FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Towards Safer Continuous Infrastructure-as-Code Deployments

Henrique Lima

UNIVERSIDADE DO PORTO

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Tiago Boldt Sousa

Second Supervisor: Hugo Sereno

July 12, 2021

Towards Safer Continuous Infrastructure-as-Code Deployments

Henrique Lima

Mestrado Integrado em Engenharia Informática e Computação

July 12, 2021

Acknowledgements

The writing of this dissertation could not have been possible without the formidable amount of support received throughout its development.

I would like to first deeply thank my supervisors, professors Tiago Boldt Sousa and Hugo Ferreira, for the consistent availability and guidance with insightful contributions and helpful feedback which ensured high quality standards and created a clearer vision for the direction to take at each step.

I want to thank Amazon Web Services (AWS) for enabling the pursuit of this work through an internship and for always having provided the necessary resources, tools and assistance to ensure its success. I would like to thank the whole AWS CDK team for welcoming and including me with overwhelmingly positive energy, as well as always providing intelligent feedback and proposing and discussing ideas whenever necessary.

I must, however, dedicate a special acknowledgement and thank you to a member of the AWS CDK team in particular, without whom this work would not have been possible, Engineer Rico Huijbers. I am profoundly honored to have called you my mentor and friend from the very conceptualization of the hypothesis, to have had your unconditional technical and moral support throughout the several obstacles, uncertainties and struggles in all phases of this project. Your expertise, insights, teachings and feedback have consistently brought this dissertation to a higher level and will proudly have a great effect on my future professional and personal life.

In addition, I would like to thank my parents and sister for actively motivating me. Your willingness to listen to my problems and constantly ensure my mental and physical well-being, while stimulating me with insightful discussions and promoting my self-development has allowed me to achieve my goals, time after time. You have given me all the discipline and emotional stability required to perform this work consistently over the past year.

To all my faculty colleagues and friends who have kept in touch and helped maintain a good mentality and work-life balance throughout the past months, a huge thank you.

Finally, I want to leave a word to my dear friend Pedro Pinho, who has always stood alongside me through every challenge with encouraging words. Our near-brotherhood friendship has been extremely important to keep me focused and motivated, as well as take my mind off stressful situations whenever they appeared. I am very privileged to be able to call you my friend and count on your understanding, support and wake up calls whenever I need them.

Abstract

As cloud computing resources become more adopted, the infrastructures in which they are used naturally grow in the amount of resources and overall complexity, becoming harder to manage.

Infrastructure-as-Code (IaC) is presented as a solution to this problem, allowing developers to manage and provision these cloud resources programmatically. The infrastructure is then maintained through a code base, allowing general software good practices like version control and peer review, as well as aiding replication of the infrastructure across environments. Some IaC solutions also offer the ability to review the effects of new infrastructure changes before deploying them.

By defining pieces of infrastructure through code or configuration files, it becomes easy to share and reuse them. As such, community-made infrastructure is now widely available - and even more accessible through platforms like CDK Construct Catalog[1], Chef Supermarket[2] or Puppet Forge[3] -, which enables developers to include third-party libraries or templates in their infrastructure. Such a process is hard to review as they can introduce large sets of changes, hampering compliance checks and creating security concerns.

Moreover, infrastructure-as-code deployments are often part of continuous delivery pipelines where the review and approval of changes are manual steps, which contradicts the fundamental principles of continuous delivery.

Throughout this dissertation, techniques for analyzing IaC code were researched and developed to ultimately increase the efficiency and experience of the review and approval processes without sacrificing safety. We were able to create a description and visualization of infrastructure changes for easier user review, as well as enabling automating approval of changes through user-defined rules, either based on recurrence or user expectations. These allow the developer to review changes more easily and eliminate some unnecessary manual actions.

This work was developed and tested for the Amazon Web Services' (AWS) Cloud Development Kit (CDK) and, consequently, uses AWS services.

Keywords: Infrastructure-as-Code, Cloud Computing, Continuous Deployment, Visualization, Automation, Security

[1] Available at https://awscdk.io/
[2] Available at https://supermarket.chef.io/
[3] Available at https://forge.puppet.com/

Resumo

Com o aumento da utilização de recursos de *cloud* computing, as infraestruturas nas quais eles são usados naturalmente se tornam mais complexas e com maior número de recursos, tornando-se mais difíceis de gerir.

Infrastructure-as-Code (IaC) é apresentada como solução para este problema, permitindo que programadores criem e giram este recursos de *cloud* programaticamente. A infraestrutura é, assim, mantida através de código fonte, permitindo a utilização de boas práticas gerais de desenvolvimento de software, como controlo de versões e *peer review*, e facilitando a replicação de infraestrutura em vários ambientes. Algumas solução de IaC dispõem, também, da capacidade de rever os efeitos de alterações à infraestrutura antes de as aplicar.

Ao definir blocos de infraestrutura através de código ou ficheiros de configuração, torna-se fácil partilha-los e reutiliza-los. Assim sendo, infraestrutura feita pela comunidade encontra-se agora largamente disponível - mais acessível ainda através de plataformas como CDK Construct Catalog[4], Chef Supermarket[5] ou Puppet Forge[6] -, permitindo a desenvolvedores a inclusão de bibliotecas *third-party* ou *templates* na sua infraestrutura. Este processo revela dificuldade em rever as alterações pois podem ser introduzidas em grande quantidade, criando preocupações em termos de segurança e dificultando verificações de conformidade.

A aplicação de código de *infrastructure-as-code* é frequentemente feita através de pipelines de *continuous delivery*, onde o processo de revisão e aprovação das alterações são tarefas manuais, algo que contradiz os princípios de *continuous delivery*.

Ao longo desta dissertação, técnicas para analisar código IaC foram investigadas e desenvolvidas para aumentar a eficiência e experiência dos processos de revisão e aprovação sem sacrificar a segurança. Foi criada uma descrição e visualização de alterações a uma infraestrutura para facilitar a sua análise, assim como possibilitar a automação da aprovação de alterações através de regras configuradas pelo utilizador, tanto por recorrência ou por vontade do utilizador. Assim, é mais fácil para o desenvolvedor rever alterações e eliminar ações manuais desnecessárias.

Este trabalho foi desenvolvido e testado sobre a framework Cloud Development Kit (CDK) da Amazon Web Services (AWS) e usa, consequentemente, serviços AWS.

Keywords: Infrastructure-as-Code, Cloud Computing, Continuous Deployment, Visualização de Alterações, Automação, Segurança

[4] Available at https://awscdk.io/
[5] Available at https://supermarket.chef.io/
[6] Available at https://forge.puppet.com/

"Power comes not from knowledge kept,
but from knowledge shared."

Bill Gates

Contents

List of Figures

List of Tables

Abbreviations

AWS	Amazon Web Services
AWS CDK	Amazon Web Services Cloud Development Kit
C2A	the AWS CDK Change Analysis tool, currently a proof-of-concept
CI/CD	Continuous Integration/Continuous Deployment
CLI	Command Line Interface
CVL	Configuration Validation Language
DAG	Directed Acyclic Graph
DLQ	Dead Letter Queue
DSL	Domain Specific Language
GUI	Graphical User Interface
IaaS	Infrastructure as a Service
IaC	Infrastructure-as-Code
IAM	AWS Identity and Access Management
PaaS	Platform as a Service
PR	Pull Request
SaaS	Software as a Service
SSI	Semi-Structured Interview
TOSCA	Topology and Orchestration Specification for Cloud Applications

Chapter 1

Introduction

1.1 Context

Cloud computing is currently used for many applications in a wide range of fields. Several companies currently provide cloud computing services which offer an easier way of meeting requirements like scaleability, resilience and availability by abstracting the management of physical resources. As developers adopt a cloud based approach and their cloud infrastructures increase in complexity, the resources and the way they interact become overwhelming to accurately manage by hand.

Naturally, attempts to automate such management tasks surfaced, such as through scripts or configuration files, introducing the concept of **Infrastructure-as-Code (IaC)**. IaC consists in programmatically provisioning and controlling cloud resources. As such, it allows general software good practices like version control and peer review. It also allows for executing monotonous or frequent cloud management tasks with little to no manual input. A common approach in IaC tools is to allow the user to define the desired state of the infrastructure rather than how it should be deployed - here referred to as declarative IaC.

1.2 Motivation

Consider a developer that may want to include a single-page web application in their cloud infrastructure, for example. In the past, they would have to manually provision and configure the necessary infrastructure. This would be a troublesome and error-prone process, as they would need to setup hosting, network routing, scaling and security services, which justifies the increased adoption of Infrastructure-as-Code solutions to facilitate an infrastructure that nowadays tends to be large, heterogeneous, and elastic [39, 27, 40, 41, 43, 42, 45, 44]. Thanks to these products, developers may define their infrastructure by writing configuration/source code files, which can be auditable both throughout their creation and evolution, and allow analysis of the infrastructure as part of the code review process. This approach to infrastructure management also allows developers to create higher level abstractions for setting up the infrastructure, reuse them and share them as third-party libraries. As these abstractions become more and more available, developers can spend less and less time configuring their infrastructure and may instead use already-developed solutions. With higher level abstractions, however, it becomes harder to determine the effects of their changes on the final infrastructure through regular code review. Considering the mentioned developer setting up a single-page web application, they may use a third-party library which can greatly simplify the provisioning of the necessary resources. Since this library would be able to control the cloud infrastructure of the developer, they would be easily exposed to security threats such as having configurations with low security standards, exposing private data or even using computing power for the creator's benefit. To ensure their safety for a given use case, the developer would have to audit them whenever they are introduced or updated - either through to code review or through infrastructure compliance features offered by some cloud platforms or IaC technologies. However, these are not able to identify worrisome changes that may have happened, just the final state of the infrastructure, which means decreases in security, functionality breaking changes or indirect consequences based on the previous state (e.g. data loss due to database table replacements) easily go unnoticed. IaC code may also be integrated in a Continuous Integration and Continuous Deployment (CI/CD) pipeline, which stresses the importance of a highly effective change review process, such as peer reviews or automatic validation tasks in the pipeline.

1.3 Problem

Like all code, IaC implementations can be easily shared and included in one's code, which can speed up developer operations significantly, but can also present security concerns.

The problem is, then, how to review and analyze the security risk of large IaC code changes efficiently, particularly when introducing third-party code which can be updated at any time. Large code reviews can be very time-consuming and lead developers to rush through them and produce lower quality reviews, hampering detection of vulnerability-prone resources, configurations or permissions.

Multiple IaC solutions offer tools to review the upcoming infrastructure changes of an IaC deployment, but they are designed for small iterations rather than optimizing the readability of large changes. Moreover, this review is an additional step separate from code review. As such, it is often used for manually approving changes during a CI/CD pipeline execution, interrupting it and causing delays in the deployment process.

Typically this results in teams not including community-made IaC code unless it is small enough to be reviewed or they fully trust its source (see Section 4.5.2, p. 23). Consequently, this greatly reduces their adoption and hampers the search for usable existing solutions of particular development problems.

1.4 Goals

In this work, we intend to achieve the following goals:

- Preliminary study - interview IaC professionals to further study their development workflows, experience and views on the subject, to better define the hypothesis and approach of this work.
- Create a reference implementation - design and implement a proof-of-concept tool that produces the artifacts needed to validate the hypothesis.
- Validate the hypothesis - Using the reference implementation, validate the hypothesis and answer the established research questions.

1.5 Document Structure

The present document is comprised of several chapters, described as follows:

- **Introduction** (see Chapter 1, p. 1) - introduces the problem under study and the motivation behind this work, as well as the proposed hypothesis and validation approach.
- **Background** (see Chapter 2, p. 5) - provides the relevant background knowledge needed for understanding the full contents of this document.
- **State of the Art** (see Chapter 3, p. 15) - covers the current state of the art of continuous deployment in IaC, as well as visualization and interpretation of programmatically defined infrastructure.
- **Preliminary Studies** (see Chapter 4, p. 21) - describes the preliminary work carried out to understand the real-world applicability and feasibility of the proposed solution.
- **Problem Statement** (see Chapter 5, p. 29) - formalizes the problem of this work and proposes a preliminary solution based on the literature and studies of chapters 3 and 4. Proposes a validation approach and describes its evaluation.

- **CDK Change Analysis Tool** (see Chapter 6, p. 35) - describes the design of the proof-of-concept tool and the decisions taken on its development
- **Validation** (see Chapter 7, p. 57) - covers the design of the survey, the decisions taken on its development and how it was conducted, and provides an analysis of the obtained results.
- **Conclusions and Work Plan** (see Chapter 8, p. 69) - briefly describes the achieved conclusions, as well as details the work plan for the development of the solution.

Chapter 2

Background

2.1 Cloud Computing

Cloud computing abstracts the creation and configuration of physical and virtual resources for web-based applications. As defined by the National Institute of Standards and Technology (NIST),

> "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." [32]

2.1.1 Services Models

The services available in cloud computing exist under three service models [32, 47]:

- **Infrastructure as a Service (IaaS)** - the network, processing and storage are virtualized. The client can then manage their own infrastructure (e.g. operating systems or application) using those resources.

- **Platform as a Service (PaaS)** - abstracted physical resources similarly to IaaS, but providing an operating system or software platform for the client to configure and deploy applications on.
- **Software as a Service (SaaS)** - access to an application is provided, ensuring the underlying infrastructure and physical resources it is deployed on.

2.1.2 Characteristics

The definition of cloud computing mentioned above (see Section 2.1, p. 5) also encompasses 5 key characteristics. Among them are:

- **On-demand self-service** - the ability to provision resource capabilities (such as storage capacity or server time) automatically from the service provider.
- **Broad network access** - the availability over standard network connections from different client platforms (e.g., smartphones, laptops, tablets).
- **Resource pooling** - the physical resources of the service provider are shared across different customers by dynamically allocating and abstracting them onto virtual ones to which the clients have access, based on demand, without knowledge of their location.
- **Rapid elasticity** - provisioned virtual resource capabilities must be able to adjust to demand, often automatically.
- **Measured service** - resources are automatically controlled and optimized according to collected resource-specific metrics (e.g., storage, bandwidth), which provide transparency and allow the client and the service provider to monitor their usage.

2.1.3 Advantages

The use of cloud computing results in a paradigm shift for developers as they perceive and interact with resources differently, which offers several advantages versus the traditional approach [23]:

- **Cost efficiency** - Companies using cloud services can avoid upfront investment and maintenance costs of on-premise hardware and take advantage of pay-as-you-go payment plans, while benefiting from less expensive computing power, storage, security and networking solutions. Due to the large scale at which resources operate in the cloud, they can be optimized and used more efficiently across multiple clients.
- **Nearly Unlimited Storage** - The large scale of cloud services requires ensuring large storage capacity for the customers' potential needs, which can be allocated on demand, seeming nearly unlimited to any given client.
- **Backup and Recovery** - Backup and restoration of information in cloud computing is usually greatly abstracted for the customer by ensuring data redundancy across physical devices and locations with little to no configuration overhead.

- **Automatic Software Integration** - Cloud solutions include many automatic integrations between services and technologies, reducing configuration efforts.
- **Easy Access to Information** - By having information in the cloud, it can be accessed and managed in any location or time zone, provided there is an internet connection.
- **Quick Deployment** - Cloud services are optimized for quick configuration and iteration of the infrastructure, accelerating deployments.
- **Easier scale of services** - Computing power, storage and networking service plans can often be scaled automatically or easily manually upgradable.
- **Delivery of new services** - Cloud providers can make available new types of interactive services otherwise not accessible to their customers.

2.2 DevOps and CI/CD

DevOps consists of integrating development and operations into the same workflow, so multi-functional teams can continuously produce operational feature deliveries. This results in less miscommunication issues and faster problem resolution [28].

Continuous Integration and Continuous Deployment (CI/CD) play a big role in DevOps. Continuous Integration tries to reduce iteration cycles by decreasing manual time-consuming tasks, such as merging developer code and checking for software quality issues. **Continuous Deployment** focuses on decreasing manual processes of deployment, such as configuration maintenance or resource provisioning, by automating such tasks [28].

CI/CD tasks can be executed as part of a pipeline in which they run sequentially, as long as no issue is detected. In this sequence, deployment tasks tend to be triggered only after the integration tasks have finished successfully.

Development workflows integrate CI/CD pipelines with collaborative version control tools (e.g., GitHub[1], GitLab[2], BitBucket[3]), running automatically on code reviews and merges. Some examples of CI/CD tools include Jenkins[4], CircleCI[5], GitLab CI/CD[6] and GitHub Actions[7]. They allow developers to configure CI/CD pipelines through configuration files, defining, among other settings: how to run each task, their execution order and whether the pipeline should abort upon task failure.

[1] Available at https://github.com/
[2] Available at https://gitlab.com/
[3] Available at https://bitbucket.org/
[4] Available at https://www.jenkins.io/
[5] Available at https://circleci.com/
[6] Available at https://docs.gitlab.com/ee/ci/
[7] Available at https://github.com/features/actions

2.3 Infrastructure-as-Code (IaC)

As cloud-based systems because more complex, it becomes harder to maintain their configurations, provision resources, replicate and ensure consistency of infrastructure across environments and contexts. This creates a need for automating these maintenance tasks, leading to the adoption of the Infrastructure-as-Code concept. As defined by Kief Morris in [35]:

> "Infrastructure as Code is an approach to infrastructure automation based on practices from software development. It emphasizes consistent, repeatable routines for provisioning and changing systems and their configuration. You make changes to code, then use automation to test and apply those changes to your systems."

Initially IaC consisted in writing scripts in general-purpose procedural languages that would interact with the cloud provider's API. However, standard tools are now available, which abstract the provisioning of the infrastructure to either declarative or imperative programming languages. Among the most popular are: Ansible [1], AWS CDK [2], AWS CloudFormation [4], Chef [7], Pulumi [14], Puppet [16] and Terraform [19].

