Infrastructure-as-Code Ecosystems

Opdebeeck, Ruben; Zerouali, Ahmed; De Roover, Coen

Published in:
Software Ecosystems: Tooling and Analytics

DOI:
10.1007/978-3-031-36060-2_9

Publication date:
2023

Document Version:
Accepted author manuscript

Chapter 9
Infrastructure-as-Code Ecosystems

Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover

Abstract Infrastructure as Code (IaC) is the practice of automating the provisioning, configuration, and orchestration of systems onto which software is deployed through scripts in domain-specific languages. With the increasing importance of reliable and repeatable deployments, ecosystems are emerging around online repositories of reusable IaC assets. In this chapter, we study two such ecosystems in detail: the one forming around the Docker Hub repository of reusable Docker images, and the one forming around the Ansible Galaxy repository of reusable Ansible roles. We start with an introduction to Docker, the most popular container management tool, and Ansible, the most popular configuration management tool. Although both tools are used to configure machines onto which applications are deployed, they differ fundamentally in the means through which this is achieved. Next, we discuss the Docker Hub and Ansible Galaxy online repositories for reusable Docker images and Ansible roles. Having introduced these emerging ecosystems, we highlight a number of approaches taken by researchers studying them. Subsequently, we survey the state of the art in research on the practices followed by their contributors and users, ranging from the versioning of releases and keeping dependencies up-to-date to detecting bugs. We conclude with the challenges that researchers face when analysing these ecosystems.

Ruben Opdebeeck
Software Languages Lab, Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium, e-mail: ruben.denzel.opdebeeck@vub.be

Ahmed Zerouali
Software Languages Lab, Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium, e-mail: ahmed.zerouali@vub.be

Coen De Roover
Software Languages Lab, Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium, e-mail: coen.de.roover@vub.be

9.1 Introduction

A key activity in IT operations is managing and configuring the digital infrastructure upon which software systems are deployed. In the broadest sense, infrastructure configuration encompasses account management, setting up firewall filters, the installation of software packages, and their configuration. As infrastructures grow to tens or even hundreds of machines, managing these configurations by hand becomes impractical. Moreover, infrastructures that need to be scalable and elastic require the ability for new machines to be spun up or down in a moment's notice. Thus, automation becomes a necessity.

Infrastructure as Code (IaC) is the practice of automatically provisioning, configuring, managing, and orchestrating the machines in a digital infrastructure through source code which can be read both by humans and machines alike. This enables applying best practices already established for application code to infrastructure code, such as change management through version control systems or quality assurance through testing and code review.

Broadly spoken, two paradigms can be discerned among the IaC practice. In the paradigm of immutable infrastructures, existing configurations cannot be changed. It requires technologies such as virtualisation and containerisation to replace the infrastructure with a newly-configured one. In the paradigm of mutable infrastructures, existing configurations can be changed. Doing so reliably and in a repeatable manner, requires technologies such as configuration management languages that automate the changes to pre-existing infrastructures.

Accompanying the rise in IaC technologies are emerging software ecosystems wherein practitioners share open-source artefacts for reuse by others. In this chapter, we discuss two of the most popular of such technologies and their emerging ecosystems, one for each IaC paradigm. We start our discourse with Docker in Section 9.2, the leading containerisation technology powering immutable infrastructures, and its Docker Hub ecosystem of reusable Docker images. Next, we continue with Ansible in Section 9.3, a leading configuration management language powering mutable infrastructures, and its Ansible Galaxy ecosystem of reusable configuration management code. For both technologies, we describe their ecosystem and its participants, metadata, and content. Moreover, we summarise a number of approaches to analysing the ecosystem, in terms of its metadata as well as its artefacts. We conclude with a number of promising avenues of research into these technologies and their ecosystems.

9.2 Docker and its Docker Hub Ecosystem

9.2.1 Introduction to Containerisation

Virtual machines are software-based simulations of the hardware upon which software is deployed. Deploying applications on a virtual machine facilitates porting them across hardware platforms and operating systems, while also isolating them from other applications that share the same physical host. A closely related technology is containerisation, a lightweight version of virtualisation [34]. Containerisation enables developers to package all components required to run an application into a "container". These components include the executable, and all dependencies such as web or database servers and their configuration files. Importantly, the container does not include the operating system or kernel itself. These are shared with the host machine, along with its physical non-virtualised hardware. This allows containers to remain lightweight, yet portable to other hosts with similar hardware and operating system. The application deployed within the container is ensured a consistent infrastructure across different hosts.

Containers are an enabling technology of continuous integration and continuous deployment pipelines (see Section 8.1.3) in which applications are put through a series of tests, each test running the application in an environment of which the last resembles or is the actual production environment. Containerisation is also a popular choice for isolating the micro-services of cloud-native applications [14] from each other, exposing only their functionality through a well-defined interface. Packaging micro-services into containers also facilitates implementing horizontal scaling and elasticity. It suffices to spin up new container instances as the load on the micro-service increases, and to spin redundant containers as the load decreases.

9.2.2 The Docker Containerisation Tool

Docker [49], which is a platform designed to build, share, and run containerised applications, standardized the use of containers with easy-to-use tools for developers, and established a universal packaging approach which subsequently accelerated the adoption of container technologies. In 2013, the Docker Engine was launched, and in 2015, Docker launched the Linux Foundation project "the Open Container Initiative (OCI)" to design open standards for operating-system-level virtualisation, most importantly Linux containers. Today, various containerisation tools support the OCI standards established by Docker (e.g., containerd, runc, CRI-O, etc). According to the 2022 Stack Overflow Developer Survey [47], Docker is the most loved containerisation tool.

Docker containers are created as instances of Docker images. Each image can be built from a blueprint named "Dockerfile"[1], a simple text-based script with

[1] `https://docs.docker.com/engine/reference/builder`

instructions to build a Docker image. Starting from a base image, each instruction creates a new "layer" for the image, which represents the changes to the previous layer caused by executing the instruction. Each layer has an associated digest, i.e., a unique hash signature of the changes, and is immutable. Thus, a Dockerfile produces a Docker image, which is a stack of layers. Once the image is built, it can be used to instantiate multiple containers, each originating from the same Docker image.

A Dockerfile can be based on another Dockerfile, thereby inheriting the latter's image and its layers as the base image. Although it is possible to create Docker images from scratch[2], most images are based on other images. This leads to a hierarchy of images, with a small number of base images forming the foundation.

As images are used to create runnable containers, they contain operating system packages that will complement the kernel provided by the host. Typically, these form the lowest layers of an image, originating from one of the aforementioned base images. Images including Linux-based distributions provide access to that distribution's toolset, such as their package manager (e.g., aptitude on Debian). Subsequent layers may use these package managers to add third-party packages, e.g., language runtimes and language-specific package managers (e.g., JavaScript and npm, or Python and pip). These can then in turn be used to add third-party language-specific libraries. Finally, the top-most layers are typically used for first-party code, binaries, and assets.

For example, Listing 9.1 depicts the Dockerfile that builds the image *redis:7-bullseye*. This Dockerfile includes different Docker commands. `FROM` specifies the parent image which is being built upon. `RUN` executes a shell command in a new layer of the image being built. For example, the shell command `groupadd` will create a new group on the container with the specified name. `ENV` defines environment variables for the container. `WORKDIR` sets the current working directory. `COPY` copies files into the image, etc. This Dockerfile will build an image which inherits the layers of the *debian:bullseye-slim* image, amended with layers containing the changes caused by the other commands. Dockerfiles can contain other commands, such as `VOLUME` to declare a mount point for data volumes and `CMD` to set the default command to run when a container is started.

Listing 9.1: Dockerfile of the image *redis:7-bullseye*

```
1  FROM debian:bullseye-slim
2  RUN groupadd -r -g 999 redis && useradd -r -g redis -u 999 redis
3  ENV GOSU_VERSION 1.14
4  RUN set -eux; \
5  savedAptMark="$(apt-mark showmanual)"; \
6  apt-get update; \
7  apt-get install -y --no-install-recommends ca-certificates dirmngr gnupg wget; \
8  ...
9  WORKDIR /data
10 COPY docker-entrypoint.sh /usr/ local /bin/
11 ENTRYPOINT ["docker-entrypoint.sh"]
```

[2] Using the Docker-reserved minimal image named "scratch", `https://hub.docker.com/_/scratch`.

Running containers are provided read-only access to the image's layers to form the container's file system. Containers can thus access all files created while the image was built. In addition, the container is provided with a new writeable layer on top of the image's layers, called the container layer, which is used to perform all file system modifications caused while the container is running.

9.2.3 The Docker Hub Ecosystem

Next to being instantiated into containers, Docker images can be shipped to online registries to be reused by third parties. Providing a common place to build, update, and share images, software ecosystems form around such online registries. With more than 9.4M images (as of August 2022), Docker Hub is the largest registry for Docker images. It has served billions of image downloads, with images such as Ubuntu, Redis, node, Alpine, and MySQL each having more than a billion downloads.

