource management functionality on a cluster. The framework supports custom user extensions via product plugins for testing. Thanks to the thorough testing of the framework, most of the implementation errors have been eliminated.

The main problem that I managed to solve was the integration of independent scripts into a single whole and the abstraction of this action for the possibility of reusing it in the future on a similar or different workflow. The implementation required careful thought, thanks to which the framework turned out to be quite flexible for the customizations.

During the creation of this thesis, my knowledge in the field of cloud and virtualization tools was improved. Specifically, their application in the working environment. This thesis helped me improve my communication skills and ability to ask the right questions.

The created framework will continue to develop and evolve for a long time besides this work, offering solutions to new problems in the field of testing and providing the *3scale API Management* team and the other quality engineering teams with an easy way to automate their workflow.

Bibliography

[1] *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models* [ISO/IEC International Standard]. ISO/IEC JTC 1/SC 7 Software and systems engineering, 2011. Available at: https://www.iso.org/standard/35733.html.

[2] *Apache Airflow Documentation 2.4.3* [online]. The Apache Software Foundation, 2022 [cit. 2022-11-18]. Available at: https://airflow.apache.org/docs/apache-airflow/2.4.3/.

[3] *Argo CD - Declarative GitOps CD for Kubernetes* [online]. 2023 [cit. 2023-04-21]. Available at: https://argo-cd.readthedocs.io/en/stable/.

[4] *Dynaconf - Configuration Management for Python* [online]. 2023 [cit. 2023-04-24]. Available at: https://www.dynaconf.com/.

[5] *Jenkins - Build great things at any scale* [online]. 2023 [cit. 2023-04-28]. Available at: https://www.jenkins.io/doc/.

[6] *Tekton - Getting Started with Pipelines* [online]. 2023 [cit. 2023-04-23]. Available at: https://tekton.dev/docs/getting-started/pipelines/.

[7] Axelrod, A. *Complete Guide to Test Automation.* 1st ed. Apress, 2018. ISBN 978-1484238318.

[8] Beizer, B. *Software Testing Techniques.* 2nd ed. Van Nostrand Reinhold, 1990. ISBN 0-442-20672-0.

[9] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* 1st ed. Addison-Wesley Professional, 1994. ISBN 978-0201633610.

[10] Harenslak, B. and Ruiter, J. de. *Data Pipelines with Apache Airflow.* 1st ed. Manning, 2021. ISBN 978-1617296901.

[11] Jaros, M. *Osia - OpenShift infra automation.* [online]. GitHub, 2020. Available at: https://github.com/redhat-cop/osia.

[12] Myers, G. J. *The Art of Software Testing.* 3rd ed. Wiley, 2011. ISBN 978-1118031964.

[13] Patton, R. *Software Testing.* 1st ed. Sams, 2000. ISBN 978-0672319839.

[14] Spillner, A., Linz, T. and Schaefer, H. *Software Testing Foundations: A Study Guide for the Certified Tester Exam.* 3rd ed. Rocky Nook, 2011. ISBN 978-1933952789.

[15] Veen, S. van der. *Weakget - Chaining getattr and .get* [online]. GitHub, 2018. Available at: https://github.com/Syeberman/weakget.

Appendix A

Framework diagram.