2.3.1 Advantages

The adoption of IaC offers clear advantages over manual infrastructure management. As mentioned by Kief Morris, it allows for [35]:

- Using IT infrastructure as an enabler for rapid delivery of value;
- Reducing the effort and risk of making changes to infrastructure;
- Enabling users of infrastructure to get the resources they need, when they need it;
- Providing common tooling across development, operations, and other stakeholders;
- Creating systems that are reliable, secure, and cost-effective;
- Make governance, security, and compliance controls visible;
- Improving the speed to troubleshoot and resolve failures.

Furthermore, it also helps ensuring consistency and allows reuse across deployments, in addition to enabling peer reviews, history audits and easier rollbacks.

2.3.2 Characteristics of IaC tools

IaC tools have inherent characteristics and concepts regarding their paradigms, approaches and intents, as explained in the following subsections.

2.3.2.1 Programming Paradigms

IaC solutions are divided into two categories regarding their coding paradigm:

- **Declarative**: the developer defines the desired state of the infrastructure, including its resources, accounts and configurations. In most cases Domain Specific Languages (DSLs) are used. A declarative-based IaC tool does not necessarily use declarative programming languages (the case of AWS CDK [2] and Pulumi [14]).

```
defineServerlessFunction("functionName", "path/to/source/code");
```

Listing 2.1: Pseudocode example of a declarative IaC implementation to deploy a serverless function

- **Imperative**: the developer defines the steps to bring the infrastructure to the desired state, allowing extra logical operations that can produce different outcomes depending on the situation. Usually general-purpose programming languages are used, which have benefits such as the wide availability of external libraries.

```
if(!serverlessFunctionExists("functionName")){
    createServerlessFunction("functionName");
}

getServerlessFunction("functionName")
    .resetConfigurations()
    .setPath("path/to/source/code");
```

Listing 2.2: Pseudocode example of an imperative IaC implementation to deploy a serverless function

In other words, while in the declarative approach the code defines *what* the infrastructure should look like, in imperative IaC tools the developer describes *how* the infrastructure should be transformed from the current state into the desired state [35].

2.3.2.2 Immutability and Configuration Drift

The immutability of an IaC approach is determined by how configuration changes are applied on each deployment. In a **mutable** approach, updates existing resources directly whereas an **immutable** one replaces the existing instances by creating and configuring new ones. The latter helps prevent **configuration drift**, which consists in introducing variations between systems that were designed to be identical, commonly caused by manual changes to the resources [35].

2.3.2.3 Stack Management vs Server Configuration tools

An **infrastructure stack** consists of a set of infrastructure resources defined, provisioned and updated as a unit [35]. **Stack management tools** are used for provisioning and managing instances of these developer defined stacks.

Server configuration tools (often called configuration management tools), on the other hand, automate the configuration of existing servers. They often require the installation of agents on each server, which constantly download and apply new changes [35].

Many server configuration tools are able to manage stacks to some degree, despite their focus on server configuration. The same applies to configuring servers using stack management tools.

2.4 Infrastructure-as-Code Tools

The purpose of this section is to analyze some of the most popular IaC solutions - Chef [7], Puppet [16], Ansible [1], [19], AWS CloudFormation [4], AWS CDK [2] and Pulumi [14] - and compare their features and approaches.

2.4.1 Overview

The following characteristics of the considered tools are compared in table 2.1:

- **Release** - The year of the tool's first release.
- **Cloud** - Which major cloud providers are supported.
- **Source** - Whether the source code is publicly available (open) or not (closed).
- **Type** - Whether it is a server configuration tool (see Section 2.3, p. 8)
- **Paradigm** - Whether the infrastructure is defined through declarative or imperative. programming (see Section 2.3, p. 8).
- **Immutable** - Whether the tool adopts an immutable infrastructure approach

Tool	Release	Cloud	Source	Type	Paradigm	Immutable
Chef [7]	2009	All	Open	Server Configuration[8]	Imperative	No
Puppet [16]	2005	All	Open	Server Configuration[2]	Declarative	No
Ansible [1]	2012	All	Open	Server Configuration[3]	Imperative	No
Terraform [19]	2014	All	Open	Stack Management	Declarative	Yes
AWS CloudFormation [4]	2011	AWS	Closed	Stack Management	Declarative	Yes
AWS CDK [2]	2019	AWS	Open	Stack Management	Declarative[4]	Yes
Pulumi [14]	2018	All	Open	Stack Management	Declarative[4]	Yes

Table 2.1: Characteristics comparison between some of the most popular IaC tools

2.4.2 AWS CloudFormation

AWS CloudFormation is a declarative stack management tool released by Amazon in 2011. It allows developers to define their desired infrastructure of AWS resources through **CloudFormation templates**. These templates can then be used to automatically provision and configure the infrastructure in multiple environments.

An instantiated template is called a **stack**, which represents all deployed resources described in the template. A stack allows treating this collection of resources as a single unit. They can be created, updated or deleted by updating the stack and modifying its template. Before applying these changes however, CloudFormation can generate a **change set** (i.e. a summary of the proposed modifications), allowing the developer to review them [5].

2.4.2.1 Intrinsic Functions and Dependencies

The format of CloudFormation templates is centered around the declaration of the AWS resources that will make up its **Stack**. These declarations are listed inside a *Resources* object. Similarly, there are other declarable entities like *parameters* — to use custom values in the template when deploying it — or *outputs* — to export values that can be used outside the deployed stack —, among others.

Since these entities must be able to access information that may not be available prior to the deployment, CloudFormation makes available a set of *intrinsic functions* to use in resource properties, outputs, metadata attributes or update policy attributes. Some of these functions allow treating data or creating conditions, while others are used for referencing other entities or their attributes, where the referenced entities may or may not be declared in the template itself. Intrinsic functions are interpreted as expressions, which may be used in conjunction with other functions [5].

The instantiation of a given resource may depend on the existance of another. Consequently, its creation must follow the creation of the dependee. CloudFormation templates allow the developer to explicitly declare this dependency (e.g. using the *DependsOn* field in resources) or, alternatively, to do so implicitly with the use of intrinsic functions such as *Ref* or *GetAtt* - referencing an entity or an entity's attribute respectively.

2.4.2.2 Nested Stacks

A CloudFormation *stack* can be used to instantiate a different Cloudformation template using a nested stack. This is done by declaring a resource in the original template, that indicates the location of the target template. This feature can be used for separation of responsibilities, reusability of stacks or to circunvent CloudFormation's limit of resources per stack. The nested stack can use

[8] Chef's *Provisioning* stack management tool was discontinued in 2019 [9]

[2] Puppet can provision cloud resources using modules such as puppetlabs-aws [17]

[3] Ansible can provision cloud resources using Ansible's Cloud Modules

[4] AWS CDK and Pulumi allow coding in imperative languages. However, both compile the code into a declarative configuration file which is then processed upon deployment, hence being considered declarative

values from the stack that instantiates it (the parent stack) by using its parameters and may also expose values to be refrenced by the parent stack, using its outputs.

2.4.2.3 Template Formats

CloudFormation templates are written in either JSON or YAML formats, sharing the same structure among both. YAML also offers a short form for applying intrinsic functions (see Section 2.4.2.1, p. 11) through explicit YAML tags [22], resulting in increased readability of the template.

2.4.3 AWS CDK

The AWS Cloud Development Kit (CDK) is an open-source framework for defining cloud infrastructure with programming languages - currently TypeScript, JavaScript, Python, Java, Go and C#/.Net. It produces, through a process called **synthesis**, AWS CloudFormation templates which are then used to provision and maintain the infrastructure resources with AWS CloudFormation.

The main building blocks of this technology are referred to as *Constructs*. A Construct represents a component of the target infrastructure state and abstracts the necessary steps for its instantiation or maintenance - in most cases its insertion onto the CloudFormation template. Simple Constructs may represent a single AWS resource, while more complex ones may represent a set of multiple resources, how they relate and other additional tasks necessary to instantiate it (e.g. the deployment of assets onto a cloud storage service). In addition, *Constructs* can use other *Constructs*, increasing the level of abstraction of the component they represent. A CloudFormation *stack* can be created by using one or more constructs, forming a conceptual hierarchical tree where its root is the CloudFormation stack, the leaves are the created CloudFormation entities and all other nodes are Constructs.

AWS CDK provides a ***diff*** tool [3], used for reviewing the proposed infrastructure changes, similarly to CloudFormation's change sets (see Section 2.4.2, p. 11). The displayed information includes permission changes, changes to the resources and their location within the code hierarchy as well as the underlying CloudFormation change sets when relevant (see Figure 2.1, p. 13).

2.5 Code Review

Code review is a good practice frequently adopted in software engineering, which consists in manual inspection of source code. This is commonly performed by developers other than the author — peer-reviews — and is adopted in many contexts, including open source and enterprise development. This process aims to help identify defects, as well as transfer knowledge, increase team awareness and shared ownership, find alternative solutions to problems and even improve other aspects of the code (e.g. readability, consistency) [24].

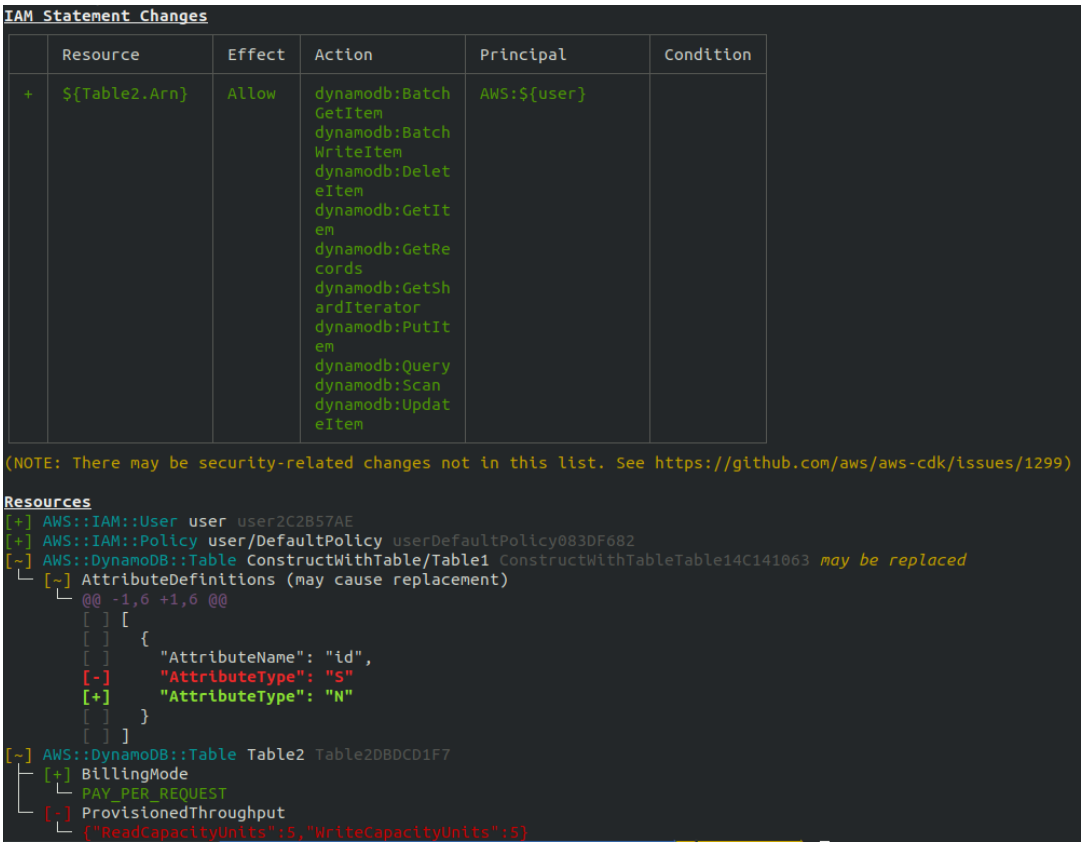

Figure 2.1: CDK's diff tool sample output. Assigning permissions and creating and updating resources. The table above presents changes to permissions (IAM Statements), indicating potential security vulnerabilities, while the *Resources* view on the bottom displays resulting changes to the AWS CloudFormation resource definitions

Multiple tools aid peer reviews, some of which are included in collaborative source code management platforms, such as GitHub, GitLab or Bitbucket. They allow developers to perform a code review before including new code changes into their active codebase. Features include:

- **Code diffing** - Displaying comparison of the code before and after the changes
- **Comments/Discussion section** - Allowing developers to communicate and discuss feedback on code changes. Some solutions allow linking discussions to specific lines of code.
- **Perform automated tasks** - Adding support for executing automated tasks which provide further feedback on the code, commonly using CI/CD pipelines. These may be used, for example, to ensure automated software tests, formatting and styling are compliant.

2.6 Summary

This chapter explains concepts regarding Cloud Computing, DevOps, CI/CD, IaC and Code Review.

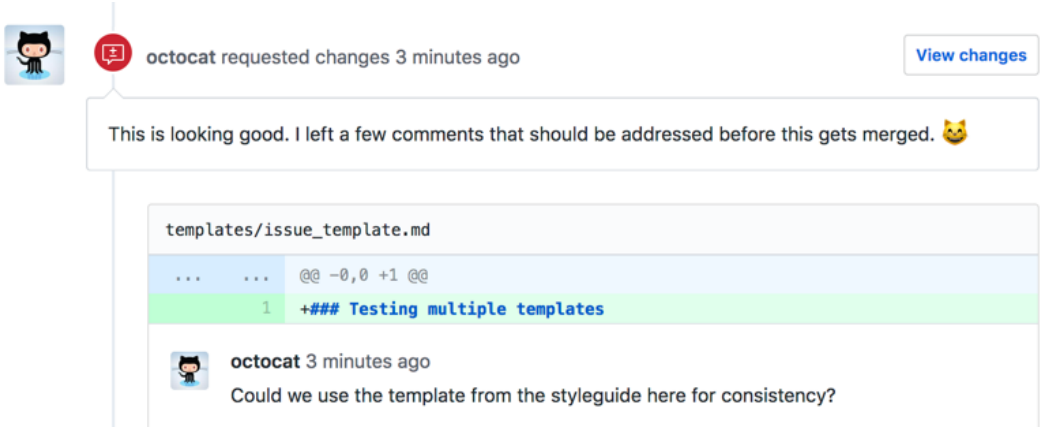

Figure 2.2: Code Review - Example of a comments/discussion section with requested changes in the GitHub platform [12]

Section 2.1 (p. 5) defines Cloud Computing, distinguishes the different service models (IaaS, PaaS and SaaS) (see Section 2.1.1, p. 5), reveals its relevant characteristics (see Section 2.1.2, p. 6) and presents advantages over traditional approaches (see Section 2.1.3, p. 6).

Section 2.2 (p. 7) describes how DevOps allows development teams to include operational tasks in their workflows, focusing on how Continuous Integration and Continuous Deployment (CI/CD) is ensured and how it helps development iteration speed and quality. It explains the idea of CI/CD pipelines and presents tools that allow their use.

Infrastructure-as-Code is described in Section 2.3 (p. 8), along with its advantages (see Section 2.3.1, p. 8). It also distinguishes relevant characteristics of IaC tools (see Section 2.3.2, p. 8) — including paradigms, immutability, configuration drift, stack management and server configuration.

Section 2.4 (p. 10) contains an overview of some of the main IaC tools available, comparing their characteristics. It also offers detailed explanation of relevant concepts concerning AWS CloudFormation (see Section 2.4.2, p. 11) and AWS CDK (see Section 2.4.3, p. 12), as specific knowledge of these technologies will be necessary to perform the validation of the hypothesis (see Chapter 7, p. 57).

Lastly, Section 2.5 (p. 12) presents and explains the purpose of code review and peer-review, as well as outlines the main review-oriented features of available tools.

Chapter 3

State of the Art

This work intends to study solutions to review and analyze the effective changes of large modifications to IaC code, particularly when introducing third-party dependencies which can be updated at any time. The purpose of this chapter is, then, to analyze the current state of the art for the relevant topics of this dissertation and review them against literature research to evaluate the proposal and obtained results. The methodology used for obtaining this literature is described in detail in Section 3.1. Section 3.2 introduces research in continuous deployment practices in Infrastructure-as-Code (see Section 3.2.1, p. 16), presents research on the customization of continuous deployment processes (see Section 3.2.2, p. 16) and analyzes and compares current IaC tools, their features and approaches (see Section 3.3, p. 17). Section 3.4 describes applicable studies on approaches for interpreting semantic changes of cloud infrastructure (see Section 3.4.1, p. 17), infrastructure representation (see Section 3.4.2, p. 18) and change detection (see Section 3.4.3, p. 19), followed by the visualization of said changes in Section 3.5.

3.1 Methodology

The research that supported this state of the art was performed across multiple trusted digital repositories of scientific articles and conference papers. Among them were IEEE Xplore, ACM

Digital Library, Scopus, ScienceDirect, Research Gate, and the Open Repository of the University of Porto. The official documentation of available Infrastructure-as-Code solutions was also consulted. The following core topics were used for this research:

- Infrastructure Change Analysis - including Semantic Analysis and Infrastructure Representation
- Identification of Changes
- Visualization of Changes for Review
- Continuous Deployment - including Custom Rules and Infrastructure-as-Code
- Similar Features of Current IaC Tools

This methodology followed a bottom-up strategy, which consisted of repeating the following steps:

- For each core topic of the problem, finding relevant keywords to form a search query (see Appendix A (p. 75) for the final keywords used)
- Searching in each selected repository using the mentioned search query and filtering the applicable results based on their title, keywords, abstract, and conclusions.
- Refining the search query and keywords based on this literature to narrow or widen the search scope or reach other relevant research.

3.2 Continuous Deployment

3.2.1 Infrastructure-as-Code

Meyer *et al.* [34] demonstrated how quality assurance can be integrated in continuous deployment, particularly for including community-made configuration scripts. This is achieved by running the scripts of *server configuration* tools (see Section 2.3.2.3, p. 10) in different platforms, analyzing the issues that arise, and displaying such information to script developers and consumers. A similar approach should be used to perform quality assurance of infrastructure security, ideally by analyzing the infrastructure definition instead of analyzing deployed configurations.