9.2.3.1 Types of Images Collected on Docker Hub

Images in Docker Hub are distributed as repositories, allowing contributors to group several variants of images (e.g., for different architectures). Repositories may be public or private, where public repositories are categorised as either official or community repositories. The official status is considered a quality label, hinting that the repository contains secure and well-maintained images produced by well-known organisations (e.g., MySQL or Debian). They are therefore often used as the base image for other images. Community repositories, in contrast, can be created by any user or organisation [5].

To facilitate the search for images in Docker Hub, they are labelled with the name of the repository (e.g., *debian*) and a tag (e.g., *buster*). An image can have multiple tags, and thus have multiple labels (e.g., both *debian:bullseye* and *debian:11* refer to the same image). The labels of community images usually start with the name of the organisation or user producing the images, followed by the image name and tag (e.g., *grimoirelab/full:0.2.26*).

Figure 9.1 illustrates the workflow of creating Docker containers from Docker images pulled from Docker Hub [3]. Pulled images can also be used as the base image in a Dockerfile for a new image. For example, Listing 9.2 shows the Dockerfile of the image *debian:bullseye-slim*. This image is built from scratch and it was used as the base image in the Dockerfile in Listing 9.1. Therefore, the layers of this image are all included in the list of layers of the *redis:7-bullseye* image.

[3] For more details about the Docker architecture, we refer the reader to `https://docs.docker.com/get-started/overview/`

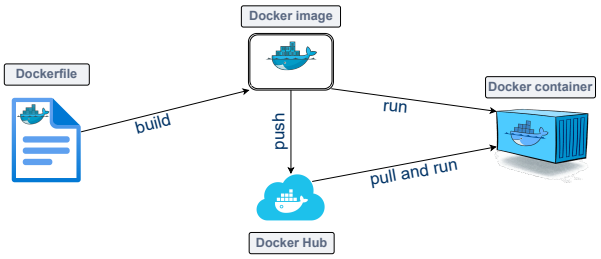

Fig. 9.1: Process of creating a Docker container.

Listing 9.2: Excerpt of the Dockerfile of debian:bullseye-slim

```
1 FROM scratch
2 ADD rootfs.tar.xz /
3 CMD ["bash"]
```

9.2.3.2 Image Metadata Maintained on Docker Hub

The Docker Hub registry maintains several types of metadata about its images. Basic information, like the repository and image name, its type (official or community), tags, number of pulls, number of stars given by other users, and the size of images, can be found on the Docker Hub homepage for each image (e.g., `https://hub.docker.com/_/debian`). This information is also provided by Docker Hub's API, and augmented with the image's creation date, home repository in GitHub (or another hosting service), last pull date, supported architectures (e.g., AMD64, ARM64), a unique SHA digest for the image, etc.

Most importantly, the API provides a manifest[4] for each Docker image, which is a JSON file with metadata about the layers within the image and their size. When trying to run an image with a specified name and tag from Docker Hub, the Docker engine contacts the registry, requesting the manifest for that image. Before downloading the image layers, the engine verifies the manifest's signature, ensuring that the content was produced by a trusted source. Once downloaded, the engine verifies and ensures that the digest of each layer matches the one specified in the manifest. Layer digests are also used to identify layers that have already been downloaded.

[4] `https://docs.docker.com/registry/spec/manifest-v2-2/`

9.2.4 Approaches to Analysing Docker Hub Images

Images within the Docker Hub ecosystem have enjoyed attention in empirical software engineering research. Before surveying the findings of this research, we discuss the different forms of research methods used to analyse Docker Hub images and their ecosystem.

9.2.4.1 Docker Hub Metadata Analysis

In a pure metadata analysis, researchers analyse information that can be gathered about Docker Hub images without running them as containers and without inspecting the source code of their Dockerfiles. Various sources of such metadata exist.

Zerouali *et al.* [57], for instance, mapped the network of popular images derived from the Debian base image without inspecting any Dockerfile. This first required extracting image repository, name, and tags of 27,760 official and 5,842,567 community images through the Docker Hub API. Using the image manifests, they could then identify 9,581 official and 924,139 community images built on top of the Debian base image. The same approach was followed later by the same authors [56] to identify node, Python and Ruby based community images.

Most of the images shared in Docker Hub belong to open source projects, of which the version control repositories are hosted on social coding platforms like GitHub, GitLab, or Bitbucket. By extracting the link to these repositories from Docker Hub, researchers can obtain more metadata about the repositories that version an image's Dockerfile. For instance, information about the age of the repository, its contributors, number of stars, watchers, forks, etc. Commit logs and change statistics from version repositories can also be used in metadata analyses. For example, Lin *et al.* [27] constructed a dataset [28] that contains information about 3,364,529 Docker images and the 378,615 git repositories behind them. The dataset's information from Docker Hub includes the image description, tags, number of pulls, publisher username, etc. The dataset's information from GitHub includes the repository's branches, releases, commit logs, Dockerfile commit history, etc.

9.2.4.2 Static Analysis of Dockerfiles and Docker Images

There are two main ways in which static analysis can be used to study the Docker Hub ecosystem. The first is to statically analyse the Dockerfiles used to build Docker images. The second is to analyse the content of an image's layers without running the image as a container.

For Dockerfiles, the canonical technique is to parse the instructions of the file into an Abstract Syntax Tree (AST). Each instruction is represented as a node in this AST, with the operands of the instruction (e.g., the shell command of a `RUN` instruction or the image name of a `FROM` instruction) appearing as child nodes. Xu *et al.* [52] further classify these nodes into one of four categories: operator (or

Docker command) nodes, resource nodes, shell command nodes, and parameter nodes.

As each `RUN` instruction contains one or several shell commands, it is possible to find some of them including other bash scripts embedded as arguments (e.g., line 5 in Listing 9.1). This adds more levels of nesting to AST representations. Henkel *et al.* [19] presented an AST representation that tackles this challenge of nesting in Dockerfiles. They employed phased parsing, wherein they first perform a simple top-level parse, resulting in an AST as described above. Then, they refine the tree by parsing `RUN` instructions into separate commands (e.g., line 3 of Listing 9.3), and parsing the options of popular command-line tools.

Listing 9.3: Excerpt of the Dockerfile of shogunshogun-dev:latest

```
1 FROM debian:buster-backports
2 MAINTAINER shogun[at]shogun-toolbox.org
3 RUN apt-get update -qq && apt-get upgrade -y && apt-get install -qq
    ↪ --force-yes --no-install-recommends make gcc g++ libc6-dev libbz2-dev
    ↪ ccache libarpack2-dev ...
```

Figure 9.2 shows the AST representation for the Dockerfile in Listing 9.3 using Henkel *et al.*'s approach [19]. The same technique was later used to characterize Dockerfile function code, which is the code that comes after the Docker command (e.g., line 3 in Listing 9.3), through AST paths [60].

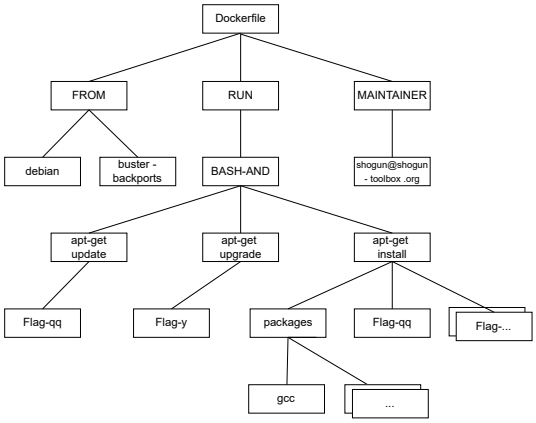

Fig. 9.2: AST of the Dockerfile in Listing 9.3 constructed using the technique of Henkel *et al.* [19].

Several tools have become available to statically analyse Dockerfiles. The DeepSource Docker analyser [5] scans for potential bugs, anti-patterns, security vulnera-

[5] `https://deepsource.io/docs/analyzer/docker/`

bilities, and performance issues. The VSCode Docker extension [6] provides basic linting, whereas Hadolint [7] detects code smells in an AST of the Dockerfile using a rule-based approach, supporting issues that affect commands as well as the shell code in their operands. It is worth mentioning that Hadolint rules can be customised. For example, rule DL4000 was used to check for the usage of the command MAINTAINER which was considered a best practice. When this command became deprecated, DL4000 was updated to indicate that the LABEL command should be used instead.

By downloading the layers of an image and analysing their content, the content of images can also be analysed statically without the need to run them. To download the layers comprising an image, researchers use layer identifiers extracted from the manifest and then download the blobs from the registry using Docker Hub's API. This technique was used by Henriksson *et al.* [21] to extract the list of packages installed in an image and to scan the packages for vulnerabilities.

9.2.4.3 Dynamic Analysis of Dockerfiles and Docker Images

Two types of dynamic analysis can be discerned in the literature: 1) analysis of the build output of Dockerfiles, and 2) analysis of running containers of built images.

Xu *et al.* [52] used dynamic analysis to detect temporary file smells in Dockerfiles; i.e., files that are created and deleted in separate layers while the image is built, which leads to a redundant increase in image size. To this end, they instrumented the host kernel to log file creation and deletion. The resulting traces enable identifying temporary files across multiple layers. Henkel *et al.* [20] combined static and dynamic analysis to warn about Dockerfile breakage. Their approach looks for common error patterns in build logs and associated Dockerfiles. For instance, they find that when Docker reports an error message *"Unable to locate package python-pip"* while building Dockerfiles containing the instructions `FROM ubuntu:latest` and `RUN apt-get -y install python-pip`, this error is due to the use of the undefined, *latest* or *20.04* tags.