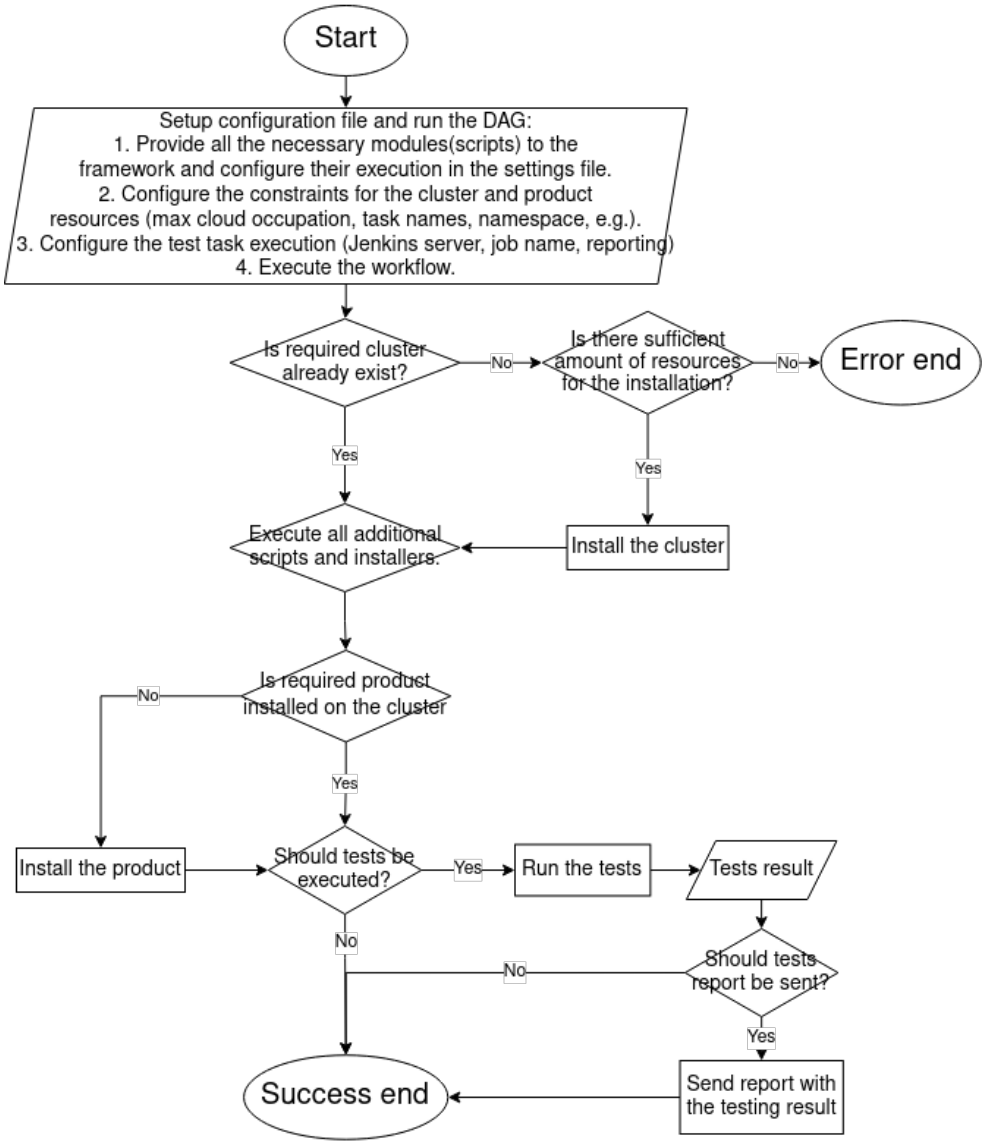

Figure A.1: Diagram of the purposed framework workflow.

Appendix B

Pipeline configuration example.

```
default:
  constraints:
    product_plugin: threescale
    limits:
      always_install_cluster: false
      max_aws_clusters: 3
      max_os_load_percentage: 100
      max_products_on_cluster: 2
    context:
      always_create_namespace: false
      cluster_namespace: "product-by-pipeline"
      cluster_url: ""
  tests:
    run_tests: true
    jenkins_context:
      server:
        uri: "http://login:password@localhost:8080"
      job:
        name: "prod-jobs/operator/testAll"
    jenkins_report:
      send_report: true
      email_address: "yourmail@redhat.com"
      send_from_login: "pipeline@gmail.com"
      send_from_password: "password123"
  modules:
    - cluster-management:
        bash: "osia"
        inline_params:
          - install
        params:
          cluster-name: ocp410
          installer-version: stable-4.10
          cloud: openstack
          dns-provider: nsupdate
```

```
      osp-image-unique: null
      cloud-env: psi
      v: null
      help: null
- cluster-setup-tools:
    bash: "./cluster-setup/setup-cluster.sh"
- testsuite-tools:
    bash: "./deploy-all.sh"
- free-deployer-workplace:
    bash: "free-deployer"
    inline_params:
      - process
    params:
      config-name: basic_aws
      overlay-configs:
        - secrets.yaml
        - nightly.yaml
        - 2_13.yaml
      param:
        - project.enable=false
```

Listing B.1: Pipeline configuration example.