3.2.2 Custom Rules

Baset *et al.* [25] developed a rule-based tool for validation of configurations which applies to several environments, such as cloud, Docker images, and running containers. This tool uses a Configuration Validation Language (CVL) to define configuration validation rules. To validate the configurations, these rules are then run against configuration files, system properties (e.g. file/directory permissions, software packages), and other configurations in custom formats. In the context of this work, a similar approach will allow for validating infrastructure changes - instead of configurations - against user-defined rules.

3.3 Relevant Features of IaC Tools

For the purpose of this work, the considered IaC tools (see Section 2.4, p. 10) offer the following particularly relevant features:

Chef - offers a web app to audit and report on the compliance of a deployment against security standards. It also allows for automatic and customizable remediation of detected vulnerabilities[8]. Through the *why-run* option, Chef also allows comparing the current and proposed states of the system without applying changes.

Puppet - provides a configurable impact analysis tool for integration with CI/CD pipelines. It has a web interface that displays the affected resources of each environment and the modifications they will suffer [18].

Terraform - offers a command for listing the set of resource and configuration changes that will be executed upon deployment of the current code. This command also checks the current state of the deployed infrastructure beforehand, which allows it to detect *configuration drift*[21]. Terraform also allows the configuration of compliance rules by using the Sentinal policy-as-code framework[20]. (see Section 2.3.2.2, p. 9)

AWS CloudFormation - provides the ability to generate *change sets*(see Section 2.4.2, p. 11), which show the differences between a Cloudformation template and a previously deployed one. Through CloudFormation Guard it is also possible to restrict wanted characteristics of resources through user-defined rules[6].

AWS CDK - offers a *diff* command (see Section 2.4.3, p. 12), similar to terraform plan. However, no *configuration drift* is detected beforehand, comparing only the upcoming infrastructure with the last stack deployment. AWS CDK also has the ability to perform checks on the configured resources through the use of *Aspects*, which are scope-dependent operations that can be used to validate some security requirements in the stack.

Pulumi - similarly to AWS CDK and terraform, the *pulumi preview* command allows for visualizing the upcoming infrastructure resource changes. It can also compare against the actual currently deployed state.[15]

3.4 Infrastructure Changes Analysis

3.4.1 Semantic Analysis

Opdebeeck *et al.* [37] studied the adoption of semantic versioning in IaC code, focusing on third-party created sets of tasks. A large dataset of code changes was obtained to study their impact on versioning. The authors then developed a structural model for the IaC tasks, along with a domain-specific change extraction algorithm to determine structural changes between two versions of their code. These were then used to evaluate the impact of code changes in the resulting infrastructure

and predict how the semantic version should be increased. The version increments performed by developers were found not to follow guidelines consistently.

Menarini *et al.* [33] studied the applicability of aiding code review through semantic analysis of changes between code versions, allowing a developer to view behavioral differences during a peer-review process. By creating a tool that analyses the existing test cases, the reviewer is provided more context on the changes (namely differences of likely invariants). After conducting a user study with 18 developers, this additional semantic information was found consistently effective for detecting insufficient testing and finding bugs upon their introduction. The study also concluded that more meaningful review comments were made with such extra information due to the added context.

This research shows a positive general correlation between semantic aware analysis and code review performance.

3.4.2 Infrastructure Representation

Topology and Orchestration Specification for Cloud Applications (TOSCA) [46] is a standard developed by the non-profit organization OASIS to define the interoperable description of services and applications - either hosted on the cloud or elsewhere -, including their components, relationships, dependencies, requirements, and capabilities. This definition is done through *topology templates*.

Because it is agnostic to the underlying platforms and infrastructure, TOSCA enables portability and automated management across cloud providers. Consequently, it expands customer choice, improves reliability, and reduces cost and time-to-value. Moreover, and perhaps most relevant to this work, it allows for portability in continuous delivery processes[13].

In an attempt to facilitate the combination and integration of DevOps artifacts of different platforms within a cloud infrastructure, Wettinger *et al.* [48, 49] worked on an automated transformation framework to generate TOSCA standard-based models from two distinct types of artifacts of different platforms. This allowed them to orchestrate arbitrary artifacts through *topology templates*.

Similar to the TOSCA standard, Wurster *et al.* [50] introduced a metamodel (Essential Deployment MetaModel or EDMM) for representing declarative deployment models, agnostic of the underlying technology. The authors also provide mappings between existing IaC models (e.g., Puppet, Chef, Ansible and AWS CloudFormation) and this metamodel. Although EDMM only uses a subset of TOSCA's entities, it is also mentioned how it can be mapped to that standard. By supporting the migration of one technology to another, EDMM allows comparing technologies besides enabling platform-independent deployment automation research.

The models mentioned form a graph-based representation of the infrastructure. Such an approach is optimal for the use case of this work, as it will allow us to trace the impact of a given change to the affected resources or configurations. Furthermore, it will allow us to detect changes that affect multiple resources by finding matching subgraphs in the representation.

The mapping of the models in the literature above is made to generate platform-specific code from the technology-agnostic descriptions. However, for the purpose of this work, the inverse operation is required to allow analyzing infrastructure regardless of its original definition. Moreover, such mapping does not assume dependency relationships in some cases when they do not impact the resulting platform-specific code. This cannot be ignored when trying to trace the impact of changes in the infrastructure. As such, the approach of this work will have to go beyond simply performing the mentioned mapping inversely, and will have to include all relevant dependency relationships between elements of the infrastructure.

3.4.3 Identification of Changes

As mentioned briefly in Section 3.4.2 (p. 18), the representation of the infrastructure should follow a graph approach, allowing the use of graph-based algorithms to detect similar sets of resources or configurations that suffered the same changes.

Eder *et al.* [29], studied a technique for detecting and representing structural changes between two versions of a Directed Acyclic Graph (DAG), in addition to its application to temporal data warehousing. The authors describe several phases of this process, namely node matching, renaming detection, detection of inserts and deletes of nodes and edges, and change detection within nodes. Their approach, optimized for change rates below 10%, reveals positive results and they reveal confidence in its suitability for describing evolution in systems represented by DAGs. The problem in question, however, is admittedly a difficult one in the field of computer science.

Pham *et al.* [38] empirically studied a scalable, accurate, and complete clone detection approach in graph-based models. This was performed by developing a tool for Matlab/Simulink models. Their approaches did not consider the detection of overlapping subgraphs. However, for the proposed solution of this work, overlapping subgraphs should be considered in order to collapse changes that may have been caused by the same modification in configurations or resources.

Similarly to Pham *et al.*, Gabel *et al.* [30] present a scalable algorithm for semantic clone detection in the context of identifying similar code fragments, detecting both syntactic and semantic code clones, based on program dependency graphs.

3.5 Visualization of Changes for Review

Baum *et al.* [26] studied the response of developers in code review situations in order to optimize the ordering of the displayed changes for improving a review's efficiency and effectiveness. The authors analyzed how much the understanding of the changes was affected in multiple scenarios such as reading natural language texts, relationships between the changes, code structure, and the grouping of the changes.

3.6 Summary

In this chapter, we describe the methodology used for literature review (see Section 3.1, p. 15) and the findings concerning the current state of the art of the relevant topics of this dissertation.

Section 3.2 (p. 16) reviews studies found on continuous deployment in IaC (see Section 3.2.1, p. 16), revealing an approach for ensuring quality of third-party configurations in continuous deployment by running configuration scripts and analyzing possible issues. This section also supports the applicability of user-configurable languages for validating development configurations (see Section 3.2.2, p. 16), including but not limited to the cloud.

Section 3.3 (p. 17) analyzes the functionalities offered by some of the most popular IaC tools, showing a general availability of auditing and compliance tools for analyzing effective infrastructure state. However, these do not allow the analysis of proposed changes and are solely dependant on the final state of the infrastructure. Furthermore, the available tools appear to allow visualization of proposed changes, but have few optimizations for large sets of changes. Similarly to compliance checks, the classification of the displayed changes is based solely on the resulting infrastructure state.

In Section 3.4 (p. 17), topics regarding the analysis of infrastructure changes are discussed.

Little relevant literature was found on developing semantics-aware solutions for problems in IaC, as seen on Section 3.4.1 (p. 17). A solution has been studied for predicting IaC code changes' effects on semantic versioning, which takes into account included third-party code. Its results were inconsistent with semantic versioning guidelines. However, semantic analysis of code changes has been found to be effective for detecting introduced bugs and testing issues. Section 3.4.2 (p. 18) presents existing models for representing cloud infrastructure that can be mapped to specific IaC platform artifacts. Being able to perform this mapping should not be necessary to evaluate infrastructure changes, but these provide a good starting point for creating a generic and complete representation. Since these models are graph-based, Section 3.4.3 (p. 19) reveals techniques for comparing versions of a graph and extracting the occurred changes. Other algorithms have also been developed that allow detection of similar sub-graphs, which could be used to group similar sets of changes.

Finally, Section 3.5 (p. 19) demonstrates an increased effectiveness of the code review process when displaying information with different techniques.

Chapter 4

Preliminary Studies

To complement Chapter 3 (p. 15), a more subjective research process was conducted to validate this dissertation's proposal. This process consisted of a user study, where 10 industry professionals were interviewed (see Section 4.3, p. 22).

4.1 Motivation

This work intends to evaluate the effectiveness of automated classification and summarizing of effective changes in IaC. To better identify the existing problems and, consequently, better define the hypothesis and the research questions of this dissertation (see Section 5, p. 29), it is relevant to collect closer insight into the current workflows adopted by IaC developers, to determine how they could be impacted. This information also allows us to prioritize the parts of the proposed solution that are more likely to be adopted and fruitful.

4.2 Preliminary Research Questions

The preliminary studies described here were conducted to better understand the current real-world difficulties and approaches regarding continuous deployment in IaC and gather feedback from industry professionals concerning this project's initially proposed solution. This enabled answering the following research questions:

PRQ1 - What are the currently adopted workflows for managing cloud infrastructure?

PRQ2 - How adopted are third-party IaC contributions, and how is their risk assessed?

PRQ3 - How well would the proposed solution be received in existing development workflows?

PRQ4 - What types of use cases are there for automating infrastructure change approval?

PRQ5 - How can a diffing tool improve the efficiency of manual infrastructure change reviews?

4.3 Interview Methodology

The study was performed as semi-structured interviews (SSIs) [36], conducted in an informal setting and using the scripted questions for steering the conversation when necessary. In order to ensure comparable responses from the interviewees, additional more specific follow-up questions were designed for each discussion topic. This approach guaranteed the coverage of all relevant matters while allowing the respondent to explore each one freely.

4.4 Interview Structure

The 45-60 minute interviews were divided into two parts:

- Current Experience - Focused on collecting information regarding the interviewee's experience, their workflows regarding cloud and IaC. To avoid introducing bias in the results, the hypothesis and goal were only revealed to the interviewee after this section.
- Approach Discussion - Dedicated to gathering feedback regarding the initial proposed solution, including usefulness, use cases, and further suggestions.

Alternative routes in case the interviewee did not use cloud services or IaC were created but were not used. A comprehensive list and description of all interview questions is available in Appendix B (p. 77).

4.5 Study Results

4.5.1 Interviewee Demographic

The characteristics of the inquired interviewees follow the below information:

- A total of 10 developers, cloud consultants, and DevOps engineers of businesses ranging from single developer consultants to multi-national companies with thousands of employees. However, half of the participants represented companies of 20 to 200 developers, while 3 belong to smaller organizations. Their clients range from general consumers to large businesses.
- All interviewees were found to use the AWS cloud services provider, although half have previous experience with Google Cloud Platform and Azure.
- Developed products include serverless architectured infrastructures and critical back-end systems, where a wide spectrum of cloud resources are used (e.g., databases, message queues, serverless functions, network routers, file storage services). In some cases, there is also communication with on-premise systems and external tools.
- All participants had some experience in IaC, and those who provided further detail (4 out of the 10 interviewees), ranged from 1 to 3 years of experience with their currently adopted IaC technology.

4.5.2 Infrastructure Development Workflows

- Used IaC solutions include Terraform, Cloudformation, and CDK, with 9 of the participants developing to some extent using the latter. Previous experience with IaC tools also includes products such as Boto3, Jinja, Troposphere, and Serverless Framework.

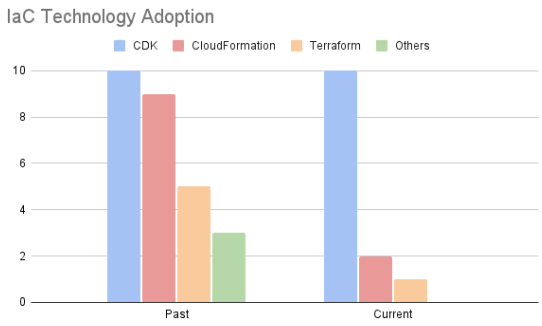

Figure 4.1: Reported adoption of IaC Technologies

- All interviewees integrated development workflows using CI/CD pipelines, and only one triggered the deployment of the infrastructure manually.

- Deployment frequency ranges from hourly to monthly, although in most cases it is done daily or a few times per week.
- Half of the participants have used third-party pieces of infrastructure code, of which 80% have used external ones. Risk assessment for such reuse relies heavily on traditional manual code analysis, although some also have automatic compliance checks through the cloud provider or IaC tool. Code is mostly reused by copying directly onto the codebase rather than including it through libraries or packages. This is done to have total control over the code and, consequently, some security reassurance.

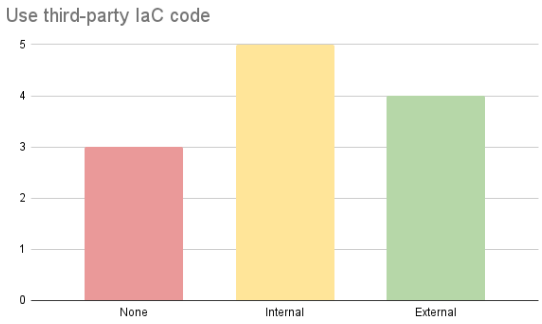

Figure 4.2: Reported usage of third party IaC code

- Since third-party code is currently reused mostly in small blocks to aid security reassurance, code changes contain small incremental infrastructure modifications and are reportedly quickly reviewed. Large changes, however, can take up to weeks to be reviewed, particularly in critical systems.
- The responsibility of approving infrastructure changes - for organizations of more than one person - is split between team developers and dedicated DevOps/QA developers. In all cases, multiple people have such authority to ensure reviewer availability.
- Half of the inquired people use additional manual review tools to analyze infrastructure changes beyond code review (e.g., CDK diff[3], CloudFormation guard[6])
- While all participants have multiple deployment environments in their pipelines, only half deploy to multiple regions and none deploy to multiple cloud providers.
- Half of the participants reported having frequent infrastructure changes that could be automatically approved, while only 3 would not benefit from automatic approvals of changes that have been approved for other environments.

4.5.3 Proposal Feedback

Figure 4.3 presents the obtained feedback about how useful each of the following abilities would be for their current workflow:

- Collapsing repetitive changes in a diff tool - consists of providing a summary of the infrastructure differences caused by a code change, where repeated changes are collapsed to aid readability of large changes.
- Automated approvals for specific changes - the ability to deploy infrastructure changes automatically only if they obey user-specified rules concerning its resources, configurations, and permissions.
- Automatic approvals based on past approvals - the ability to deploy infrastructure changes automatically in a given environment (stage or region) if such changes have previously been approved for a distinct specified environment.

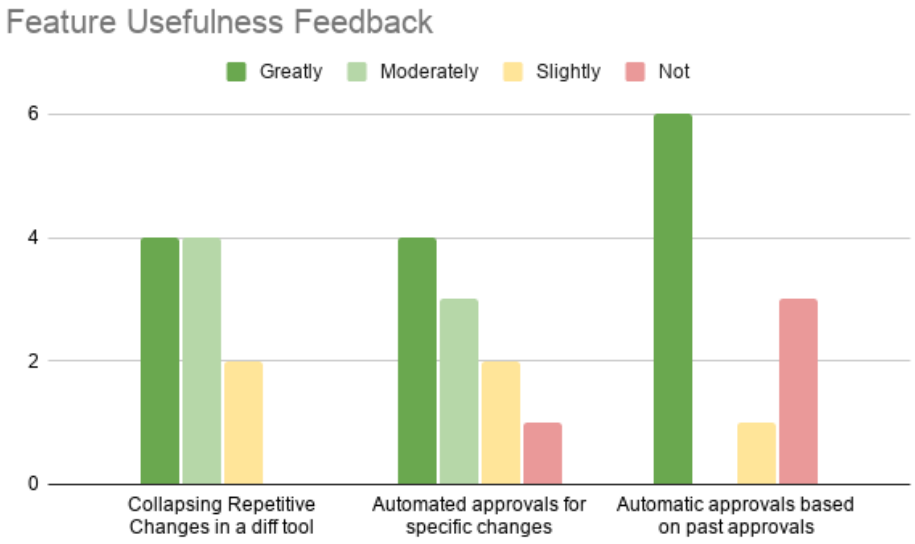

Figure 4.3: Reported usefulness of 3 proposed continuous deployment features for IaC code by number of responses in each degree of usefulness for the participant's workflow

The feedback obtained for the first two abilities demonstrates their positive impact in nearly all workflows while having a more significant impact in 70% to 80% of the cases. However, automatic approvals based on past approvals revealed a more split distribution of opinions, where 60% claimed to benefit from it greatly, and 40% would benefit only slightly or not at all from it.

The interviews' obtained responses revealed two main workflow scenarios regarding possible automation of infrastructure change approval: *approve by default* and *reject by default*. The first

consists of always approving the deployment of the infrastructure except if any user-defined rejection rule is violated. Similarly, the latter requires human approval unless all of the changes are specifically allowed in the user-defined rules.