Dynamic analysis of running Docker containers can be costly due to the resources required. Nonetheless, it is an effective method for scanning Docker images for security vulnerabilities and bugs. For example, Zerouali *et al.* [55] developed ConPan, a tool that pulls and runs a Docker image from Docker Hub to perform software composition analysis. Once the container is running, the tool executes commands to extract the list of packages available at run-time and compares them to package registries. The resulting list is inspected to analyse the packages' outdatedness (i.e., technical lag [53]) and to identify vulnerable and buggy packages. Similarly, Shu *et al.* [45] proposed the Docker Image Vulnerability Analysis (DIVA) framework to automatically discover, download, and analyze Docker images for security vulnerabilities.

---

[6] `https://github.com/microsoft/vscode-docker`

[7] `https://hadolint.github.io/hadolint/`

9.2.5 Empirical Insights From Analysing Docker Hub Images

Having discussed the analysis methods through which the Docker Hub ecosystem and its assets can be studied, we turn to the latest insights uncovered in this manner.

9.2.5.1 Technical Lag and Security in the Docker Hub Ecosystem

A survey among Docker users revealed that the absence of security vulnerabilities is a top concern in the decision to adopt Docker images [4]. Another survey showed that Docker users are also concerned about the presence of bugs in third-party software loaded within images, and about outdated versions of this software [1]. In contrast, another survey showed that only 19% of developers claim to test their Docker images for vulnerabilities during development [50]. This signals a tendency to produce and consume Docker images without inspecting their software in detail.

Through an in-depth investigation of the attack surface and definition of an adversary model, Combe *et al.* [9] provided a comprehensive overview of the vulnerabilities brought about by the use of Docker. Shu *et al.* [45] analysed Docker images for security vulnerabilities. On a set of 356,218 images, they observed that both official and community images contain an average of 180 vulnerabilities. Many images had not been updated for hundreds of days, calling for a more systematic analysis of the content of Docker containers.

Package changes within Docker images can lead to broken functionality, poor performance, or security issues in the applications that depend on the packages. Gholami *et al.* [16] carried out an analysis in which they studied how packages are changing in official Docker images. After analysing 37k images from official repositories, they found that 50% of the images conducted at least 8 package upgrades. To shed more light on this problem, Sabuhi *et al.* [43] proposed a method to assess the impact of upgrading packages of Docker images on application performance.

Zerouali *et al.* [59] studied the relationship between outdated system packages in Debian-based images, their security vulnerabilities, and their bugs. They computed the difference between the outdated system packages and their latest available releases in terms of versions, vulnerabilities and bugs. They found that no Debian-based image is free of vulnerabilities or bugs, so deployers cannot avoid them even if they deploy the most recent packages in these images. To ensure that they only consider Debian-based images, they relied on Docker's inheritance mechanism previously explained in Section 9.2.3. Later, the same authors extended this study by instantiating the formal technical lag [58] framework along five different dimensions: package lag, time lag, version lag, bug lag and vulnerability lag [57]. The technical lag refers to the difference between deployed software package releases and the ideal (e.g., most fresh, secure or stable) available releases. Then, they carried out an empirical study on 140,498 popular Debian-based images from official and community Docker Hub repositories. For each dimension they found that community images have higher lag than official images. Depending on the lag dimension, images from specific Debian distributions were found to have a higher lag than those coming from others.

For example, version lag was highest for images relying on Debian Testing, while vulnerability lag was highest for OldStable images. They also found that in some cases, the lag increases over time, for example for package and version lag in Debian Testing images.

In a similar study, Zerouali *et al.* [54] focused on npm third-party packages and evaluated their outdatedness and vulnerabilities using 961 official node-based images coming from three Docker Hub repositories, namely node, ghost and mongo-express. They found that the presence of outdated npm packages in official node images increases the risk of security vulnerabilities. Later, the same authors extended this study to include Ruby and Python packages [56]. They found that the last time community images were updated, they had more outdated and vulnerable core packages than non-core ones. After some time, these packages missed more updates leading to more vulnerabilities present in Docker Hub community images. They also reported that the presence of vulnerable packages is considerably more pronounced for node and Ruby images, which tend to be more outdated and more vulnerable than Python images. Moreover, node images tend to have the highest proportion of packages missing major updates, as well as a high number of duplicate package releases. Figure 9.3 shows the process and pipeline followed in the work of Zerouali *et al.* [56] to construct a representative dataset of community Docker Hub images and their installed third-party packages.

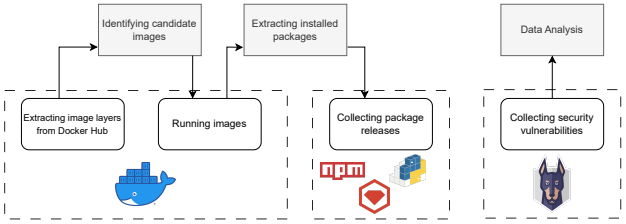

Fig. 9.3: Overview of the data collection pipeline used by Zerouali *et al.* [56].

9.2.5.2 Technical Debt and Code Smells in Dockerfiles

Docker documents several best practices for writing Dockerfiles[8], but developers do not always follow them [51, 42, 19]. This may lead to technical debt and smells in Dockerfiles with a negative impact on the reliability and performance of Docker images. Azuma *et al.* [2] studied self-admitted technical debt (SATD) in Dockerfiles. SATD are comments left by developers as a reminder about code manifesting technical debt. They manually classified all comments found in 3,149 Dockerfiles coming

[8] `https://docs.docker.com/develop/develop-images/dockerfile_best-practices/`

from 462 GitHub repositories. They found that 3% of the comments are SATD, and that the 3 most common SATD classes concern maintainability, testing, and defects.

In a large-scale study of 6,334 Docker projects, Wu *et al.* [51] categorized Dockerfile smells into two categories: DL-smells (i.e., violations of the official Dockerfile best practices) and SC-smells (i.e., violations of shell script best practices). They found that nearly 84% of the analysed projects have smells in their Dockerfiles. Furthermore, they found that DL-smells appear more often than SC-smells.

Ksontni *et al.* [25] manually analysed the Dockerfile and Docker-compose files of 68 projects for technical debt and refactoring opportunities. Docker-compose is a tool that allows the creation and operation of multi-container applications on a single host using YAML files that include configurations such as services, networks, and volumes for each container. They found six Dockerfile technical debt categories related to Image size, Build Time, Duplication, Maintainability, Understandability and Modularity. For Docker-compose files, they found four: Duplication, Maintainability, Understandability and Extensibility. As a remedy, they propose 14 refactorings for Dockerfiles and 12 for Docker-compose files. They conclude that these smells are widespread, and that there is a lack of automatic tools that support developers in fixing them.

9.2.5.3 Challenges in Maintaining and Evolving Dockerfiles

Next to the freshness, security, and quality of Docker images, other socio-technical aspects such as their evolution, reproducibility, and adoption have been studied.

Cito *et al.* [8] characterised the Docker ecosystem by discovering prevalent quality issues and studying the evolution of Docker images. Using a dataset of over 70,000 Dockerfiles, they contrasted the general population with samples containing the top 100 and top 1,000 most popular projects using Docker. They observed that the most popular projects change more often than the rest of the Docker population with an average of 5.81 revisions per year and 5 lines of changed code. Most importantly, based on a representative sample of 560 projects, they observed that one out of three Docker images could not be built from their Dockerfiles.

Oumaziz *et al.* [33] carried out a study on duplicates in Dockerfiles families (e.g., Python Dockerfiles). After inspecting Dockerfiles from 99 official Docker repositories they found that duplicates in Dockerfiles are frequent. They also found that maintainers are aware of the existence of these duplicates. However, they have mixed opinions regarding them. Tsuru *et al.* [48] proposed a method to detect Type-2 code clones in Dockerfiles.

Henkel *et al.* [19] found that Dockerfiles on GitHub have on average nearly five times more best practice violations than those written by Docker experts. They argue for more effective tooling in the realm of Dockerfiles. Eng *et al.* [15] revisited the findings of previous studies about Dockerfiles. After inspecting a large set of 9.4M unique Dockerfiles spanning from 2013-2020, they could confirm previous findings of a downward trend in using open source images and an upward trend of using language images. They also confirmed that the numbers of Dockerfile smells

are slightly decreasing. They concluded that their results support previous studies' recommendations for improving tools for creating Docker images.

9.3 Ansible and its Ansible Galaxy Ecosystem

Having discussed the leading containerisation technology powering immutable infrastructures, we turn our attention to the leading technology for configuration management of mutable infrastructures.

9.3.1 Introduction to Configuration Management

Configuration management tools such as Ansible, Chef, and Puppet provide automation and replicability to infrastructure deployments using domain-specific languages. Practitioners can use these languages to write configuration management scripts wherein they declaratively specify the steps required to configure the machines in a digital infrastructure. The tools then execute these scripts by executing each step on each individual infrastructure machine automatically.

These domain-specific tools often provide built-in abstractions to perform common configuration actions, such as managing users or installing software packages. These abstractions also take care of ensuring idempotence to prevent making changes to a machine's configuration if it is already configured correctly. Such abstractions enable practitioners to evolve their infrastructure mutably, by re-executing changed scripts on machines which were configured with earlier versions of the script.

Nonetheless, tools cannot offer built-in abstractions for each potential scenario and thus provide mechanisms for extension by means of modules and plug-ins. They also offer mechanisms for the reuse of such modules and plug-ins. Some tools even allow reusing whole configuration scripts. Consequently, the more popular of these tools are surrounded by sizeable communities of open-source developers who contribute their own solutions to infrastructure configuration tasks. This has led to a number of new Infrastructure-as-Code software ecosystems, such as the ones surrounding the Ansible Galaxy, Chef Supermarket, and PuppetForge platforms. In this section, we focus on the former, Ansible and the Ansible Galaxy ecosystem, since it is the most popular tool of its kind [47].

9.3.2 The Ansible Configuration Management Tool

Ansible is an automation platform offering solutions for configuration management and provisioning of cloud machines. As such, it has become one of the most popular Infrastructure-as-Code tools today [17]. To configure groups of remote machines,

the Ansible tool pushes configuration changes over the network based on *tasks* written in a YAML-based domain-specific language. This language offers many of the programming constructs found in general-purpose languages, such as variables, expressions, conditionals, simple loops, and exception handling. A complete Ansible program is called a *playbook*, which is further subdivided into one or more *plays*. Ansible code can also be modularised into *roles*, which can be reused in multiple plays or playbooks.

9.3.2.1 Ansible Plays and Playbooks

A playbook contains all code necessary to configure a complete infrastructure. Playbooks consist of several plays, each configuring a group of machines with the same responsibilities. For example, the playbook depicted in Listing 9.4 contains one play to configure database servers (lines 1–16) and another to configure web servers (lines 18–24).

Listing 9.4: Contrived example of an Ansible playbook to configure database and web servers

```
1  - hosts: databases
2    vars:
3      psql_db_name: prod
4      psql_db_user: app
5    tasks:
6      - name: Ensure database exists
7        postgresql_db:
8          name: "{{ psql_db_name }}"
9          state: present
10
11     - name: Ensure user exists and has access to database
12       postgresql_user:
13         db: "{{ psql_db_name }}"
14         name: "{{ psql_db_user }}"
15         priv: ALL
16         state: present
17
18 - hosts: webservers
19   tasks:
20     - name: Ensure apache is installed
21       apt:
22         name: apache2
23         state: present
24     # ...
```

Each play consists of a sequence of tasks, which Ansible executes in sequential order on each machine individually. Each task performs one *action* corresponding to one step in the configuration of a machine. For instance, the first task of the first play in Listing 9.4 (lines 6–9) uses the `postgresql_db` action to create a database. Tasks can also be executed conditionally, or executed in a loop on each item in a list. Ansible offers a number of built-in actions, such as `user` to manage user accounts, or `apt` to install software packages (line 21 in Listing 9.4). Practitioners can also implement their own actions in Python through plug-ins.

Within tasks, practitioners can write expressions in the Jinja2 templating language. Naturally, expressions can refer to variables, which can be defined on many different levels, e.g., play-level variables, variables local to a task, or variables whose value is specific to individual machines. Our example's first play defines two play-level variables (lines 3–4), storing the name of the created database and user. These variables are referenced in three expressions, demarcated by double braces (lines 8, 13, 14). Ansible evaluates the part between double braces and substitutes the result of this evaluation into the string in which the expression is embedded. For instance, the result of evaluating the expression on line 8, which merely refers to a variable, will be "`prod`". Expressions can also manipulate data through *filters*, perform *tests* for conditional logic, or use *lookups* to produce data from external sources. As with actions, users can also define their own filters, tests, and lookups using plug-ins.

9.3.2.2 Ansible Roles

It is common for parts of different plays to perform similar configuration tasks. For instance, both web servers and database servers may require network interfaces to be configured and certain firewall rules to be set up. Similarly, one may want to set up a test environment with a database server that uses the same configuration as the production environment. Rather than duplicating the tasks across the different plays, it is possible to modularise and reuse them with a *role*. An example of a role for the latter situation is depicted in Figure 9.4, which is adapted from the play in Listing 9.4.

Listing 9.5: The `postgres` role's `tasks/main.yml` file

```
1 - name: Ensure database exists
2   postgresql_db:
3     name: "{{ psql_db_name }}"
4     state: present
5
6 - name: Ensure user exists
7   postgresql_user:
8     db: "{{ psql_db_name }}"
9     name: "{{ psql_db_user }}"
10    priv: ALL
11    state: present
```

Listing 9.6: The `postgres` role's `defaults/main.yml` file

```
1 psql_db_name: prod
2 psql_db_user: app
```

Listing 9.7: Playbook using the role

```
1  # Configure production DB
2  - hosts: databases
3    roles:
4      - role: postgres
5
6  # Configure test DB
7  # with separate parameters
8  - hosts: test-database
9    roles:
10     - role: postgres
11       vars:
12         psql_db_name: test
13         psql_db_user: test
```

Fig. 9.4: Contrived example of Ansible role to configure database servers.

Roles follow a strict directory structure. Like a play, a role contains a sequence of tasks, listed in the `tasks/main.yml` file. In our example, this file contains the same tasks as the play it is adapted from (Listing 9.5). Roles commonly have a set of parameters, called *default variables*, listed in `defaults/main.yml`, that can be used to customise the role's behaviour. The example role lists as its default variables the same variables that were present in the original play (Listing 9.6).

When a role is included in a play, these variables can be overridden with values specific for that play. This facilitates reuse within an infrastructure configuration project. Listing 9.7 exemplifies this, where we override the variables specifically for the test environment. Moreover, roles can also be shared across different playbooks, and can thus be used for separate infrastructures. Therefore, it is possible to construct roles and share them with other practitioners for use in their infrastructure projects. This forms the foundation of the Ansible Galaxy ecosystem, which we cover next.

9.3.3 The Ansible Galaxy Ecosystem

Ansible Galaxy is a registry operated by Ansible to facilitate reusing open-source Ansible code. It collects and displays metadata about Ansible roles and plug-ins written by open-source developers. Ansible practitioners can discover the reusable content via Ansible Galaxy's web interface, while a command line utility can be used to install, update, and manage the content in a project.

As of January 2023, Ansible Galaxy indexes over 31,500 roles, the most popular of which have been downloaded millions of times. For instance, the most downloaded role is `geerlingguy.java`[9], with over 15M downloads, closely followed by `geerlingguy.docker`[10] (13.5M downloads). These roles install Java and Docker, respectively, on various Linux systems and offer many customisation options for their users.

Ansible Galaxy is merely an indexer, and does not store the content itself. Instead, the code is stored in GitHub repositories, and installing consists of cloning the repository. To add (or update) roles or plug-ins to Ansible Galaxy, an open-source developer must import their repository. Ansible Galaxy then populates (or updates) various pieces of metadata according to the information found in that repository.

9.3.3.1 Types of Ansible Galaxy Content

Ansible Galaxy aggregates two types of content, namely *roles* and *collections*. As described above, roles contain reusable tasks that are made generic through the use of parameters. When such roles are committed to open-source repositories, they can be imported into Ansible Galaxy to be reused by others. Such roles generally aim

[9] `https://galaxy.ansible.com/geerlingguy/java`

[10] `https://galaxy.ansible.com/geerlingguy/docker`

to configure one facet of an infrastructure. For instance, the `geerlingguy.docker` role mentioned above installs and configures the Docker driver on various Linux platforms. Not only does it install all necessary software packages, it can also set various configuration options, configure the Docker software to start automatically when the system boots, etc.

Ansible Galaxy collections are, as the name implies, collections of related Ansible content. Although collections can include roles, their primary use case is to extend the Ansible language by means of plug-ins for actions, filters, lookups, etc. As such, they intend to facilitate writing configuration tasks, rather than bundling configuration tasks for a specific purpose. A collection's content commonly shares a single theme. For instance, the `community.dns` collection aggregates plug-ins to manage DNS configurations, and the `amazon.aws` collection contains content related to Amazon AWS. Collections are backed by GitHub mono-repositories that contain all of the code for the collection's constituents.

9.3.3.2 Types of Metadata Maintained by Ansible Galaxy

Three types of metadata can be retrieved from Ansible Galaxy, namely *role and collection metadata*, *repository metadata*, and *usage and quality metadata*. Role and collection metadata comprises information that is extracted from a role's `meta/main.yml` file or a collection's manifest file. For roles, this includes its name and author, a description, its license, the platforms and Ansible versions it supports, any dependencies on other roles or collections, tags submitted by the author, etc. For collections, this also includes basic data such as the name and author, but further includes all constituents of the collection and their information.