Obtained examples of useful infrastructure change approval rules include:

- Rejecting removal of specific or general resources (e.g., databases, computing containers, user authentication services, file storage)
- Rejecting changes in exposed ports of a web server, protocols, configured domain names, etc
- Rejecting particular permission changes
- Rejecting encryption downgrading or creation of non-encrypted resources
- Rejecting resources in specific regions
- Approve the creation of specific or general resources (e.g., serverless functions, message queues)
- Approve configuration changes of resources (e.g., message queue configurations, small computing container size changes)

A participant made an additional suggestion for optionally associating a written justification or context to be associated with rules, which could then be viewed upon rule violation to aid interpretation.

Obtained examples of useful security threats to detect include, besides the approval/rejection rule examples mentioned above:

- Any computing resources that can execute arbitrary code. Possibly performing static and dynamic security and dependency scanning or checking base images.
- Cloud provider account grant access
- Any modification that results in higher cloud infrastructure maintenance costs.

Obtained examples of useful infrastructure change visualization optimizations include:

- Prioritizing shown changes according to the type of resource (stateful resources should have higher priority) and type of modification (removal, creation, update).
- Taking advantage of traditional **left vs. right** code comparison standard for intuitive analysis.
- Signaling the proposed changes in a diagram representation of the infrastructure, including resource relationships.
- Collapsing changes based on abstraction hierarchy in the code.

- Revealing the code files that generated each change.

A code annotation feature to provide context to a set of defined resources in the IaC code - which would then be included in the summary of infrastructure changes to aid readability - was considered beneficial by 70% of the interviewees.

4.6 Summary

This chapter describes and evaluates the results of the preliminary study performed on 10 industry professionals. This required conducting semi-structured interviews to evaluate how developers manage their development, review, and deployment workflows in IaC, as well as gather feedback regarding the initial proposed solution, including usefulness, use cases, and further suggestions (see Section 4.4, p. 22).

Section 4.5.2 (p. 23) revealed that all interviewees integrated their development workflows with CI/CD pipelines. Only half of the inquired people reported using tools for additional manual review and an overall tendency was observed towards reusing community-made code, despite the difficulty in ensuring safety when doing so.

Section 4.5.3 (p. 25) demonstrates a generally positive feedback regarding the proposed solutions of collapsing the presented repetitive changes, automating approvals for specific changes, and automating deployment approvals based on previous approvals. Furthermore, it contains additional relevant insights collected from the performed interviews.

Chapter 5

Problem Statement

5.1 Problem

Like all code, third-party IaC implementations can be easily shared and included in one's code, either by manually copying blocks of community-created code or including external dependencies, which can speed up developer operations significantly, but can also present security concerns.

IaC code may also be integrated in a Continuous Integration and Continuous Deployment (CI/CD) pipeline, which stresses the importance of a highly effective change review process, such as peer reviews or automatic validation tasks in the pipeline.

In Chapters 3 and 4, we observe the emerging availability of community-created IaC code. Its adoption is suffering a natural hesitation due to potential security risks and the necessary time-consuming review process typically required for deployment in production environments. The referenced literature has revealed how little information there is concerning potential generalized solutions to this problem, particularly regarding automated change approval and classification approaches or optimization of the manual review process.

Existing IaC solutions offer tools to review the upcoming infrastructure changes of an IaC deployment, but they are designed for small iterations rather than optimizing the readability of large changes. Moreover, this review is an additional step separate from code review. As such, it

is often used for manually approving changes during a CI/CD pipeline execution, interrupting it, and causing delays in the development process.

Typically this results in teams not including community-made IaC code unless it is small enough to be reviewed or they fully trust its source (see Section 4.5.2, p. 23). Consequently, this greatly reduces their adoption and hampers the search for usable existing solutions to particular development problems.

With this research, we will focus on how to review and analyze the security risk in large IaC code changes efficiently, particularly when introducing third-party code. Large code reviews are very time-consuming and may be unable to detect subtle dangerous resources, configurations, or permissions.

5.2 Hypothesis

This work intends to validate the following hypothesis:

> *The productivity and security of declarative IaC code deployments can be increased by: a) automating the approval or rejection of specific infrastructure changes in a continuous deployment pipeline; b) generating a comprehensive summary of upcoming infrastructure modifications, suitable for manual review of large changes. These methods provide the highest impact when changes are hard to detect in regular code reviews or introduced in third-party code.*

The mentioned summary of the upcoming infrastructure changes would display the changes according to their impact on the infrastructure, while condensing repetitive occurrences and highlighting vulnerability-prone ones. The goal is to allow developers to view the effective changes to the infrastructure, even if they are caused by updates to third-party dependencies or a byproduct of refactoring attempts. This ideally decreases the ability to hide dangerous infrastructure changes in the code.

Automation of change classification, both in regards to their potential impact on the infrastructure or to whether they should require manual review, should be based on customizable user configuration and adaptable to the specific project, team, or company where it is used. By including such automated classification into a continuous deployment pipeline, certain deployments could be approved or rejected without requiring human intervention. In other words, infrastructure deployments with non-compliant modifications would automatically be canceled and the ones with indubitably safe changes would be deployed with no need for human review. This allows for ensuring consistent standards across development iterations, detecting potential threats, and increasing the efficiency of frequent similar deployments.

5.3 Research Questions

This work attempts to answer the following research questions, extracted from the hypothesis in Section 5.2 (p. 30):

ERQ1. Can code flaws introduced in IaC code changes be avoided by performing peer-review?

ERQ2. How can the peer-review process in IaC be facilitated to minimize undetected code flaws?

ERQ2.1. What effect do the summary and automated classification of changes during code review have on the percentage of accurately detected changes? What effect does it have on both security-threatening and functionality-breaking changes?

ERQ2.2. What effect do the summary and automated classification of changes have on how confident developers are in detecting all impactful changes? What effect does it have on both security-threatening and functionality-breaking changes?

ERQ2.3. What effect does the extra information provided by the summary and classification of changes have on how much time the code review takes? Does having changes pre-approved and risk-sorted decrease the amount of time spent? Does it decrease the amount of effort allocated to non-impactful changes?

ERQ3. Is it possible to automate the change review process in IaC? Can it be fully automated in a CI/CD pipeline?

5.4 Validation

The validation of the hypothesis encompasses:

1. Identifying the features of a tool that can support the peer-review of IaC code changes - including the summary and configurable automatic approval of upcoming changes, as described in Section 5.2 (p. 30).
2. Implementing such functionalities in a reference implementation (see Chapter 6, p. 35)
3. Conducting a user survey, targeted at industry professionals, to validate the adopted techniques and proof-of-concept tool.

This experiment entails the following:

- Using three codebase modifications of different sizes:

  A small set of changes to understand the impact of the solution in a more incremental development

  Two larger sets of changes, affecting multiple files, to evaluate the effectiveness of the proposed solution

- Providing the codebase modifications to a sample of 10-20 IaC developers, while only allowing half of the participants to access the summary and define automatic approval rules. This will ensure a control group of 5-10 people.
- Requesting the developers to review each of the three sets of code changes.
- Analyzing the performance differences between the control group and the remaining participants, for each of the three experiments.

To perform this experiment, a tool was developed as a proof-of-concept, intended to be used in continuous deployment pipelines. It generates user-friendly reports for human review and automatically asserts approval or rejection of infrastructure changes based on user configuration.

5.5 Proposed Solution

Solving this problem requires, as a starting point, the definition of a semantic representation of the infrastructure and the interpretation of code changes between two versions of a codebase - Section 6.1.1. With this mechanism, the gathered information should be summarized for human review - Section 6.1.2. The automatic approval or rejection of infrastructure changes should then be established based on user-defined rules - Section 6.1.3. Lastly, it should be possible to detect semantically identical changes for improving the efficiency of both the review process and automatic approvals - Section 6.1.4.

5.6 Validation

In order to validate the hypothesis, a reference implementation of the strategies that fulfill the points in Section 5.5 should be developed. With access to such a proof-of-concept implementation, a survey targeted at industry professionals should be conducted. The goal of the survey is to evaluate the efficiency and vulnerability detection success when the workflow includes the summary and configurable automatic approval of upcoming changes, as described in Section 5.2 (p. 30).

This experiment would entail the following:

- Using three codebase modifications, mixing real-word and manually designed changes, of different sizes:

  A small set of changes to understand the impact of the solution in a more incremental development

  Two larger sets of changes, affecting multiple files, to evaluate the effectiveness of the proposed solution

- Providing the codebase modifications to a sample of 10-20 IaC developers, while only allowing half of the participants to access the summary and define automatic approval rules. This will ensure a control group of 5-10 people.

- Requesting the developers to review each of the three sets of code changes.
- Analyzing the performance differences between the control group and the remaining participants, for each of the three experiments.

To perform this experiment, a tool was developed as a proof-of-concept, intended to be used in continuous deployment pipelines. It generates the user-friendly reports for human review and automatically asserts approval or rejection of infrastructure changes based on user configuration.

Although a generic solution was developed, the AWS CDK framework and AWS ecosystem were considered for validation purposes. Consequently, the tool was adapted to work with this technology.

Chapter 6

CDK Change Analysis Tool

In order to validate the formulated hypothesis, a tool - the **CDK Change Analysis tool (C2A)** - was developed, which generates a visual report of changes based on two declarations of infrastructure states and optionally a custom set of change classification rules. Although C2A was developed to be easily extended and support any declarative IaC technology (see Section 2.3.2.1, p. 9), the current version supports solely the AWS CDK and its underlying platform, AWS CloudFormation, since they both define infrastructure state through CloudFormation Templates (see Section 2.4.2, p. 11).

Section 6.1 (p. 36) outlines the goals of this tool, while the remaining sections describe the decisions and approaches in the developed reference implementation.

The development of this proof-of-concept was split into the following milestones:

- Infrastructure Model Design - Designing a representation of every component of an infrastructure state and their relationships.
- IaC Declaration Parsing - Instantiating the infrastructure model from existing declarations of infrastructure state.

- Detection of changes between infrastructure states - Comparing two infrastructure models to predict the effective changes.
- Visualization of the Infrastructure Model and its Changes - Grouping changes based on change characteristics and displaying them in a web application.
- Classification of changes - allowing user configuration to classify risk and pre-approval behavior of changes

6.1 Proposed Solution Overview

6.1.1 Semantic Representation and Change Interpretation

In order to improve the continuous deployment experience with IaC, it is necessary to be able to interpret semantic changes between versions of the code, so they can later be displayed and allow automation tasks. To do so, there should be an optimal internal semantic representation of the infrastructure for a given version, encompassing the resources, configurations, and permissions resulting from its deployment. This representation, similar to the existing models for platform-agnostic IaC definition, should follow a graph-based approach, connecting dependent resources and configurations.

```
{
    stack_id: ...,
    stack_cloud: ...,
    ...
    resources: [
        {
            resource_id: ...,
            resource_type: ...,
            resource_configurations: {},
            depends_on: [],
            dependents: [],
            hierarchy_path: ...
        }
    ]
}
```

Listing 6.1: Example of an initial semantic representation

6.1.2 Change Visualization and User Review

In order to improve the efficiency of human review, a visualization of the semantic infrastructure modifications should be provided, including permission changes, in a readable and concise manner. Ideally, it should also highlight breaking changes to further prevent unintended deployments.

This visualization should initially be a high-level summary of the changes for interpreting large changesets, but be able to expand the provided information as needed during the review. To allow such interaction in an efficient manner, it will be developed in the form of a web application.

6.1.3 User-defined Change Approval

The user should also be allowed to configure which types of changes should be automatically approved for deployment, as well as those which should be rejected. It should also allow automating approvals based on previous deployments.

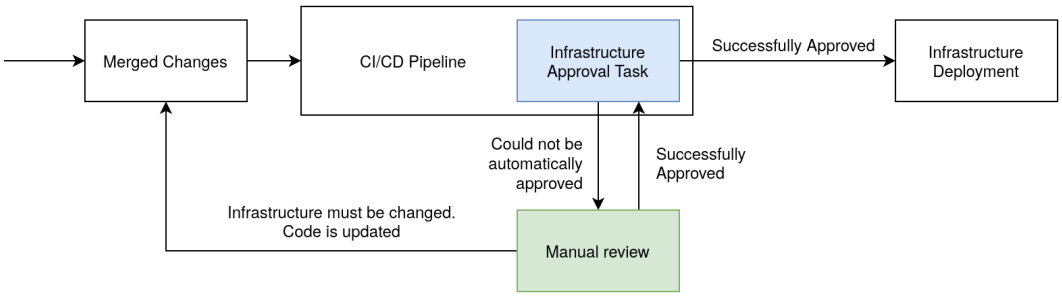

Figure 6.1: Automated approval in a continuous deployment workflow

```
{
    auto_approval: [ //configure automatic approvals
        {
            resource_type: "aws.dynamodb",
            changes: [
                {
                    // automatically allows creating DynamoDB instances
                    type: "create_resource"
                },
                {
                    type: "config_update",
                    config: "partitionKey",
                    config_rule: {
                        changes: [
                            { // automatically allows changing 'partitionKey' type
                                to STRING
                                type: "config_update",
                                config: "type",
                                before: [],
                                after: ["STRING"]
                            }
                        ]
} } ] } ] }
```

Listing 6.2: Initial draft change approval configuration

6.1.4 Change Detection

The last core aspect of the proposed solution is that semantically similar changes should be detected, both to reduce visual clutter in the change visualization (see Section 6.1.2, p. 36) and to improve the identification of possible automatic approvals (see Section 6.1.3, p. 37). This should be done by performing clone detection on two instances of the graph-based semantic representation(see Section 6.1.1, p. 36)

6.2 Design and Technology Decisions

This tool was developed in the programming language of Typescript on the Node.js Javascript runtime. This choice was due to the following:

- The core of the AWS CDK is developed on these technologies and, as such, increases future maintainability by the AWS CDK team. This fact also allows reusability of implementations or resources previously done for the AWS CDK product, which proved to be particularly relevant when parsing infrastructure definitions of the target platforms (AWS CDK and CloudFormation).
- Typescript being a strongly typed language increases maintainability over standard Javascript.
- Quicker development iterations due to reusability of modules between the Front-end web application and the core infrastructure analysis implementations, besides having previous experience with the technology.

Throughout the development of the C2A tool, there was a big focus on remaining as independent from the IaC technology (AWS CDK and CloudFormation) as possible, so it can be easily expandable to support other platforms and use cases. Hence, there is only one module that holds platform-specific logic: *platform-mapping*. Its responsibility is to map each supported platform's declarations of infrastructure into the C2A's platform-agnostic infrastructure model - *infra-model*.

The purpose of this isolation of responsibilities is to ensure that any supported platform can be easily added and kept up-to-date without affecting the change detection logic, visualization, change classification, or the remaining platform integrations. Therefore, any technology-specific maintenance is kept under its own submodule and all other logic is done over C2A's infrastructure model. A clearer view over this separation of concerns can be found on Figure 6.2 (p. 39).

6.3 Infrastructure Model

The definition of the infrastructure model and all of its entities is kept isolated in the *infra-model* module (see Figure 6.2, p. 39).

As previously mentioned in (see Section 3.4.2, p. 18), models for describing cloud infrastructure already exist, such as the TOSCA standard or EDMM. These representations attempt to

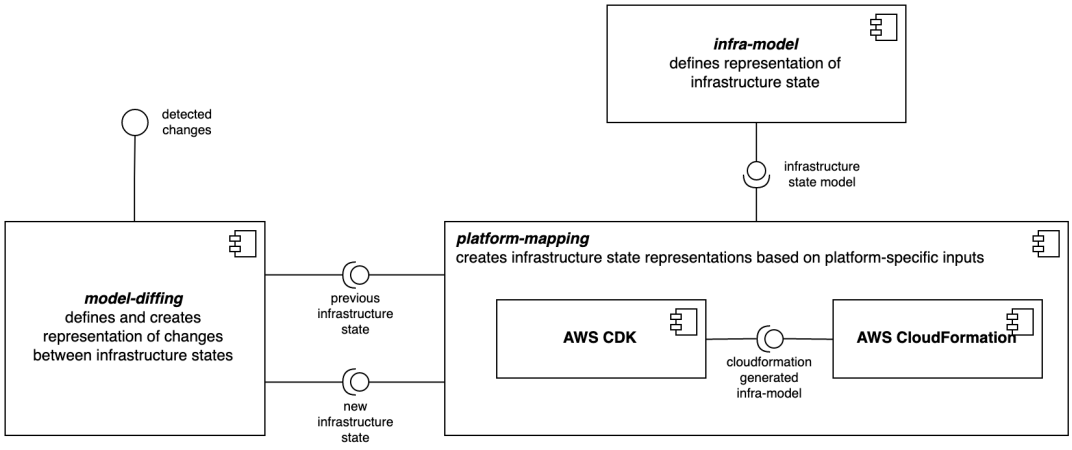

Figure 6.2: C2A - Separation of concerns of software modules

allow a generic definition of infrastructure resources or configurations that can be mapped onto platform-specific IaC code. In this work, however, the purpose of the model is not to be transformed into platform-specific descriptions of the infrastructure, but to instead infer performed changes between two states of such a model.

As a first approach in designing the model, the existing representations from the literature were taken as a starting point and iteratively reduced in complexity by removing unnecessary distinctions of entities. However, this proved to be a difficult task, as one cannot fully predict whether certain details of the model will be necessary for a comprehensible description of the detected changes.

Ultimately, the chosen approach consisted of taking the core entities from EDMM: Components, Relationships, and Properties - which in the TOSCA standard have the nomenclature of Nodes, Relationships, and Properties. By starting with a simple model and adding complexity as necessary, a simplistic model was achieved while keeping the standardized underlying structure. Figure 6.3 (p. 40) depicts the original model considered upon the first implementations of platform parsers.

Figure 6.3 (p. 40) also reveals a distinction between Structural and Dependency Relationships.