Repository information comprises information gathered from the GitHub repository. This includes the GitHub URL, repository description, and information about the owner of the repository. Ansible Galaxy also stores the README file of the repository, URLs to Travis CI build information if available, and metrics such as the number of stars, watchers, forks, and open issues at the time the content was imported. It also lists the known versions of the repository, which Ansible Galaxy identifies as git tags that adhere to the semantic versioning format ($x.y.z$). Finally, the metadata includes information about the latest commit at the time of import, and timestamps related to the last import that was performed on the repository.

Usage and quality metadata includes a role or collection's download count and quality information submitted by users and generated by tools. Users of the ecosystem can rate content through community surveys, which inquire about the quality of the role or collection's documentation, its ease of use, readiness for production, etc. These survey responses can be retrieved individually or through an aggregate "community score". Roles have additional scoring metrics, the "quality score", which is an average of a "syntax score" and a "metadata score". Ansible Galaxy applies linters and additional checks during import time, whose warnings are used to calculate these scores and can also be retrieved individually.

9.3.4 Approaches to Analysing Ansible Galaxy

The Ansible Galaxy ecosystem has been the subject of empirical software engineering research in recent years. This section surveys the analyses that form the foundation for the research methods that have been applied to Ansible Galaxy and Ansible client code. We also highlight some key challenges addressed in such research, and point the reader towards datasets.

We distinguish between three types of analyses on Ansible Galaxy. Metadata analysis concerns any analysis of role metadata extracted from Ansible Galaxy and related services, such as GitHub repositories. Static and dynamic analysis concerns analyses that operate on the source code of Ansible projects. The distinguishing factor is whether or not Ansible code is executed. In static analysis, the Ansible code is never executed throughout the analysis, whereas in dynamic analysis, (parts of) the code may be evaluated by specialised interpreters.

9.3.4.1 Ansible Galaxy Metadata Analysis

The Ansible Galaxy indexer can be used as a valuable source of information for ecosystem metadata analysis. Metadata extracted from a role's git repository, such as commits, tags, contributors, issues, and pull requests, can further enrich this metadata. Finally, roles and repositories are often linked to additional sources of metadata, such as CI services like Travis CI, CircleCI, and GitHub Actions, which can provide information about a project's build and test statuses.

Much of this information has been aggregated in Opdebeeck *et al.*'s [29] "Andromeda" dataset. It represents a full snapshot of Ansible Galaxy's content, collected in January 2021. It contains the aforementioned Galaxy metadata of over 25,000 Ansible roles, as well as metadata about their repository's git commits and tags. The authors also published Voyager [31], the tool used to collect this dataset. It implements the pipeline depicted in Figure 9.5. It starts by querying Ansible Galaxy's JSON-based API to discover roles and collect their metadata. Subsequently, it clones the roles' repositories and collects git metadata and versions. It later analyses the code in these repositories to create a structural representation and to categorise code changes between two consecutive role versions.

Importantly, since the dataset mainly contains raw data, it may not be immediately usable for analysis. For instance, one should likely filter out low-quality repositories using standard metrics such as number of downloads and repository activity. Moreover, Ansible Galaxy supports monolithic repositories, or "monorepos", which store multiple roles in a single repository. Such monorepos need to be handled with care when cross-referencing with git data, e.g., not every commit in the repository will apply to every role in the repository. Finally, Ansible Galaxy may serve outdated information, such as repository metrics and role versions, since this information is not continually updated in the index. However, since all Galaxy repositories are linked to GitHub repositories, standard tools can be used to compute up-to-date information.

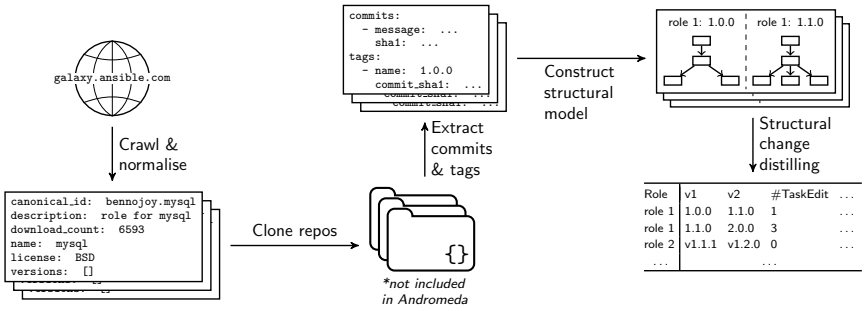

Fig. 9.5: Overview of the data collection pipeline of Voyager [32].

9.3.4.2 Static Analysis of Ansible Infrastructure Code

Prior work has proposed various approaches to static analysis of Ansible code, ranging from code smell detection and defect prediction, over change distilling, to model checking of behavioural properties. Many of these approaches have been summarised for IaC in general by Chiari *et al.* [7]. Static analyses share a need to represent the code they reason about. In this section, we describe a range of representations that have been used in prior work, starting with the simplest and later delving into more advanced representations.

Lexical representations are the simplest code representations. They often take the form of *token streams*, wherein the original source code is split based on positions of certain character classes, e.g., whitespace, or separators such as colons. Such representations do not distinguish between different program elements. This makes it difficult to perform an in-depth analysis of the code. However, token stream representations can be used for tasks such as code smell detection, wherein the detector uses matching rules corresponding to sequences of tokens to highlight potential problems in code. Such approaches have been proposed for other IaC languages, such as Puppet [38]. Nonetheless, research on Ansible has skipped such representations, as it is trivial to obtain syntactical, tree-based representations, as described below.

Syntactical representations are the result of transforming token streams into richer, often tree-based structures by assigning syntactical classes to tokens and reassembling them to represent program elements. For instance, in a syntactical representation, a task appears as a single element with sub-elements for the task's constituents, rather than a subsequent sequence of tokens in a flat token stream. Such representations therefore encode the structure of the program and elide lexical details such as whitespace and separators. This substantially facilitates the analysis of the code. For example, counting the number of tasks that appear in a role requires little more than a simple tree traversal.

Since Ansible code is written in YAML, one can easily obtain a tree-based syntactical representation merely by parsing the YAML file. The resulting representation

is a data structure consisting of lists and key-value dictionaries, closely matching the structure of the source code. However, this representation exhibits a number of shortcomings. First, the elements of the data structure are unlabelled, i.e., without context, one cannot determine whether a key-value mapping is a task, a block of tasks, a collection of variables, or a variable value. This burdens the designer of the analysis, who has to reconstruct this context themselves.

Second, YAML parsing does not take Ansible-specific syntax into account, such as the task argument shorthand which allows one to write an action and its arguments on a single line. For example, the shorthand `apt: name=apache2 state=present` is equivalent to the action of the last task in Figure 9.4. Such shorthands further burden the designer of the analysis, who may need to perform additional transformations on the representation. Nonetheless, many static analysis approaches for Ansible merely parse the YAML code, since it is often sufficient for their use cases, such as code linting [39] or defect prediction [12].

Structural representations build upon syntactical representations to capture more of the structure of an Ansible program. Opdebeeck *et al.*'s [32] structural model for Ansible roles is one such representation which addresses the limitations of the previously-described syntactical representations. It abstracts over the parsed YAML data structures such that each program element is tagged with its type, e.g., tasks, variables, or blocks. Moreover, the representation is normalised such that syntactical variants (e.g., task argument shorthand) map to the same representation.

Figure 9.6 depicts the structural model for the example role in Figure 9.4. One can see that the model represents relevant files in the role, including files containing tasks or default variables. Each unit of the script, including variables and tasks, is modelled as a separate node as a child of the component in which it is contained. For instance, the two variables are part of the default variable file component. The two tasks are part of an implicit top-level block, which in turn is part of the task file.

Opdebeeck *et al.* used this model to distil fine-grained changes to role source code, which were used to compare different role versions. The structural models of over 125,000 role versions and the distilled changes are also contained within the aforementioned Andromeda dataset [29]. Both the structural model builder and change distiller have been made open-source in their Voyager data collection tool [31].

Behavioural representations can store behavioural information derived by prior static analyses alongside the representation of the source code. Static analysis tools can then employ the already-derived information in their own analyses.

Several kinds of behavioural information can be considered. Control and data flow information are among the most common requirements for in-depth static analyses. Control flow information describes the possible paths that a program may take, and includes information about control order, branching points, and loops. Data flow information describes how and where data is defined and used in a program.

The Structured Resource Tree representation by Dai *et al.* [10] is a tree representation whose nodes represent components (files, blocks), units (tasks), variables, and operations (expressions). The tree is structured according to a structural containment relationship, similar to the aforementioned structural model, i.e., a component

node's children are those nodes (other components or units) that are contained within it. Contrary to purely structural representations, Structured Resource Trees contain some behavioural information, such as data flow dependencies between variables and expressions, and a partial execution order relationship. However, since the representation relies on "define before use" heuristics, it cannot account for the intricacies of Ansible's semantics, such as lazy evaluation and complex variable precedence rules. Moreover, the representation does not relate control flow to data flow, and can therefore not capture the influence of values on which control flow branch is taken.

The latter is traditionally captured in Program Dependence Graphs (PDGs), a representation that joins control and data flow information into a single graph. Opdebeeck *et al.*'s [30] PDG representation implements this concept for Ansible roles. It consists of control nodes interlinked through control flow order edges, and data nodes representing unique abstract values present in the program. The latter are linked to control and data nodes alike via data flow (data definition and data usage) edges. For data nodes, the representation further distinguishes between different types of data, namely literals, expressions, named value nodes (variables) and unnamed value nodes (results of expressions). The PDG builder also accounts for the aforementioned intricacies of Ansible's semantics and ensures that no two different concrete values are ever represented by the same abstract data node. To this end, when a variable is used multiple times, the builder ensures that the node for its value is linked to both usages if it can be statically proven that the usages will always receive the exact same value. Otherwise, the builder will use multiple data nodes to represent the different usages of a variable, since its value may have changed in between two usages. An example of a PDG for the role of Figure 9.4 is depicted in Figure 9.7. From the representation, one can immediately see that both tasks use a common value, as well as how these values are defined and used.

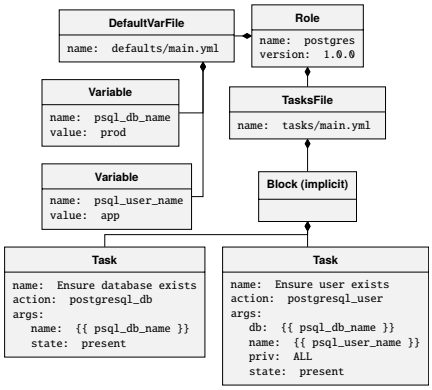

Fig. 9.6: Structural model representation of Figure 9.4.

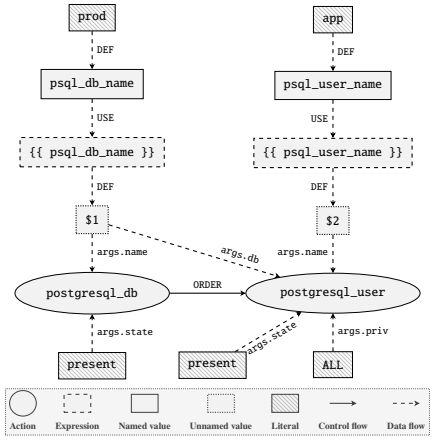

Fig. 9.7: Program dependence graph representation of Figure 9.4.

9.3.4.3 Dynamic Analysis of Ansible Infrastructure Code

Dynamic analysis of configuration management languages in general and Ansible in particular is largely unexplored. Nonetheless, we discern three dynamic analysis approaches that can be applied to Ansible roles.

First, since configuration management tools need to interact with a machine's operating system to configure the machine, a dynamic analysis could collect system call traces while the code is being executed. Such traces provide insights into the behaviour of a configuration script at runtime. The collected traces can then be subjected to further analysis to find faults and defects in the code, as has been applied to find missing control order dependencies in Puppet programs [46]. Moreover, traces for different programs can be matched to one another, e.g., to suggest migrations of imperative shell scripts to Ansible tasks [22].

Second, test cases and their outcomes can serve as a source of information for empirical studies. To test their Ansible roles, many developers use *molecule* [41], a test framework designed for Ansible. Behind the scenes, molecule uses Docker to set up a virtual infrastructure on which it executes an Ansible role. The role developer can subsequently test the final infrastructure state through assertions, also written in the Ansible language. Test outcomes could provide interesting insights for developers. Moreover, these test cases provide an automated means to execute Ansible roles, and could thus be used to generate the aforementioned system call traces.

Finally, rather than relying on test cases written by developers, some approaches generate their own test cases to check behavioural properties of infrastructure code. Hummer *et al.* [23] used model-based testing to find idempotence issues in Chef code. Their approach uses a system model and state transition graph representations (cf. Section 9.3.4.2) to generate a sequence of test cases. Subsequently, it executes these test cases, monitors the actions undertaken by the IaC tool, and analyses them to detect idempotence issues.

9.3.5 Empirical Insights from Analysing Ansible Infrastructure Code

We now discuss the most recent insights from empirical software engineering research into the Ansible ecosystem gained through the analysis methods discussed above. For a more general overview of prior work on Infrastructure as Code, we refer the reader to the systematic mapping study by Rahman *et al.* [37].

9.3.5.1 Code Smells and Quality in the Ansible Galaxy Ecosystem

Code smells are recurring coding patterns indicating flaws in a program's design or implementation. Importantly, code smells themselves are not necessarily defects, their presence merely indicates a potential defect as a suggestion for a developer

to investigate in more depth. Moreover, code smells may highlight potential maintenance issues and areas worthy of refactoring. A related concept is that of code quality metrics, which quantify various quality-related aspects of source code.

Initially, much of the work on quality metrics and code smells in Infrastructure as Code focused on investigating whether metrics for general-purpose languages (e.g., number of statements, inconsistent naming conventions) were applicable to IaC languages. For instance, both Sharma *et al.* [44] and van der Bent *et al.* [3] transposed code smells and quality metrics, respectively, from general-purpose languages to Puppet code. Dalla Palma *et al.* [11] extended these efforts to propose a set of 46 quality metrics for Ansible code, such as the number of tasks, loops, and conditionals. These metrics are computed from simple syntactical source code representations (cf. Section 9.3.4.2), mainly by counting the number of occurrences of keywords in the source code.

Rahman *et al.* [39] defined seven security-related code smells for Ansible, such as the use of hardcoded passwords or a lack of integrity checking on files downloaded by a task. The implement a rule-based detection tool, called SLAC, to identify these smells using a syntactical code representation. They subsequently used this tool to investigate the prevalence of security smells in nearly 15,000 open-source Ansible files, belonging to over 350 repositories. They observed that 25.3% of the studied Ansible repositories found on GitHub contain at least one of the security smells. They furthermore found a total of more than 17,000 security smells throughout these repositories, over 80% of which are uses of hardcoded secrets. The authors further conducted a qualitative analysis by submitting issue reports to GitHub repositories, whose results suggest that most Ansible practitioners agree with the reports.

Opdebeeck *et al.* [30] hypothesise that some of Ansible's semantics, such as lazy evaluation of expressions and a complicated variable precedence system, may lead to defects in Ansible code. The authors therefore propose 6 code smells related to the usage of variables and expressions, such as multiple usages of a variable whose value may have changed between the usages, variables that have been defined through unnecessarily complicated mechanisms, and potentially accidental redefinitions of variables. They developed an approach which operates on their Ansible program dependence graph representation and detects the proposed smells through graph matching. The authors apply this approach to study the prevalence of the proposed smells in Ansible roles indexed by Ansible Galaxy. They observed that these code smells could be detected in the development history of over 4,200 roles, comprising nearly 20% of the studied dataset with a total of more than 31,000 unique smell instances, roughly 22,000 of which are still present in the role's latest version. They also found that although some of the smells are getting fixed by developers, the rate at which the smells are introduced outpaces that of the fixes. However, they observed that it may take multiple years before a smell is fixed.

Kokuryo *et al.* [24] investigated the usage of imperative actions that can run arbitrary shell commands. To this end, they investigated the tasks of 864 Ansible roles discovered through the Ansible Galaxy ecosystem. They found that nearly half of these roles use at least one imperative action, mainly to perform operations for which there is no dedicated Ansible action, but also to perform filesystem operations.

They further found that many of these imperative actions can be replaced by dedicated Ansible actions, which are preferred over arbitrary shell commands.

Hassan and Rahman [18] empirically studied the quality of 4,831 test files in 104 open-source Ansible repositories. They found that 1.8% of these files contain test bugs, which they categorise into 7 types of test defects, such as excessive amounts of logs being generated, or software artefacts that are unavailable when a test script runs. They further identified a number of testing patterns that correlate with the identified bugs, such as assertion roulette and testing playbooks solely in a local environment that may not accurately represent real-world situations.

Many opportunities for code smell and code quality research on Ansible still exist. One particular aspect that is worthy of investigation is a potential connection between Ansible Galaxy's quality survey scores and the detected code smells or proposed code quality metrics. Such an empirical study could be beneficial to the ecosystem and its practitioners, by providing more insights into automated tools and how their results correlate with user-provided qualitative data.

9.3.5.2 Defect Prediction for the Ansible Galaxy Ecosystem

Defect prediction is the use of machine learning models to predict whether code or code changes are defective. Such models are trained to distinguish between defective and defect-free code using a set of code metrics as features. They are then tasked with predicting whether previously unseen code is defective, based on the model's inferences from the training phase. For Ansible, Dalla Palma *et al.* [12] trained and compared 5 different machine learning models using 104 features and a dataset of more than 100 open-source Ansible repositories. They classify their metrics into 3 categories. First, *IaC-oriented metrics* relate to structural properties of the Ansible code, and include the 46 aforementioned metrics proposed by Dalla Palma *et al.* [11] as well as 14 metrics transposed from prior work by Rahman and Williams [40]. Second, *delta metrics* capture the amount of change in these IaC-oriented metrics between successive releases of a script. Finally, *process metrics* capture information regarding the development process rather than the code, e.g., the number of developers that worked on a file. Their empirical analysis uncovered that a Random Forest model provides the best defect prediction performance for their features. They also found that IaC-oriented metrics and specifically the number of tokens and lines of code in a file, tend to maximise the prediction performance. They further observed that, contrary to traditional defect prediction, process metrics do not contribute a significant performance improvement for Ansible code. This could be due to infrastructure code being changed less often than traditional application code.