- **Dependency Relationships** allow tracking components that might be affected by changes on other components. The two main use cases for this relationship in AWS CDK and AWS Cloudformation are explicit dependencies or implicit ones in intrinsic functions (see Section 2.4.2.1, p. 11).

- **Structural Relationships** have no behavior and purely allow linking components hierarchically for better user understanding of the infrastructure. In the case of the AWS CDK, this can be mapped directly to the construct hierarchy (see Section 2.4.3, p. 12). In standard AWS CloudFormation templates, this concept of hierarchy could be used to represent Nested Stacks, for example (see Section 2.4.2.2, p. 11).

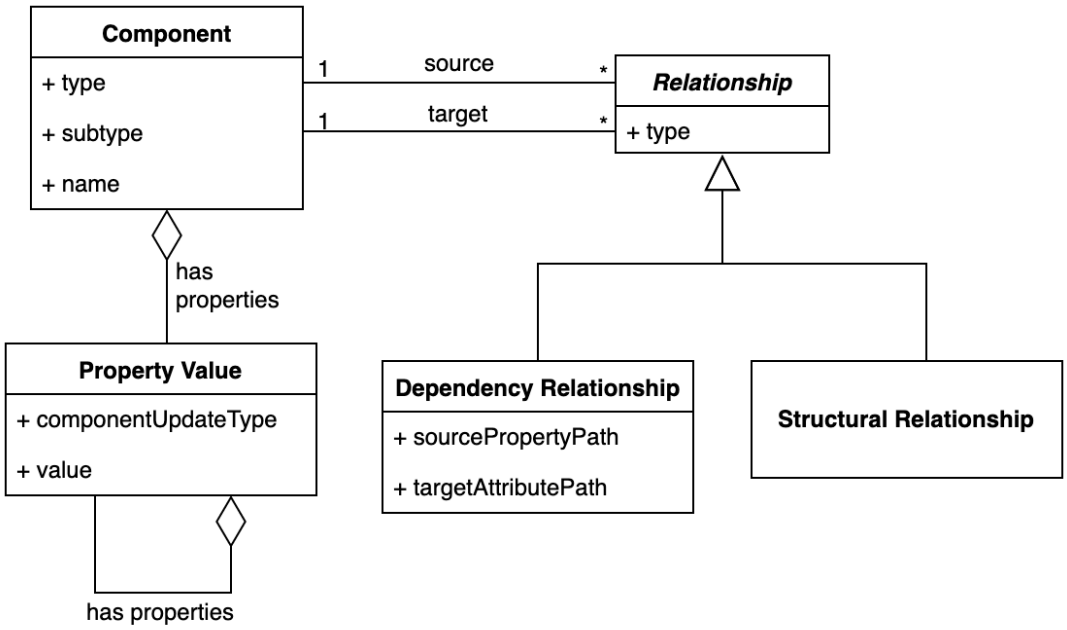

Figure 6.3: Initial simplistic infrastructure model, with Components, Relationships, and Properties.

A **Relationship** instance references its *source* and *target* **Components**. As such, a given infrastructure state - the ***InfraModel*** - is represented by a list of **Components** and a list of their **Relationships**.

Property Value represents the properties of components and can also be a collection of other *Property Values*, hence the aggregation line to itself in Figure 6.3. Property Values also contain information on how potential changes may affect their Component, as specified in the *componentUpdateType* field. There are three types of values for this field:

- **None** - Changes to the property have no consequence to the component itself.
- **Replacement** - Changes to the property cause the resource or configuration represented by the Component to be deleted and recreated, instead of mutated.
- **Possible Replacement** - Changes that may have the same behavior as in "Replacement", but such behavior cannot be confirmed (e.g. due to unclear specifications or dependency on unavailable data).

See Section 6.8 (p. 52) for how this model was later adapted to allow easier implementation of change classification.

6.4 IaC Declaration Parsers

As Section 6.2 (p. 38) suggests, the parsing of infrastructure definitions was implemented for AWS CloudFormation templates, since this format is used in both AWS CDK and CloudFormation to define the target desired state of the infrastructure. These templates can be declared in either JSON or YAML data formats (see Section 2.4.2, p. 11) which can be directly mapped onto Javascript objects. This fact allowed the parsing process to focus on validating and mapping CloudFormation concepts onto the developed *InfraModel* instead of focusing on data deserialization. Since the AWS CDK only produces JSON CloudFormation templates, and since YAML-based ones also require parsing the available short forms (see Section 2.4.2.3, p. 12), the parsers currently only support JSON inputs (see Section 8.4, p. 72). This allowed for reduced development overhead.

6.4.1 Basic CloudFormation Parsing

AWS CloudFormation templates have lists of different entities, including Resources, Parameters and Outputs (see Section 2.4.2, p. 11). Each of these entities in the template is used to instantiate subclasses of **CFEntity** - creates the *InfraModel* Components corresponding to their CloudFormation template entities and their respective relationships. Figure 6.4 describes this structure in more detail.

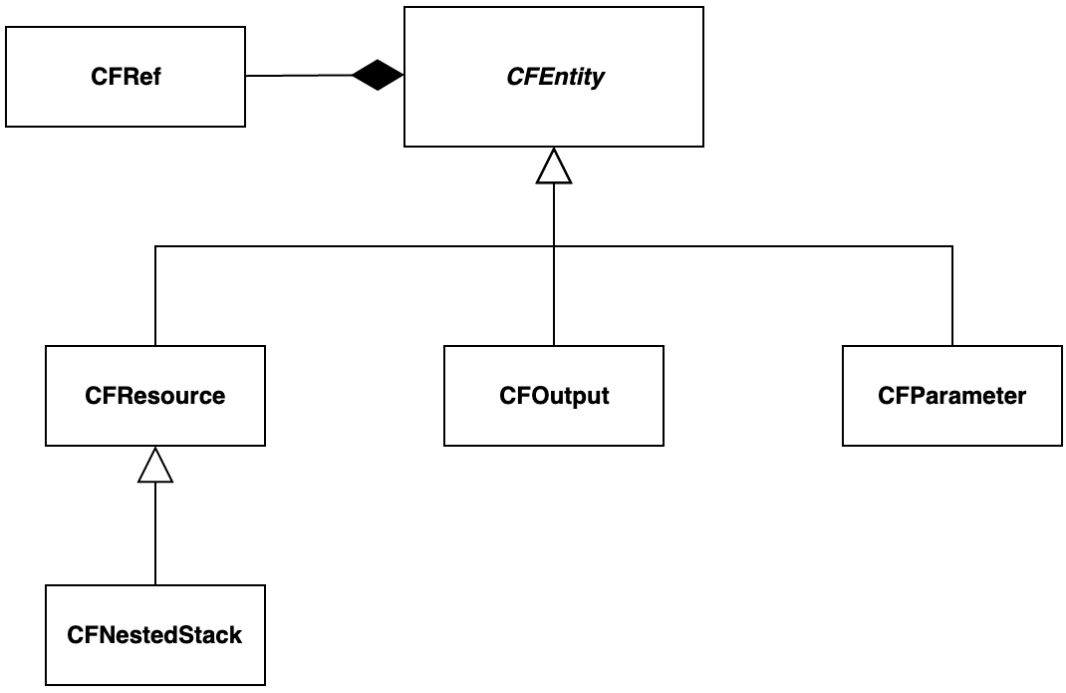

Figure 6.4: CloudFormation parsing helper classes - component diagram

The *CFRef* class extracts references to entities in an entity's declaration, from the used intrinsic functions (see Section 2.4.2.1, p. 11) and resources' *DependsOn* field.

The CloudFormation parser builds instances of *CFEntity*'s subclasses, which have the responsibility of properly building the respective components, property values, and outgoing dependency relationships.

The type of CloudFormation entity (Resource, Parameter, or Output) gets mapped to the type of **Component**. In the case of CloudFormation resources, in particular, their type gets mapped to the **Component**'s subtype (i.e. an AWS Lambda Function resource generates a **Component** with type "Resource" and subtype "AWS::Lambda::Function").

The CloudFormation definition of each entity is mapped directly onto each Component's *Properties* field, which is a Component Property Value. From this definition, *CFRef*s are extracted and mapped to **Dependency Relationships** where the target is the referenced entity. These references are obtained either from CloudFormation intrinsic functions or the *DependsOn* parameter of CloudFormation resources (see Section 2.4.2.1, p. 11). In the first case, the expressions are currently not evaluated in an attempt to save development time, since direct references can still be detected as they are always preceded by a given set of keywords (e.g. Ref, Fn::GetAtt). In other words, instead of reasoning on the behavior of the different intrinsic functions to determine which references will in fact be used for instantiating the *Stack*, the tool assumes all references from these keywords anywhere in the expression. This does not guarantee that a referenced entity is used once the *Stack* is ultimately instantiated by CloudFormation and the intrinsic functions are evaluated (e.g. when used in conditional intrinsic functions), but it is a strong indicator.

As CloudFormation does not offer a clear hierarchical structure of its entities, **Structural Relationships** are used purely to describe CloudFormation Stacks and Nested Stacks (see Section 2.4.2.2, p. 11). All Components originating from CloudFormation entities in a template file are linked to a Component that represents the whole stack, through a **Structural Relationship**. In the same way, a Nested Stacks' inner entities are linked to the Nested Stack resource of the referencing template (caller). This resource also defines the outside scope available to the Nested Stack's inner entities through its parameters and can be used by the caller to reference its outputted values.

Nested Stacks, however, require getting their respective templates, which are often located remotely. For the purpose of this work, no automated mechanism to fetch this extra information was implemented and these additional templates have to be manually provided to the CloudFormation parser (see Section 8.4, p. 72).

This module of the tool also determines the values of the *componentUpdateType* field of property values (see Section 6.3, p. 38) due to implicit consequences of property changes in CloudFormation. Changes to some properties in CloudFormation resources may lead to the re-deployment of those resources. These are obtained from a CloudFormation specification found on the AWS CDK repository [10]. This specification contains the mutability of each field of the supported resources. In other words, it states whether changing a field requires updating its parent. This is then converted into whether changing a field requires deleting and recreating the resource entirely by recursively following the immutable field's parent until reaching a mutable one - in which case the *componentUpdateType* is "None" - or the root of a resource's properties - in which case the

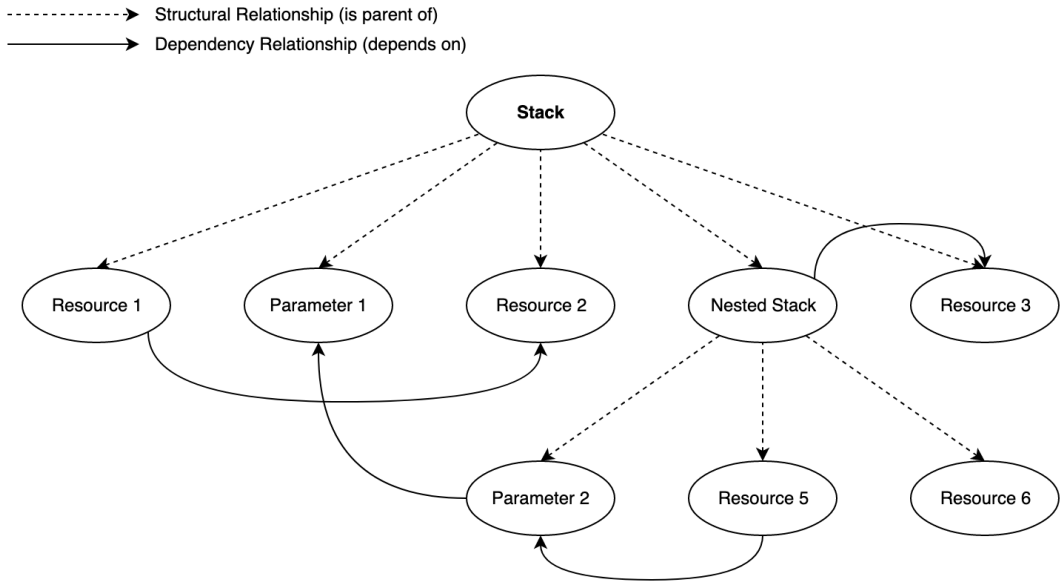

Figure 6.5: InfraModel structure example resulting from execution of the CloudFormation parser

componentUpdateType is "Possible Replacement" or "Replacement" depending on whether a field with "Conditional" mutability was found.

6.4.2 AWS CDK Parser

Despite providing a development experience based on imperative programming languages, AWS CDK code produces a definition of the target infrastructure state in a CloudFormation template format through synthesis (see Section 2.4.3, p. 12). This process also generates additional artifacts, which are currently not taken into consideration since they have little impact on the resulting infrastructure, although they could be used in the future to complement the contextual information provided in the CloudFormation template (see Section 8.4, p. 72).

The AWS CDK generates CloudFormation templates with additional information that is useful for this tool: the *AWS CDK paths* of every CloudFormation entity that logically fits into the *Construct*-based structure of the AWS CDK. These paths contain the identifiers for the ancestors (i.e. Stacks, Constructs, Resources) of each entity in the AWS CDK construct tree (see Section 2.4.3, p. 12), which can be used to infer the construct tree itself - excluding, of course, the branches that do not generate any CloudFormation entity, which are not relevant for analyzing the effective infrastructure changes.

The AWS CDK parser, then, starts by executing the CloudFormation parser (see Section 6.4.1, p. 41) on the synthesized CloudFormation template. On top of the obtained *InfraModel*, this parser then adds a Component for each element in the construct tree, obtained from the AWS CDK paths, and links them through **Structural Relationships** to build the hierarchy. The leaves of the tree are also connected through **Structural Relationships** to the corresponding CloudFormation entities. An example of the resulting *InfraModel* can be seen on Figure 6.6 (p. 44).

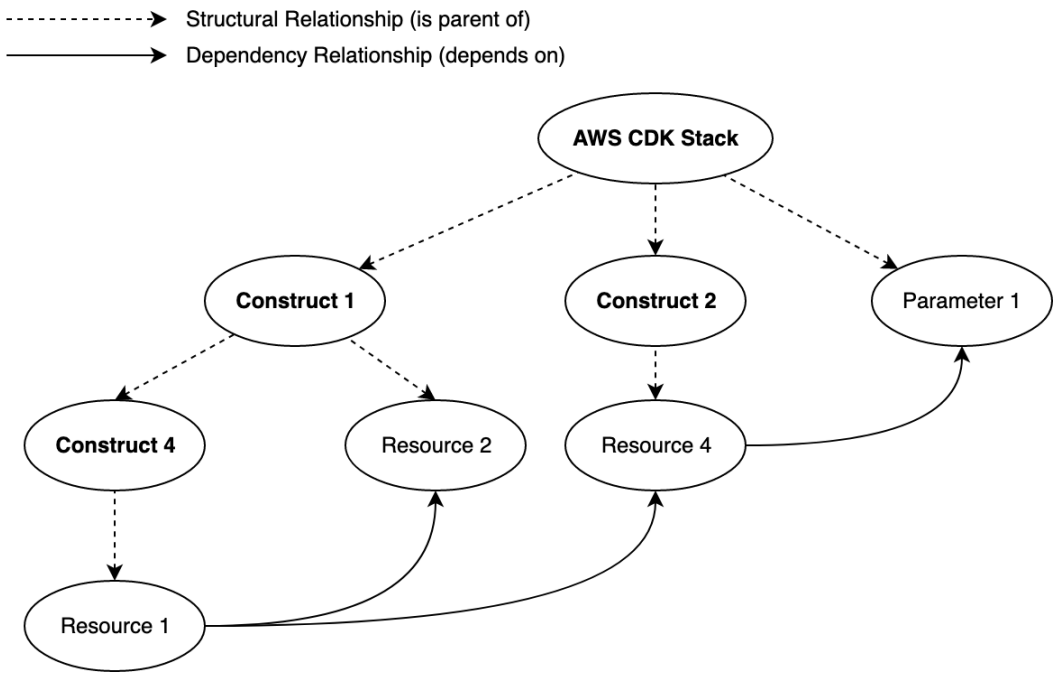

Figure 6.6: InfraModel structure example resulting from execution of the AWS CDK parser

6.5 Detection of Changes between Infrastructure Models

The main goal behind this tool is to provide relevant insight into the effective changes of infrastructure definition. This includes not only detecting which Components were removed, inserted or modified, but also which properties were changed and in what way. For the following sections, we will refer to the two versions of the infrastructure definition, and their generated *InfraModel*'s as ***old*** and ***new***. We introduce the concept of **Transition** to describe an entity (e.g. Component, Relationship, Property) over time, containing its two versions, old and new.

The current *InfraModel* defines a graph where the components are vertices and relationships are edges. The methods obtained in the researched papers (see Section 3.4.3, p. 19) describe approaches for Directed Acyclic Graphs (DAGs). Although the *InfraModel* does not explicitly disregard cyclic graphs, its semantics exclude this possibility, both in dependency relationships and structural ones. Based on this research, the following high-level algorithm was designed:

- Match Components with the same *name* if their similarity is above a given threshold.
- For the remaining Components, calculate the similarity for each potentially matching pair, matching those above the threshold as renamed ones
- Consider remaining unmatched old Components to have been removed and the new ones to have been inserted

Component similarity is a value in the unit interval (any real number between 0 and 1), where 1 represents exact copies and 0 represents totally dissimilar Components. Components with different *type*s or *subtype*s are immediately considered to have similarity 0, since semantically they cannot match. Otherwise, calculating this value is currently only based on the similarity of a Component's *Properties*. Taking a Component's relationships into account was considered, in addition to the similarity of their targets, but throughout multiple experiments with different CloudFormation templates this showed no improved accuracy in matching, in addition to reducing this analysis's performance. This might not be the case when applied to other IaC technology approaches, however, leaving the future possibility of performing this comparison in topological order (of either dependency or structural relationships) and taking into account the similarity of the entire incoming subgraph (see Section 8.4, p. 72).

6.5.1 Component Property Similarity

Analyzing the similarity of a Component's properties takes a similar approach to the comparison of the Components themselves. When calculating this similarity, an auxiliary weight should be used, which specifies the number of primitive values in a given property. The overall calculation follows the following steps:

- If the properties are primitive values, their similarity's weight is 1 and calculated in the following manner:
  - If they are of different types or completely different values, the similarity is 0.
  - If the values are exactly alike, the similarity is 1.
  - If the properties are both strings, the similarity of the strings is considered. In this tool, this similarity is calculated with Dice's Coefficient algorithm, since it provides a normalized similarity in O(m+n) time complexity.
- If the property is a collection of values (either a list or a key-value map), the weight is the sum of the weight of its elements and the similarity is the weighted average of the elements' similarity, where the matching of elements is performed as follows:
  - Match same-key elements if their similarity is above a given threshold.
  - Match remaining unmatched elements in pairs by the most similar, if their similarity is above the threshold.
  - Consider unmatched old elements to have been removed and the new ones to have been inserted (both have similarity 0).
- If the properties are not both primitives nor both collections, then they are considered to be different. The similarity will then be 0 and the weight will be the sum of the weights of both new and old versions.

Over different experiments, the property similarity algorithm evolved to improve accuracy. This included:

- Giving a similarity of 1 and weight of 1 to unchanged object keys of matching values - this ensures property structure has an influence on similarity.
- Disregarding array indexes when looking for matches - arrays are often reordered or elements are introduced or removed, which removes relevance of matching by index.

6.5.2 Change Propagation

Certain changes may have indirect side effects, which also need to be identified.

One example of such changes is updating the name of a 'resource' Component. Since these identifiers are used by CloudFormation to uniquely identify a resource, renaming them will lead to the deletion and recreation of such resource. Therefore, for every such renaming operation, a new replacement operation is instantiated for that Component. Another cause for these replacement operations is the change of properties with "Replacement" or "Possible Replacement" *componentUpdateType* values. In the case of AWS CloudFormation (and AWS CDK), this is derived from the CloudFormation specification as mentioned in Section 6.4.1 (p. 41).

Replaced Components, in turn, may result in new attribute values, updating the values of properties that reference them even though this reference might not have been lexically changed. To tackle this, additional property update operations are created for every property that references a replaced Component's attribute.

This process of detecting consequential changes is, necessarily, performed after matching all entities between the *InfraModels* and detecting the direct changes suffered. To distinguish them from direct changes, they are assigned a *cause*, which indicates the originating change, if such change exists.

6.5.3 Notes on Change Detection

Originally, this module detected changes in Relationships (updated, removed, or inserted). However, this was found to provide redundant information in the final changes report. The root causes for Relationship changes are more easily perceptible through the underlying causing changes. Furthermore, tracking of these changes did not increase the ability to automate classification or approval behavior (see Section 6.8, p. 52).

6.6 Tool output

The output of this tool has two main goals:

- Provide a quick manual review of the effective infrastructure definition changes (see Section 6.7, p. 47)

- Allow automation of change approval based on user-defined rules (see Section 6.8, p. 52)

This output can be useful in either of the following scenarios:

- In local development before committing code changes.
- When reviewing a set of proposed changes in a collaborative code. development environment (ideally accessed through an automated process in a continuous integration pipeline)
- When executing the infrastructure deployment in a continuous deployment pipeline, as a manual step. Reasons for taking this approach include:
  - using the same code in different environments - i.e. merged code automatically gets deployed on a testing environment but must be approved in this tool before getting deployed to the production environment.
  - requiring approval of a different team, such as Quality Assurance or Security.
  - performing additional checks against a target deployment environment, such as evaluating or fixing configuration drift (see Section 2.3.2.2, p. 9). Such features are unavailable in the current proof-of-concept (see Section 8.4, p. 72)).

6.7 Visualization of changes

When creating a visualization of the changes, several aspects were taken into account based on the state of the art and the results from the preliminary studies:

- Collapsing repetitive changes
- Collapsing changes based on abstraction hierarchy
- Follow commonly used standards for code comparison interfaces
- Grouping changes by type of resource and type of modification
- Allow semantic navigating of Components and their Relationships

With this in mind, two main views over the changes were devised: one focusing on characteristics (see Section 6.7.3, p. 50) of the changes and one focusing on the semantic structure of the Components, according to their Structural Relationships (see Section 6.7.4, p. 51).

This interface would ideally be built as a Graphical User Interface (GUI) on a web application, or as a Command Line Interface (CLI). Although the latter would help recognize important changes locally, it would likely provide harder navigation and a steeper learning curve in comparison with the GUI. Instead, to make the interface more appealing and easier to interpret, it was built as a web application. This also allows it to be easily shareable and used across device platforms, as well as seamlessly integrated with collaborative code workflow contexts.

6.7.1 Aggregations

To collapse repetitive changes while grouping them by resource type and type of modification, changes are grouped in ***Aggregation***. *Aggregations* identify changes with a given set of characteristics and can have sub-aggregations with more specific characteristics, forming a tree of characteristics that group changes (see Figure 6.7, p. 49). If an aggregation has sub-aggregations (i.e. is not a leaf), all of its changes are guaranteed to be included in its sub-aggregations.

Aggregations are currently generated by splitting the changes over multiple levels of increased specificity. Each level is defined by modules (referred to as Aggregation Modules or AMs) that are responsible for grouping changes based on their characteristics. Currently, there are only equality-based modules, which means aggregations are grouped on characteristics with the same value. Please note that *Entity* in this section refers to a Component or one of its Properties. The modules per level are the following:

- **Risk** - with values *High*, *Unclassified* and *Low*. This module was based on the user classification of changes (see Section 6.8, p. 52).
- **Type and subtype of the affected Component**.
- **Component Operation Type** - the type of change to the component (Insert, Remove, Update, Rename or Replace).
- **Operation Entity** - whether the change was to a Property or the Component itself.
- **Entity Operation Type** - the type of change to the operation entity (Insert, Remove, Update or Rename).
- **Operation Cause**, **Old Property Path** and **New Property Path** (if applicable).
- **Old Property Value** and **New Property Value** (if applicable and necessary for further distinguishing changes).

Each AM, when generating the aggregations, can be applied with slightly different behaviors:

- By default, aggregations are collapsed with their parent if they have no siblings. This collapse can also be disabled or forced regardless of the number of siblings.
- Some aggregations are formed exclusively if their changes have a given set of characteristics - e.g. grouping by property path does not make sense in changes to the Component itself.
- Exclude aggregations that have no siblings - this is useful when the information on characteristics is only relevant for differentiating changes.

By displaying these aggregations in a tree-like manner, two main goals of the UI are achieved (see Section 6.7, p. 47):

- Collapsing repetitive changes
- Grouping changes by type of resource and type of modification

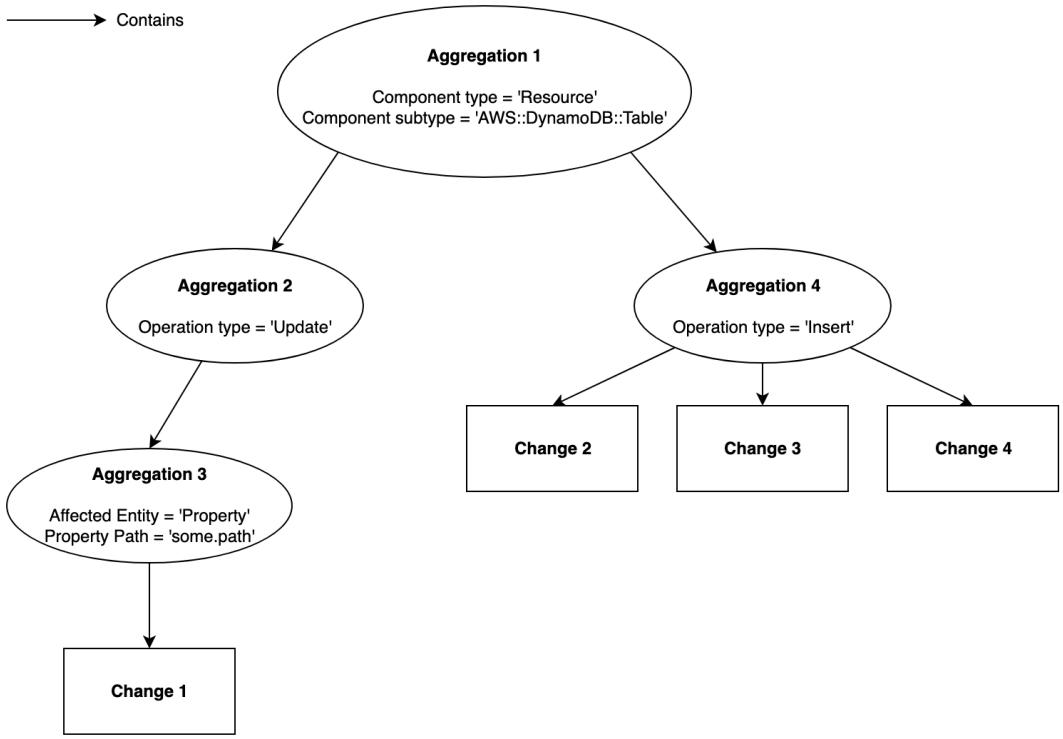

Figure 6.7: Example of aggregations for three DynamoDB Tables insertions and one property update

6.7.2 Display Component Property Operations

To allow users to view the changes in the Component Property changes (which in the case of AWS CDK/CloudFormation are the template definition changes for that Component), a similar interface to the traditional UNIX *diff* tool [31] was taken. Although existing text-diffing implementations could be used for this purpose, the operations were already being detected by C2A (see Section 6.5, p. 44) with additional semantics, such as indirect updates to components (see Section 6.5.2, p. 46). Besides this, the interface should not only display the differences, but also be able to seamlessly locate and link each displayed operation to its additional information.

Consequently, a custom implementation was developed for this purpose, divided into the following steps:

- Convert the new properties to a string. Initially attempted to do so with the existing *JSON.stringify* functionality of Javascript. However, this does not offer the necessary control over the output to add the changes with the necessary references and highlighting information. The chosen format for the output string was JSON due to the familiarity and consistency with the chosen IaC technologies.

- Modify the previously developed conversion algorithm to split the output string into sections with additional information, namely highlighting information. New property values,

whether resulting from property insertions or updates, were associated with an *insert* highlight.

- Allowing additional structures or values to be included in the final output, such as old values (from removals, updates, or renamings) with their own highlights.
- Associating metadata with the output string sections to be able to associate them with the Component Property Operations.
- Displaying each section with the appropriate highlighting color - green for insertions, red for removals, and blue for updates.

6.7.3 Changes View

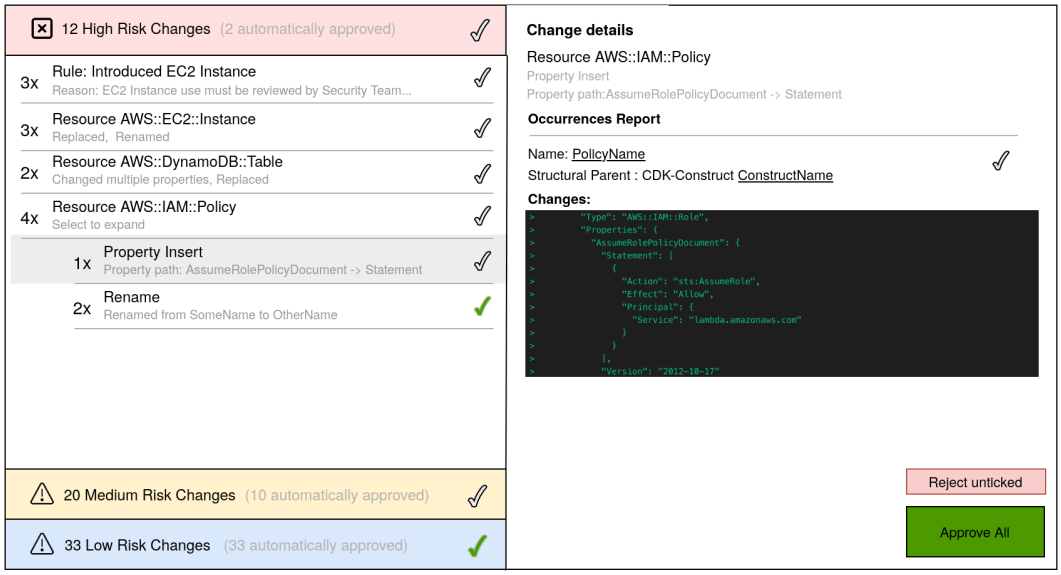

Figure 6.8: Initial Mockups for the User Interface

The first focus of the visualization was to provide easy navigation of the changes based on the generated *aggregations*, with an emphasis on highlighting the different risk levels - displayed on the left pane in Figure 6.8.

For each of these aggregations, the user is able to expand and collapse them - if they have sub-aggregations - or view their changes' details. As a first iteration, these details included only the main characteristics of the changes and a traditional diffing of the affected Component's properties (see Figure 6.9, p. 51).

In subsequent iterations of the interface, other details were introduced in the right panel, split by tabs, such as:

- **General Component Information** (see Figure 6.10, p. 52) - Provides details about the affected Component

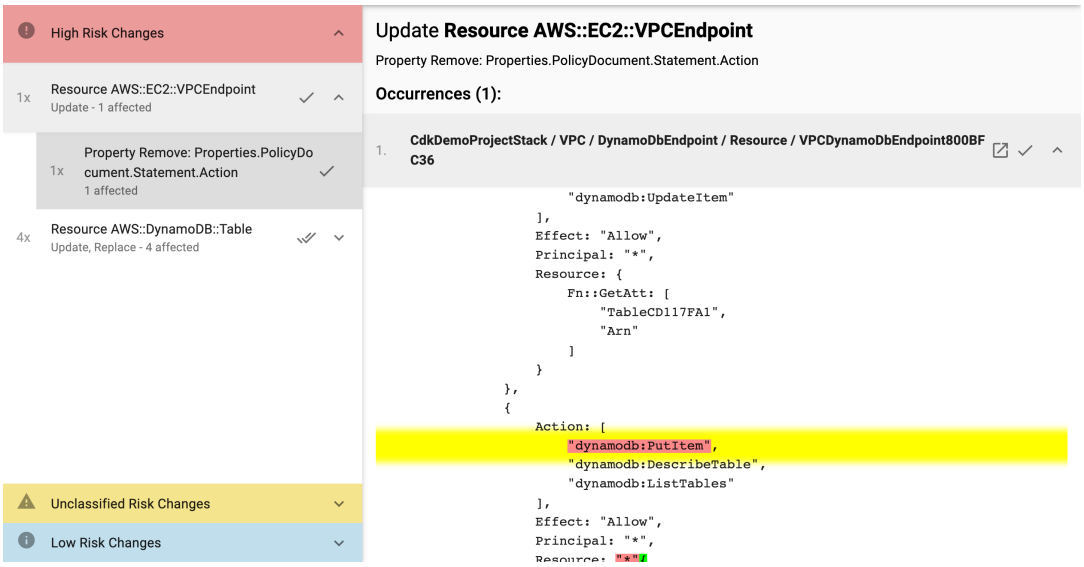

Figure 6.9: Changes View of the visualization interface - includes aggregations tree on the left pane and details on the right pane with property diffing

- **Causal chain** (see Figure 6.11, p. 53) - Indicates any inderect causes on consequences of the selected change
- **References** (see Figure 6.12, p. 54) - Enables navigation to Components that reference or are referenced by this Component - i.e., allows following incoming or outgoing Dependency Relationships.

On every displayed *aggregation* and change, there is also a checkmark icon to mark those changes as approved or not. Some of these checkmarks may be selected by default according to the pre-approval rules defined by the user (see Section 6.8, p. 52)

6.7.4 Hierarchical View

The Hierarchical View was kept with the same overall structure as the *Changes View* (see Section 6.7.3, p. 50). It is visually divided between a left pane and a right pane. Each item on the left pane is a component - expandable to show its children according to the Structural Relationships - and, when selected, has its details shown on the right pane. The details are nearly identical to the ones on the *Changes View*, with the exception of the *Causal Chain* tab being replaced with a *Changes* one. This *Changes* tab uses *Aggregations* (see Section 6.7.1, p. 48) to show the changes of a given component, but excluding characteristics like Risk or Component type and subtype (see Figure 6.13, p. 55).

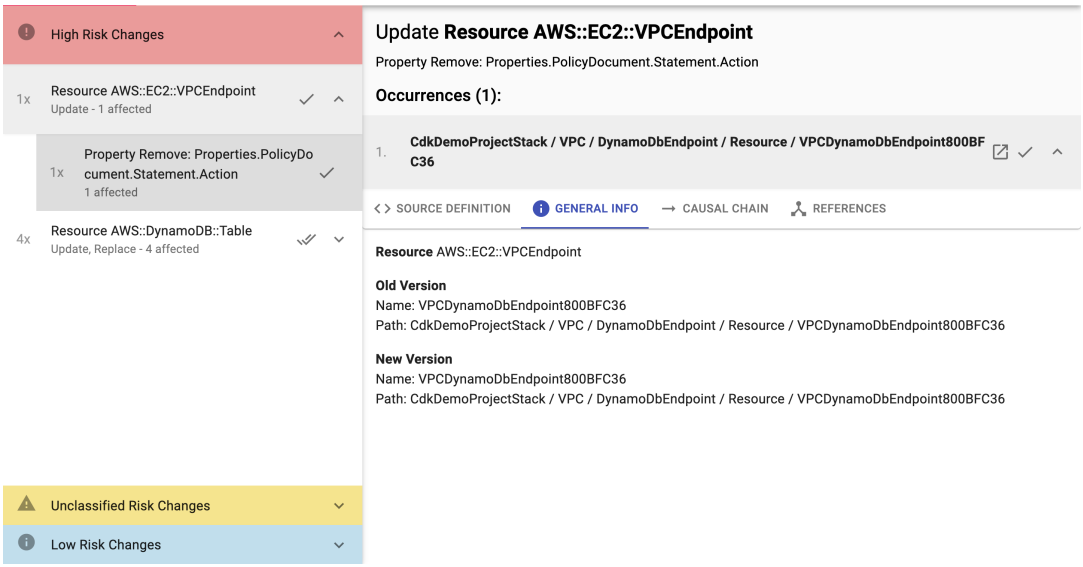

Figure 6.10: Visualization Interface - Change Details - The right pane has the 'General Component Information' tab selected, which displays a brief summary of the *Component* that suffered the selected change

6.8 Classification of changes

This tool attempts to not only display the effective infrastructure changes, but also limit user attention to the relevant ones. For this, it is crucial that users are able to customize what constitutes relevant changes. This configuration might also vary between contexts, such as the specific project or company in which the infrastructure used. Therefore, this customization should be partially or totally reusable. Base configurations could be provided by the IaC platforms and modified by developers to suit their needs.