Borovits *et al.* [6] used a machine learning approach to detect linguistic inconsistencies in Ansible tasks. To this end, they built a synthetic dataset by mutating tasks found in open-source repositories, thereby creating inconsistencies between their description and actual behaviour. They then use this dataset to train and evaluate multiple machine learning classifiers that predict such inconsistencies. Their results

suggest that both classical machine learning techniques and deep learning techniques are effective at finding these inconsistencies.

9.3.5.3 Evolution within the Ansible Galaxy Ecosystem

Many software ecosystems recommend using the semantic versioning (SemVer) standard [35] (*x.y.z*) to denote software release versions [36, 13, 26]. Ansible Galaxy is no different, and recommends its contributors to use SemVer to denote their role release versions. However, it provides no guidelines in regards to interpreting the semantics of semantic versioning, such as selecting a release type (i.e., major, minor, or patch version increments). To alleviate this gap, Opdebeeck *et al.* [32] conducted a large-scale empirical study into the versioning practices employed in the Ansible Galaxy ecosystem. Their findings indicate that although a majority of Ansible Galaxy roles adheres to the semantic versioning syntax, a substantial portion of developers may choose their version increment arbitrarily.

They further conducted a qualitative survey with role developers, querying them about their interpretation of semantic versioning for Ansible roles. The survey reveals that many developers consider a role's default variables to be its main interface. Backwards-incompatible changes to these default variables, e.g., removing or renaming, are considered backwards-incompatible and therefore lead to a major version bump. Adding new variables is often seen as the addition of functionality, and constitutes a minor version increment.

With these insights, the authors extract a set of structural change metrics by performing change distilling on their structural model representation (cf. Section 9.3.4.2). These metrics are used as the features to train a Random Forest machine learning model tasked with predicting the appropriate version bump for a set of changes. The model indicates that the most important features are generally aligned with practitioners' responses, such as changes to default variables or additions of tasks. Nonetheless, their classifier experiences difficulty in correctly predicting the version increment. A subsequent manual analysis uncovered that in many cases, the model was making the correct prediction, yet the role developer chose a wrong version increment in practice.

Finally, the authors synthesised a set of recommendations for Ansible practitioners and the ecosystem as a whole. For instance, they recommended a set of versioning guidelines based on the practitioner responses, and recommended role users to thoroughly test their Ansible code after updating a dependency.

9.4 Conclusion

This chapter discussed the emerging Infrastructure-as-Code ecosystems forming around Ansible and Docker, the most popular embodiments of the mutable and

immutable IaC paradigms respectively, the different approaches to analyse them, and the latest empirical insights.

Both of these ecosystems pivot around registries in which software artefacts are being created and shared. Ansible has Ansible Galaxy while Docker has Docker Hub. Artefacts in these registries are similar to general-purpose libraries, in the sense that they can be reused by others in creating and configuring clusters of (virtual) machines, clouds and containers. Ansible Galaxy hosts Ansible roles and collections, while Docker Hub hosts Docker images.

Previous studies [17, 19, 1] have shown that developers and users of these artefacts face many challenges when they are developing or reusing these artefacts. Example challenges include the understandability and testability of these artefacts. Developers complain that testing is difficult and code review is too basic, i.e., no clear reviewing guidelines are available. In addition, because these ecosystems are relatively young and growing rapidly, their reusable artefacts suffer from inconsistency, i.e., their versions are often backward incompatible. Moreover, many artefacts suffer from security vulnerabilities [57, 39], which may jeopardise dependent artefacts and millions of their users.

Next to the developers, researchers also face different challenges when analysing these ecosystems. First, a lack of tooling requires researchers to create new tools and analyses from scratch that can cope with the specifics of IaC artefacts. Moreover, in many cases, they need to understand the use and development of these artefacts in the wild in order to come up with the appropriate rules to follow when creating data collection and analysis pipelines. This is due to the lack of clear coding conventions and standards such as semantic versioning for Ansible roles and Dockerfile naming. They are also required to choose the right sample of artefacts to analyse. For example, inheritance in Docker can lead to duplicate images being studied multiple times. Similarly, Ansible scripts can be cloned and distributed across different projects. Biased datasets do not accurately represent reality which may lead to misinterpretations of empirical results. Furthermore, the vast scale of available data forces researchers to consider frozen snapshots of an IaC ecosystem, which may impede gaining insights into the ecosystem's evolution. Finally, since most ecosystem research on Ansible and Docker focuses solely on a single ecosystem, the generalisability of their findings is hampered. It is often unclear whether similar observations can be made for other IaC ecosystems, such as the ones formed around Chef and Puppet. We conclude that there are ample opportunities for further research into the domain.

References

1. Anchore.io: Snapshot of the container ecosystem. `https://anchore.com/wp-content/uploads/2017/04/Anchore-Container-Survey-5.pdf` (2017). Accessed 2023-04-15
2. Azuma, H., Matsumoto, S., Kamei, Y., Kusumoto, S.: An empirical study on self-admitted technical debt in dockerfiles. Empirical Software Engineering **27**(2), 1–26 (2022)
3. van der Bent, E., Hage, J., Visser, J., Gousios, G.: How good is your Puppet? an empirically defined and validated quality model for Puppet. In: International Conference on Software

Analysis, Evolution and Reengineering (SANER), pp. 164–174 (2018). DOI 10.1109/SANER.2018.8330206
4. Bettini, A.: Vulnerability exploitation in Docker container environments. FlawCheck, Black Hat Europe (2015)
5. Boettiger, C.: An introduction to Docker for reproducible research. ACM SIGOPS Operating Systems Review **49**(1), 71–79 (2015). DOI 10.1145/2723872.2723882
6. Borovits, N., Kumara, I., Di Nucci, D., Krishnan, P., Dalla Palma, S., Palomba, F., Tamburri, D.A., van den Heuvel, W.J.: FindICI: Using machine learning to detect linguistic inconsistencies between code and natural language descriptions in infrastructure-as-code. Empirical Software Engineering **27**(178) (2022). DOI 10.1007/s10664-022-10215-5
7. Chiari, M., De Pascalis, M., Pradella, M.: Static analysis of infrastructure as code: a survey. In: International Conference on Software Architecture (ICSA), pp. 218–225 (2022). DOI 10.1109/ICSA-C54293.2022.00049
8. Cito, J., Schermann, G., Wittern, J.E., Leitner, P., Zumberi, S., Gall, H.C.: An empirical analysis of the Docker container ecosystem on GitHub. In: International Conference on Mining Software Repositories (MSR), pp. 323–333. IEEE (2017). DOI 10.1109/MSR.2017.67
9. Combe, T., Martin, A., Di Pietro, R.: To Docker or not to Docker: A security perspective. IEEE Cloud Computing **3**(5), 54–62 (2016). DOI 10.1109/MCC.2016.100
10. Dai, T., Karve, A., Koper, G., Zeng, S.: Automatically detecting risky scripts in infrastructure code. In: Symposium on Cloud Computing (SoCC), pp. 358–371. ACM (2020). DOI 10.1145/3419111.3421303
11. Dalla Palma, S., Di Nucci, D., Palomba, F., Tamburri, D.A.: Toward a catalog of software quality metrics for infrastructure code. Journal of Systems and Software **170** (2020). DOI 10.1016/j.jss.2020.110726
12. Dalla Palma, S., Di Nucci, D., Palomba, F., Tamburri, D.A.: Within-project defect prediction of infrastructure-as-code using product and process metrics. Transactions on Software Engineering **48**(6), 2086–2104 (2022). DOI 10.1109/TSE.2021.3051492
13. Decan, A., Mens, T.: What do package dependencies tell us about semantic versioning? Transactions on Software Engineering **47**(6), 1226–1240 (2021). DOI 10.1109/TSE.2019.2918315
14. Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: yesterday, today, and tomorrow. Present and ulterior software engineering pp. 195–216 (2017)
15. Eng, K., Hindle, A.: Revisiting Dockerfiles in open source software over time. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), pp. 449–459. IEEE (2021)
16. Gholami, S., Khazaei, H., Bezemer, C.P.: Should you upgrade official Docker Hub images in production environments? In: International Conference on Software Engineering - New Ideas and Emerging Results (ICSE-NIER), pp. 101–105. IEEE (2021)
17. Guerriero, M., Garriga, M., Tamburri, D.A., Palomba, F.: Adoption, support, and challenges of infrastructure-as-code: Insights from industry. In: International Conference on Software Maintenance and Evolution (ICSME), pp. 580–589. IEEE (2019)
18. Hassan, M.M., Rahman, A.: As code testing: Characterizing test quality in open source Ansible development. In: International Conference on Software Testing, Verification and Validation (ICST), pp. 208–219 (2022). DOI 10.1109/ICST53961.2022.00031
19. Henkel, J., Bird, C., Lahiri, S.K., Reps, T.: Learning from, understanding, and supporting devops artifacts for docker. In: International Conference on Software Engineering (ICSE), pp. 38–49. IEEE (2020)
20. Henkel, J., Silva, D., Teixeira, L., d'Amorim, M., Reps, T.: Shipwright: A human-in-the-loop system for Dockerfile repair. In: International Conference on Software Engineering (ICSE), pp. 1148–1160. IEEE (2021). DOI 10.1109/ICSE43902.2021.00106
21. Henriksson, O., Falk, M.: Static vulnerability analysis of Docker images (2017)
22. Horton, E., Parnin, C.: Dozer: Migrating shell commands to Ansible modules via execution profiling and synthesis. In: International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp. 147–148 (2022). DOI 10.1145/3510457.3513060

23. Hummer, W., Rosenberg, F., Oliveira, F., Eilam, T.: Testing idempotence for infrastructure as code. In: ACM/IFIP/USENIX International Middleware Conference, pp. 368–388 (2013). DOI 10.1007/978-3-642-45065-5%5C_19
24. Kokuryo, S., Kondo, M., Mizuno, O.: An empirical study of utilization of imperative modules in Ansible. In: International Conference on Software Quality, Reliability and Security (QRS), pp. 442–449 (2020). DOI 10.1109/QRS51102.2020.00063
25. Ksontini, E., Kessentini, M., Ferreira, T.d.N., Hassan, F.: Refactorings and technical debt in docker projects: An empirical study. In: International Conference on Automated Software Engineering (ASE), pp. 781–791. IEEE (2021). DOI 10.1109/ASE51524.2021.9678585
26. Lam, P., Dietrich, J., Pearce, D.J.: Putting the semantics into semantic versioning. In: International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!), pp. 157–179. ACM (2020). DOI 10.1145/3426428.3426922
27. Lin, C., Nadi, S., Khazaei, H.: A large-scale data set and an empirical study of Docker images hosted on Docker Hub. In: International Conference on Software Maintenance and Evolution (ICSME), pp. 371–381. IEEE (2020). DOI 10.1109/ICSME46990.2020.00043
28. Lin, C., Nadi, S., Khazaei, H.: A large-scale data set of Docker images hosted on Docker Hub (2020). DOI 10.5281/zenodo.3862987
29. Opdebeeck, R., Zerouali, A., De Roover, C.: Andromeda: A dataset of Ansible Galaxy roles and their evolution. In: International Conference on Mining Software Repositories (MSR), pp. 580–584 (2021). DOI 10.1109/MSR52588.2021.00078
30. Opdebeeck, R., Zerouali, A., De Roover, C.: Smelly variables in Ansible infrastructure code: Detection, prevalence, and lifetime. In: International Conference on Mining Software Repositories (MSR). ACM (2022). DOI 10.1145/3524842.3527964
31. Opdebeeck, R., Zerouali, A., Velázquez-Rodríguez, C., De Roover, C.: Replication package of SCAM 2020 Ansible role semantic versioning empirical study (2020). DOI 10.5281/zenodo.4041169
32. Opdebeeck, R., Zerouali, A., Velázquez-Rodríguez, C., De Roover, C.: On the practice of semantic versioning for Ansible Galaxy roles: An empirical study and a change classification model. Journal of Systems and Software **182** (2021). DOI 10.1016/j.jss.2021.111059
33. Oumaziz, M.A., Falleri, J.R., Blanc, X., Bissyandé, T.F., Klein, J.: Handling duplicates in Dockerfiles families: Learning from experts. In: International Conference on Software Maintenance and Evolution (ICSME), pp. 524–535. IEEE (2019)
34. Pahl, C.: Containerization and the PaaS cloud. IEEE Cloud Computing **2**(3), 24–31 (2015)
35. Preston-Werner, T.: Semantic versioning 2.0.0. `https://semver.org/` (2013). Accessed 2023-04-15
36. Raemaekers, S., van Deursen, A., Visser, J.: Semantic versioning and impact of breaking changes in the maven repository. Journal of Systems and Software **129**, 140–158 (2017). DOI 10.1016/j.jss.2016.04.008
37. Rahman, A., Mahdavi-Hezaveh, R., Williams, L.: A systematic mapping study of infrastructure as code research. Information and Software Technology **108**, 65–77 (2019). DOI 10.1016/j.infsof.2018.12.004
38. Rahman, A., Parnin, C., Williams, L.: The seven sins: Security smells in infrastructure as code scripts. In: International Conference on Software Engineering (ICSE), ICSE '19, pp. 164–175 (2019). DOI 10.1109/ICSE.2019.00033
39. Rahman, A., Rahman, M.R., Parnin, C., Williams, L.: Security smells in Ansible and Chef scripts: A replication study. Transactions on Software Engineering and Methodology **30**(1) (2021). DOI 10.1145/3408897
40. Rahman, A., Williams, L.: Source code properties of defective infrastructure as code scripts. Information and Software Technology **112**, 148–163 (2019). DOI 10.1016/j.infsof.2019.04.013
41. Red Hat, Inc.: Ansible Molecule. `https://molecule.readthedocs.io/en/latest/` (2023). Accessed 2023-04-15
42. Rosa, G., Scalabrino, S., Oliveto, R.: Fixing dockerfile smells: An empirical study. International Conference on Software Maintenance and Evolution (ICSME) (2022)

43. Sabuhi, M., Musilek, P., Bezemer, C.P.: Studying the performance risks of upgrading Docker Hub images: A case study of WordPress. In: International Conference on Performance Engineering, pp. 97–104. ACM (2022)
44. Sharma, T., Fragkoulis, M., Spinellis, D.: Does your configuration code smell? In: Working Conference on Mining Software Repositories (MSR), pp. 189–200 (2016). DOI 10.1145/2901739.2901761
45. Shu, R., Gu, X., Enck, W.: A study of security vulnerabilities on Docker Hub. In: International Conference on Data and Application Security and Privacy, pp. 269–280. ACM (2017). DOI 10.1145/3029806.3029832
46. Sotiropoulos, T., Mitropoulos, D., Spinellis, D.: Practical fault detection in Puppet programs. In: International Conference on Software Engineering (ICSE), pp. 26–37 (2020). DOI 10.1145/3377811.3380384
47. Stack Overflow: 2022 stack overflow developer survey. `https://survey.stackoverflow.co/2022` (2022). Accessed 2023-04-15
48. Tsuru, T., Nakagawa, T., Matsumoto, S., Higo, Y., Kusumoto, S.: Type-2 code clone detection for Dockerfiles. In: International Workshop on Software Clones (IWSC). IEEE (2021)
49. Turnbull, J.: The Docker Book: Containerization is the new virtualization. James Turnbull (2014)
50. Vermeer, B., Henry, W.: Shifting Docker security left. `https://snyk.io/blog/shifting-docker-security-left/` (2019). Accessed 2023-04-15
51. Wu, Y., Zhang, Y., Wang, T., Wang, H.: Characterizing the occurrence of dockerfile smells in open-source software: An empirical study. IEEE Access **8**, 34127–34139 (2020)
52. Xu, J., Wu, Y., Lu, Z., Wang, T.: Dockerfile TF smell detection based on dynamic and static analysis methods. In: Annual Computer Software and Applications Conference (COMPSAC), vol. 1, pp. 185–190. IEEE (2019). DOI 10.1109/COMPSAC.2019.00033
53. Zerouali, A., Constantinou, E., Mens, T., Robles, G., González-Barahona, J.: An empirical analysis of technical lag in npm package dependencies. In: International Conference on Software Reuse (ICSR), *Lecture Notes in Computer Science*, vol. 10826, pp. 95–110. Springer (2018). DOI 10.1007/978-3-319-90421-4_6
54. Zerouali, A., Cosentino, V., Mens, T., Robles, G., Gonzalez-Barahona, J.M.: On the impact of outdated and vulnerable JavaScript packages in Docker images. In: International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 619–623. IEEE (2019)
55. Zerouali, A., Cosentino, V., Robles, G., Gonzalez-Barahona, J.M., Mens, T.: Conpan: a tool to analyze packages in software containers. In: International Conference on Mining Software Repositories (MSR), pp. 592–596. IEEE (2019)
56. Zerouali, A., Mens, T., De Roover, C.: On the usage of JavaScript, Python and Ruby packages in Docker Hub images. Science of Computer Programming **207**, 102653 (2021)
57. Zerouali, A., Mens, T., Decan, A., Gonzalez-Barahona, J., Robles, G.: A multi-dimensional analysis of technical lag in Debian-based Docker images. Empirical Software Engineering **26**(2), 1–45 (2021)
58. Zerouali, A., Mens, T., Gonzalez-Barahona, J., Decan, A., Constantinou, E., Robles, G.: A formal framework for measuring technical lag in component repositories—and its application to npm. Journal of Software: Evolution and Process **31**(8) (2019). DOI 10.1002/smr.2157
59. Zerouali, A., Mens, T., Robles, G., Gonzalez-Barahona, J.M.: On the relation between outdated docker containers, severity vulnerabilities, and bugs. In: International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 491–501. IEEE (2019). DOI 10.1109/SANER.2019.8668013
60. Zhang, Y., Zhang, Y., Mao, X., Wu, Y., Lin, B., Wang, S.: Recommending base image for docker containers based on deep configuration comprehension. In: International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 449–453. IEEE (2022)