In addition to classifying the relevance of changes, C2A also attempts to enable automated pre-approval of changes to keep developers from spending time reviewing certain changes. This would even allow skipping the manual change approval step entirely when every proposed change is deemed as pre-approved.

The rules language was designed with two main processes in mind: identifying objects and setting effects on identified changes.

6.8.1 The choice of graph-based querying

The desired target for a rule is a change or a set of changes to be classified. The user may want to identify these changes by their type, the details of the Component it affects, the properties it affects or even distant Components, their changes and how they relate to each other.

These entities relate to each other in several unique ways, which would require interpreting and processing the language distinctly for every entity and every possible type of connection between entities. To simplify this process and its implementation, we can consider all of these entities and

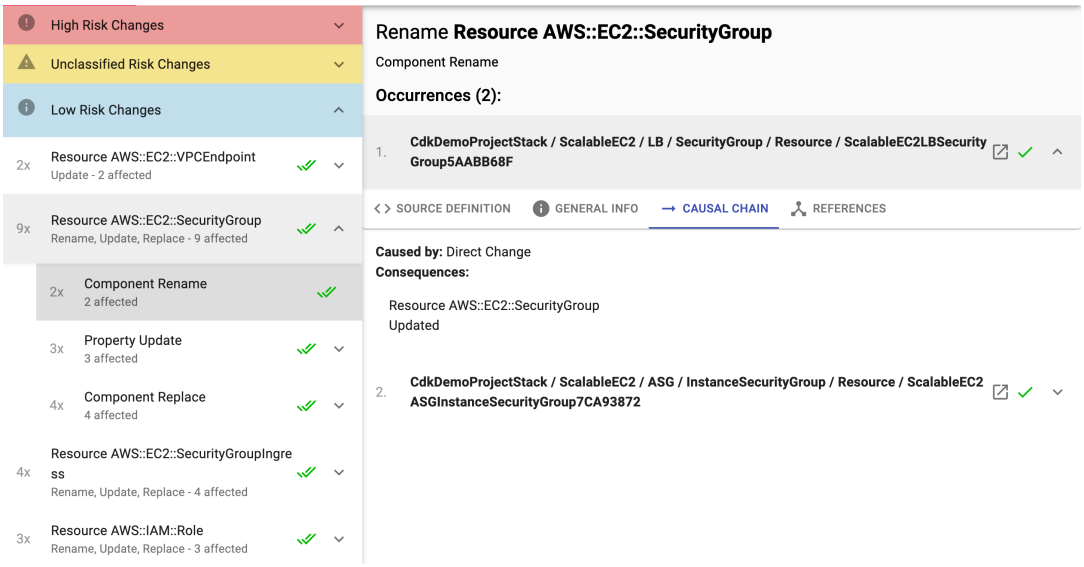

Figure 6.11: Visualization Interface - Change Details - Visualization Interface - Change Details - The right pane has the 'Causal Chain' tab selected, which displays the origin and consequences of the selected change

their connections to be part of a graph representation, where each entity is a node and they relate to each other through labeled edges. This allows the tool to query the resulting graph directly and follow edges as specified by the user, without any knowledge of the underlying model.

6.8.2 Adapting the Infrastructure Model

All relevant entities were transformed into nodes and any connection between them into labeled graph edges. This was done by creating a *Model Entity* that is extended by all of these entities and represents a node with the necessary properties and edges (see Figure 6.14, p. 56). The final graph includes the following node types:

- Components
- Relationships
- Property Values
- Operations (i.e. the detected changes)
- Transitions
- InfraModel
- InfraModelDiff - Node connected to both *InfraModels* and all Operations

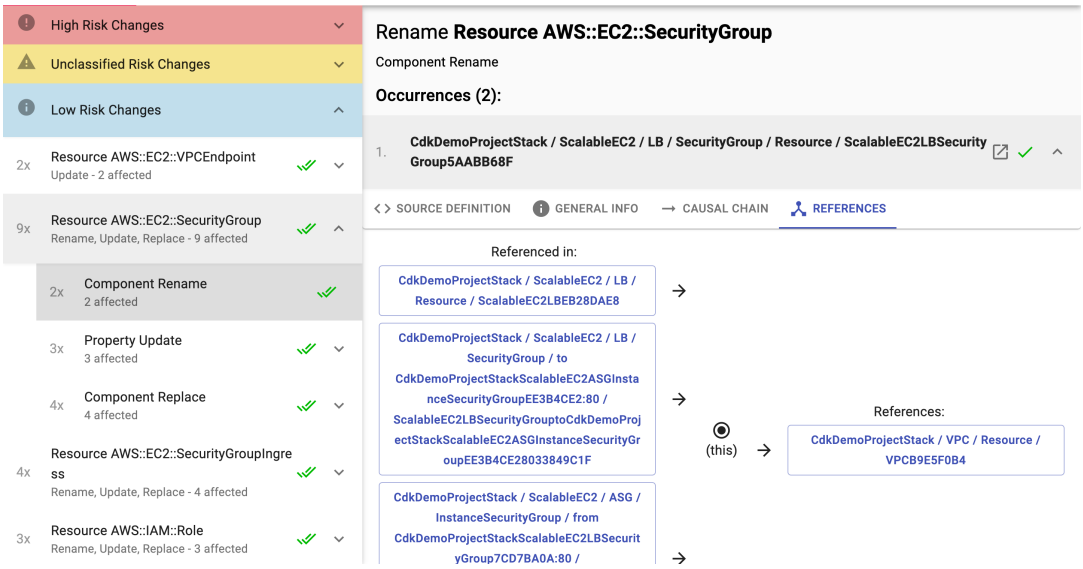

Figure 6.12: Visualization Interface - Change Details - The right pane has the 'References' tab selected, which displays the *Components* that are connected to the *Component* of the current change through *Dependency Relationships*

6.8.3 General Purpose Graph Querying Languages

Given the full graph representation of the change analysis objects, it is possible to navigate it using regular general-purpose graph querying languages such as Cypher [11]. Using such languages the user could write queries to select the targeted changes and classify them as intended. However, this approach showed issues regarding rule creation overhead and readability, particularly when navigating the graph, resulting in extremely verbose queries (e.g. navigating to a Component's property in a given path would require explicitly traversing a *hasProperty* label before every section of the path).

6.8.4 Selecting Objects

A custom syntax was designed specifically for navigating the graph, reducing verbosity:

- For selecting a change, one or more starting points are necessary. These can be any type of entity (Component, Relationship, Component Property Value, Change) and can be directly filtered by their own properties.
- Through *bindings* the user assigns a custom identifier to each selected entity, which can later be used to reference it.
- Using bound identifiers, it is possible to navigate their property edges (*hasProperty*, *value* or *exposesValues* seen in Figure 6.14 (p. 56)) using dots (".").
- Conditions allow further specification of selected entities. A condition is an expression that evaluates to a boolean value and has two operands and an operator.

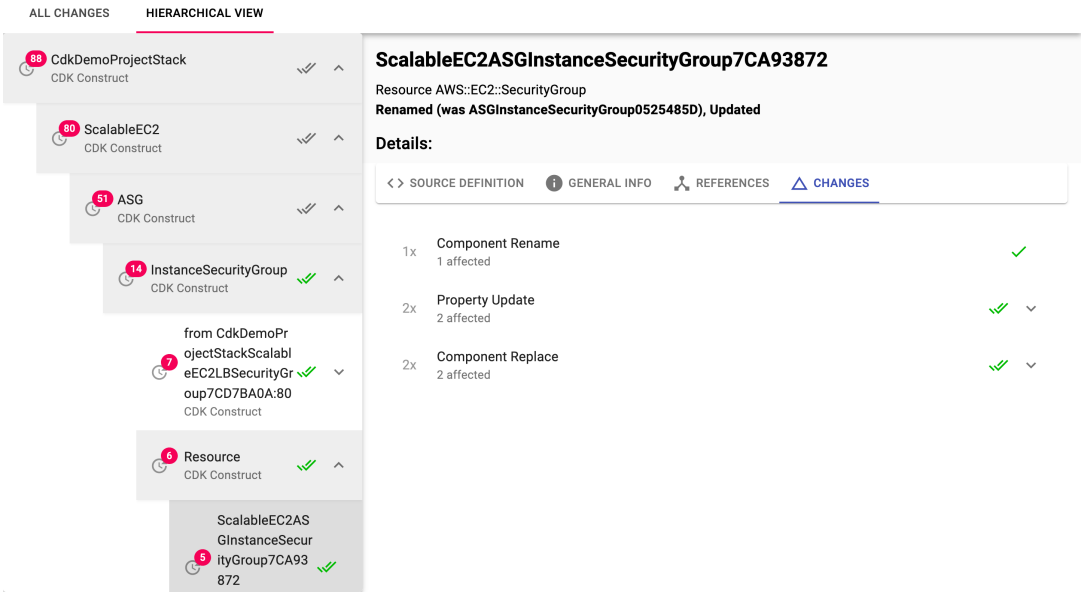

Figure 6.13: Visualization Interface - Hierarchical View - Component Details - Changes

6.8.5 Effects

Once the bindings of a rule have been defined and one of those bindings represents a change, an effect can be applied to it. The effect contains the target change and at least one consequence: the pre-approval behavior and/or the risk classification.

6.8.6 Nested Rules

Multiple rules were found to use bindings referring to the same objects. To avoid repetition of context for multiple related rules, nested rules were implemented, where inner rules have access to the bindings of their parents. Any effects applied on nested rules are applied over the effects of their parents.

6.8.7 Final Change Classification Language Grammar

A grammar definition for the final change classification language for JSON format can be found in Appendix C (p. 81).

6.9 Summary

To validate the hypothesis, a tool - the **CDK Change Analysis tool (C2A)** - was developed, which generates a visual report of changes based on two declarations of infrastructure states and optionally a custom set of change classification rules.

To achieve this, it was necessary to define a model for infrastructure representation (see Section 6.3, p. 38), map platform-specific code (AWS CloudFormation and AWS CDK) onto this

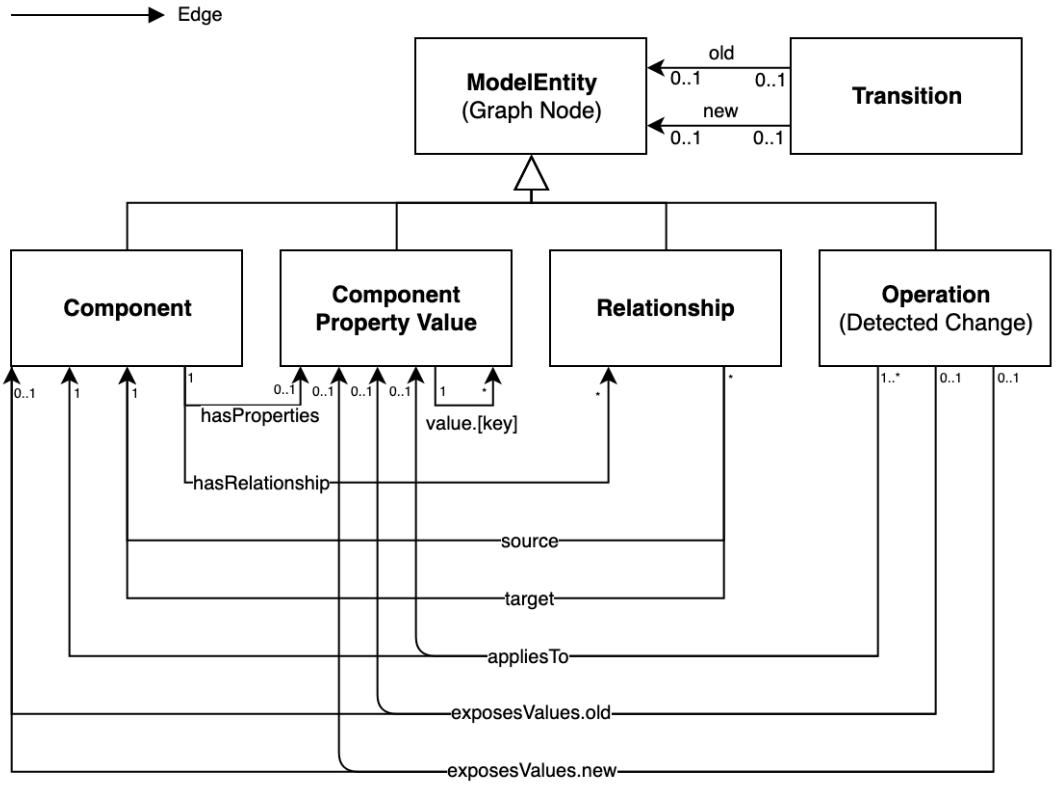

Figure 6.14: Rule-ready graph containing entities. Does not include InfraModel or InfraModelDiff

mentioned model (see Section 6.4, p. 41), perform detection of direct and indirect changes between two instances of this model (see Section 6.5, p. 44), generate a comprehensive summary of these changes — e.g. offering semantic context by grouping changes according to their characteristics and code hierarchy, grouping changes by risk, providing property comparison and other relevant information (see Figure 6.6, p. 46) — , classify changes based on user-configurable rules — e.g. assign them risk levels and mark specific ones as approved (see Section 6.8, p. 52).

Chapter 7

Validation

Due to the nature of the hypothesis of this work, its validation requires the execution of a user-based study. To this end, a survey was conducted on developers of the AWS CDK, as it is the best-supported platform of the current Proof-of-Concept tool (see Chapter 6, p. 35).

The study's structure and design are discussed in Section 7.1 and Section 7.2 (p. 62) covers the obtained results. The full structure of the survey can be found in Appendix D (p. 83).

7.1 Conducted Survey

The survey was made available through an online platform and was designed to take about 30 minutes. Its format was based on having users perform code reviews in a standard code review platform to simulate a realistic IaC development workflow. Three scenarios were created as a basis for this validation (see Section 7.1.1, p. 57): a simple *Pull Request* with few changes for benchmarking participant performance - named scenario 0 - and two other distinct ones with higher levels of complexity to measure the impact of the Proof-of-Concept in the quality of the responses - named scenarios 1 and 2.

7.1.1 Considered Scenarios

The three considered scenarios are *Pull Requests* hosted on the GitHub platform of purposely-built AWS CDK projects, covering common use cases of its usage: Building container-based web applications and building serverless web applications.

Each scenario has multiple changes of different categories:

- **Non-infrastructure related changes** - do not affect the deployed cloud infrastructure itself, but instead affect the application layer.
- **Non-critical infrastructure-related changes** - affect the deployed cloud infrastructure but are unlikely to impact functionality or present security threats

- **Critical infrastructure-related changes** - affect the deployed cloud infrastructure and are likely to impact functionality or present security threats

7.1.1.1 Scenario 0

The first scenario is used to control and benchmark the performance of each participant, while ensuring familiarity with the platform and type of project before proceeding to the experimental scenarios. It consists of a serverless web application that suffers the following small changes, in no particular order:

- Introduction of a Dead Letter Queue (DLQ) to an AWS SQS Queue
- Removal of encryption on an SQS Queue.
- Set Lambda execution timeout to 4 seconds (from the inferred default value of 3).
- Assign full IAM policy permission to all AWS S3 accessible on the deploying AWS account.
- Remove unused IAM policy permission for sending a message to SQS Queues.
- Introduction of code in the existing Lambda function to list information on all Lambda functions registered to the deploying AWS account.
- No introduction of permissions to allow the Lambda function to perform the listing operation mentioned in the previous point.
- Added project dependency to the AWS SDK.

7.1.1.2 Scenario 1

The first experimental scenario is used to compare the performance of users with and without access to the generated information. It consists of a container-based web application which has been refactored. Its main infrastructure-impacting changes are the following, in no particular order:

- Newly created abstraction for creation of Amazon VPC DynamoDB Endpoints lacks *PutItem* permission when trying to grant full access.
- Introduced a sort key in the schema definition of an existing DynamoDB table.
- As a result of the schema change mentioned in the previous point, the DynamoDB table is replaced.
- IAM policy for *ListTables* and *DescribeTable* permissions limits the accessible resources to a single table
- The IAM policy change mentioned in the previous item disables the use of the *ListTables* permission.

- Boolean parameter for allowing SSH access to the EC2 is wrongfully named and consequently invoked with an inverted value.

7.1.1.3 Scenario 2

The first experimental scenario is used to compare the performance of users with and without access to the generated information. It consists of a more complex serverless web with a small number of changes. Its main infrastructure-impacting changes are the following, in no particular order:

- New version of updated construct library contains dangerous CloudFormation Origin change to URL.
- A new function is invoked in the configured Step Functions, which intends to send push notifications.
- A renamed DynamoDB Table gets replaced as a consequence.
- IAM policy for the Lambda function that operates over the users DynamoDB table can now only perform operations on that table.

7.1.1.4 Notes on the Designed Scenarios

Some of these scenarios may have other inherent flaws in their functionality, such as misconfigurations or small implementation bugs. However, the purpose of this research is to study the quality of review of code changes and, as such, potentially existing flaws in the system are not taken into account.

7.1.2 Participant Criteria

The targeted respondent demographic was defined as developers with some level of experience with the AWS CDK using Typescript, since the scenarios were designed in this language. The participant is not expected to have any given level of experience or organizational context, as the hypothesis should apply to all users. It is also expected for the developer to be familiar with code reviews, preferably with the GitHub code review interface.

7.1.3 Survey Structure

In order to guarantee the relevance of the results, each experimental scenario (scenario 1 and 2) must be subject to a control group - a set of participants who do not receive the treatment (the proof-of-concept tool's output) - and an experimental group - who do get access to the additional information. Several approaches were considered when establishing these groups, as described below.

7.1.3.1 Considered Solutions

The first considered approach consisted of dividing the entire study into two groups and requesting code reviews of the scenarios in the following way:

- The control group would have to review all scenarios in a fixed order without access to the additional information.
- The experimental group would have to review the same scenarios, in the same order. However, in the second scenario, the change analysis interface would be provided without classification of changes, and in the third scenario, the fully categorized output would be made available.
- The scenarios would be ordered in increasing complexity.
- To evaluate the accuracy of the reviews, a checklist would be shown with both accurate and inaccurate changes for participants to select those that they were able to detect.

However, this approach presented threats to validity and execution:

- Due to the niche nature of the target group, raising candidates for the experiment was expected to be a difficult task - which was later confirmed. Having the control group review three scenarios without experiencing the hypothesis in question was feared to cause users to quit the survey without filling it entirely, consequently risking the group's statistical significance.
- Simultaneously increasing complexity and providing more detail in the additional information could hamper determination of cause behind participant performance.
- The validity of the full hypothesis would only be analyzed in one scenario.
- A checklist of changes would introduce a considerable bias, as participants would know what changes to look for when reviewing.

To tackle these problems, multiple solutions were considered:

- Providing the change report in both groups and reducing the scope of control groups to a per-scenario basis, instead of survey-wide. With this solution, both participant groups could have the additional information on one of the scenarios. The order of the experimental scenarios would be switched to ensure neither participant group would have that information in the first scenario and avoid introducing bias in the following scenario (i.e. participant group A would review scenario 1 without the information and then scenario 2 with it; participant group B would review scenario 2 without the information and then scenario 1 with it).
- Reducing the number of scenarios to two, while increasing the number of groups. Keeping one control group and having two experimental groups - one accessing the change report

with the customized rules and one without it. This solution still was greatly dependent on having sufficient survey adoption, besides limiting the experimental portion to just a single scenario.

7.1.3.2 Adopted Solution

The adopted solution resulted from a combination of these two proposed solutions. The survey is then structured in the following manner:

- Keeping the three scenarios - one scenario for benchmarking participant performance (scenario 0) and two experimental ones (scenario 1)
- Having two participant groups with access to the three scenarios.
- Requesting purely code review on scenario 0 in both participant groups, on scenario 1 in group A and on scenario 2 in group B
- Requesting code review with the additional information on scenario 2 in group A and on scenario 1 in group B.
- Keeping the order of scenarios as 0, 1, and 2 across participant groups.
- Having developers describe the reviewed changes in an open response text box.
- Introducing a third participant group if the original participant groups reach a satisfying number of responses - 20.

The final contents of the survey followed this solution and are available in Appendix D (p. 83). This solution, however, remains with significant threats to validity, presented in Section 7.1.4.

7.1.4 Threats to Validity

The adopted validation approach includes the following significant threats to validity:

- Possible introduction of bias in scenario 2 - its control participants will be aware of some of the additional information to detect as they have been shown a similar one previously. Nonetheless, while this bias only affects one scenario, the alternative of reviewing scenarios in different orders would introduce bias in both, which would be harder to measure.
- Inability to accurately measure review duration. Heightened by participants not being accustomed to the user interface of the C2A tool, which leads them to spend more time in this learning process.
- Having unclear or ambiguous descriptions of changes in the written reviews.

7.2 Findings

The following subsections analyze the results of the conducted study. These results are based on a total of 21 participants with 10 participants in group A and 11 participants from group B.

7.2.1 Participant Population Analysis

The survey was filled by 155 individuals. However, in the large majority of responses participants quit the survey without completing it fully. A large number of participants abandoned the survey when asked to review a Pull Request, in either scenario. Group A had significantly less valid responses and because of this, users from that group who fully completed at least scenario 1's questions as requested were also considered. This allowed having a balanced number of considered participants in each group, creating a total of 21 considered participants. Unavoidably, scenario 2 holds fewer responses (16) and has, therefore, less statistical significance. In addition, some outliers were removed, who did not follow or may have completely misunderstood the instructions.

All participants reported having at least basic experience in cloud services, IaC tools, AWS Services and AWS CDK. Figure 7.1 reveals the distribution of experience levels across all participants and technologies.

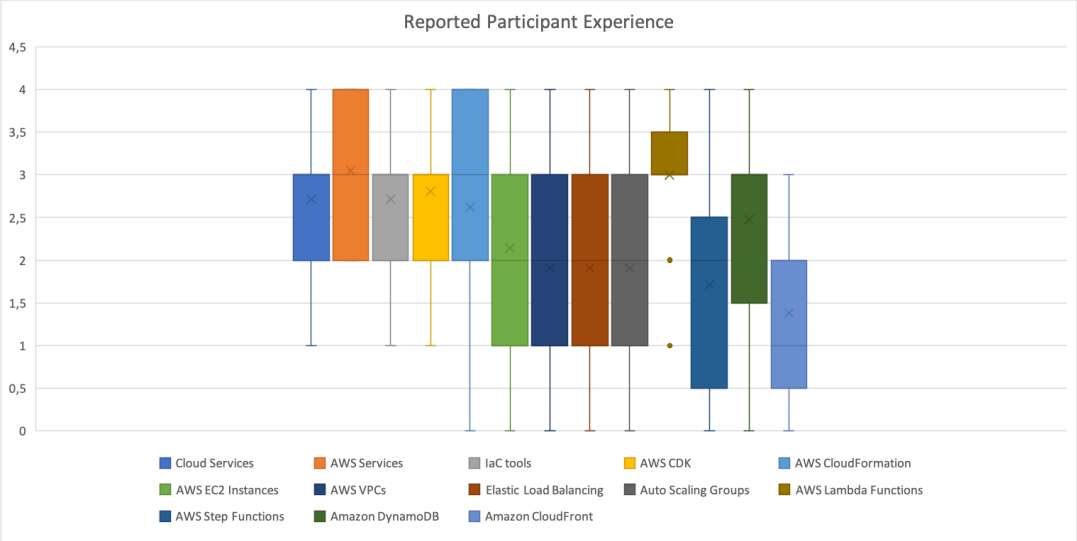

Figure 7.1: Reported participant experience, on the following scale: 0 - Not experienced; 1 - Somewhat experienced; 2 - Fairly experienced; 3 - Very experienced; 4 - Extremely experienced

7.2.2 Benchmarking Participant Performance

The changes from scenario 0 were given weights according to their categories (see Section 7.1.1, p. 57). Non-infrastructure changes were given a weight of 1, non-critical infrastructure changes were given a weight of 2 and critical infrastructure changes were given a weight of 3.

The detected changes of Section 7.1.1.1 (p. 58) were extracted from each answer and assigned a value of 0 or 1 that were either not detected or detected, respectively. Answers with implicit mentions of each change were given a value of 0,5. By performing a weighted sum - multiplying these values with their corresponding change and summing them for a given participant -, we can assign them a performance score that describes how complete and descriptive their answers were. After normalizing these scores in relation to their theoretical maximum value, we obtain an average score of 60% and a standard deviation of 17%.

Although the scores are lower than expected, this can be explained by the recommended 2 minute review time which limits the number of changes one can review and inherently affects their overall score. Another factor that may have contributed to these values is the average experience of 69% for the technologies used in this scenario (with a 16% standard deviation).

7.2.3 Accurately Detected Changes

To the answers from scenarios 1 and 2, we applied the performance analysis method in Section 7.2.2 (p. 62), with a slightly different weight system. Since the participants were not asked to identify changes other than security threats or functionality breaking ones, these cannot count towards how accurate their reviews were. The applied weights were: 1 for security-related changes and 2 for immediate security threats and functionality-breaking changes.

In scenario 1, we can observe an average score of 25,6% when the proof-of-concept tool is not provided and an average score of about 27,3% when the participant does have access to it. The overall scores in this scenario appear to be even lower than those obtained in scenario 0, which can be explained by an even lower average experience score for its technologies (62%) and a 10 minute recommended time for analyzing a fairly complex refactoring with multiple hidden changes. Based on this data we can estimate that having this extra information leads to a 6,5% increase in performance.

Scenario 2, instead of featuring code refactoring, focuses on small changes and hiding one inside external dependencies. This shift of context leads to more optimal performance results, with a 50,0% improvement on the average score when using C2A. The lower response volume for this scenario, however, decreases the statistical significance of this obtained value.

When assigning a weight to each participant's score proportionate to their performance in scenario 0 - emphasizing the performance of more committed reviewers, we can observe a 10,0% and 55,8% increase in performance with C2A in scenarios 1 and 2 respectively.

The overall increases in performance of 6-56% are lower than expected, which can be explained by the large portion of the sample who failed to detect any of the relevant changes - 33% in scenario 1 and 25% in scenario 2 -. These participants either identified flaws in the existing system instead of the presented changes, or failed to find any security or functionality concerns at all.

The distribution of the unweighted performance scores can be found in Figures 7.2 and 7.3. In both cases, having the proof-of-concept tool reveals a more condensed score distribution, meaning that having the tool adds consistency among reviews. It also slightly increases the average obtained

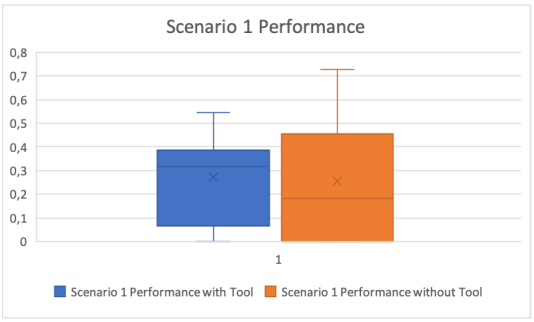

Figure 7.2: Box Plot of the performance scores obtained in scenario 1

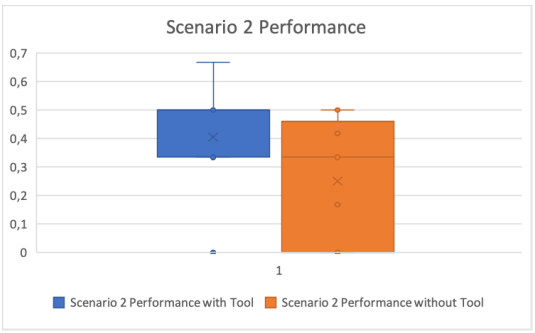

Figure 7.3: Box Plot of the performance scores obtained in scenario 2

score. It can also be observed that the best scores were obtained without the use of the tool, which can be explained by certain participants having spent much more time than recommended in the review process, as mentioned in Section 7.2.5 (p. 65).

When participants with access to the tool were asked what changes they would not be able to identify without the tool, at least one change was reported by 90% in scenario 1 and 36% in scenario 2. This confirms that the two scenarios were very different in complexity and that participants found the tool to be helpful to some degree.

One particular change - Limiting IAM Policy permission resources of *DescribeTable* and *ListTables* actions to a single table disables the usage of *ListTables* (see Section 7.1.1.3, p. 59) - was only detected by one participant, who did not have access to C2A's output. This change required very specific knowledge to be detected and was not included in the change classification configuration and, therefore, not highlighted in the corresponding change summary.

7.2.4 Review Outcome Confidence

In both scenarios 1 and 2, the average participant confidence in having detected all security-threatening changes suffered little impact by providing the C2A tool (having increased just 2,7% and 0,8% respectively, while the standard deviation increased 4,8% and 32,1% respectively). On the other hand, functionality-impacting changes saw an 18,5% increase in confidence in scenario 1 and a 4,8% decrease in scenario 2 (with 0,5% and 29,4% increases in standard deviation). Figures 7.4 and 7.5 further reveal this distribution in confidence across the different contexts.

Despite the mentioned effects on the average, the median remained unchanged across contexts. Figure 7.4 (p. 65) reveals a more condensed distribution of the reported confidence when using the tool, both in security threats and functionality breaking changes. By contrast, Figure 7.5 (p. 65) revels the opposite consequence of making the tool available. This can be due to multiple factors, including slightly less familiarity with its technologies (see Figure 7.1, p. 62), the refactoring nature of scenario 1, the difficulty in checking possibly hidden changes in external packages, a natural decrease in performance throughout the survey due to exhaustion or attention span.

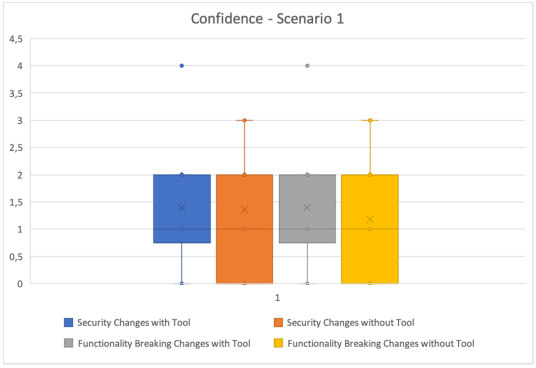

Figure 7.4: Box Plot of the reported confidence in scenario 1 - With and without access to the tool, for changes representing security threats and functionality impact

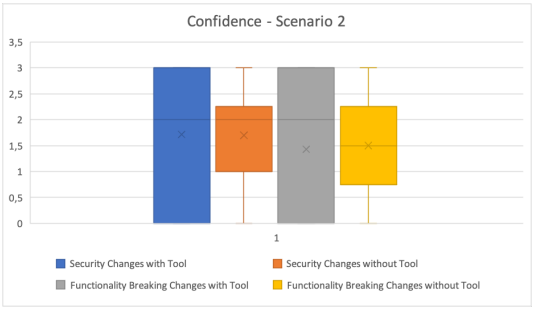

Figure 7.5: Box Plot of the reported confidence in scenario 2 - With and without access to the tool, for changes representing security threats and functionality impact

7.2.5 Review Duration

The format of this survey has an inherent flaw in measuring time spent using the tool. Although developers are traditionally accustomed to using code review interfaces, the C2A tool's interface requires a learning process which participants had not been subject to prior. For this reason, the review duration was expected to increase slightly when having the tool available.

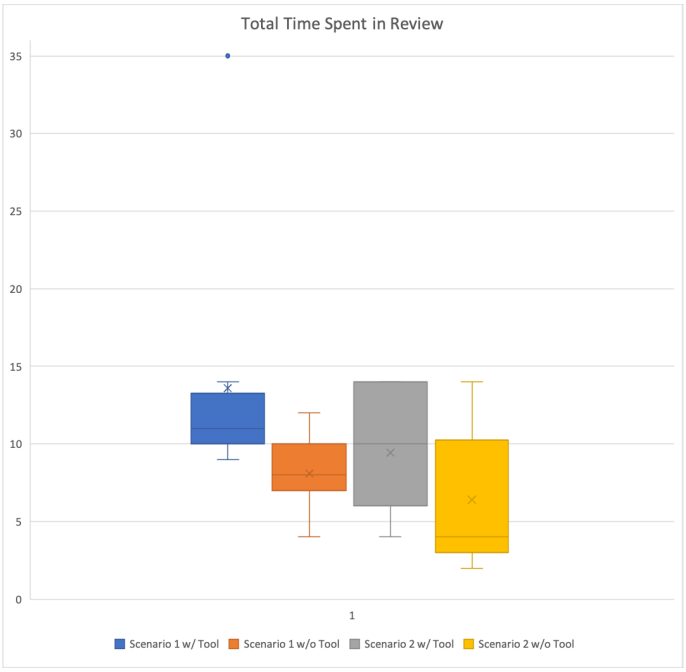

Figure 7.6: Total Reported Time Spent in Review (in minutes), for both groups and scenarios 1 and 2

Figure 7.6 contains the distribution of the reported total duration of the review process. As expected, having access to the tool with no prior experience leads to a greater review time in both

scenarios. The majority of participants with access to C2A's output in each scenario exceeded the recommended time, having one outlier reported spending 350%.

Since the survey recommended a 10-minute review for each scenario, participants may also have stopped the review before being fully satisfied with their conclusions. Hence, it is more useful to compare the time spent to achieve similar performance scores for participants of similar experience backgrounds and benchmarked scores. The currently limited volume of obtained responses does not allow us to ascertain such a relationship with statistical significance.

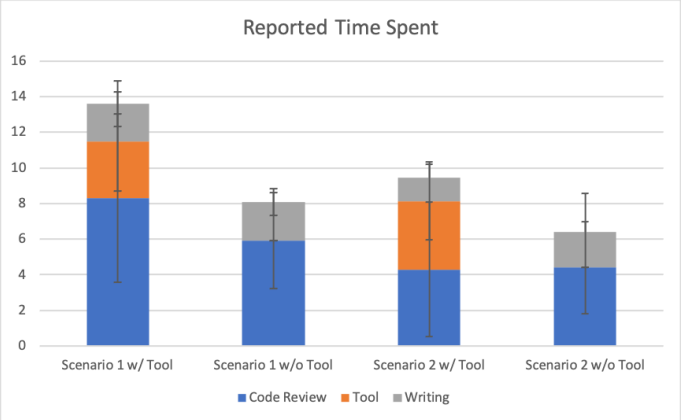

Figure 7.7: Reported time spent across scenarios 1 and 2 in the multiple processes of review

7.2.6 Reported Experience

An experience score was produced for each participant in each scenario by averaging the reported experience values of the technologies relevant for that given scenario:

- **Scenario 0** - Cloud Services, AWS Services, IaC tools, AWS CDK, AWS CloudFormation, AWS EC2 Instances, AWS VPCs, Elastic Load Balancing, Auto Scaling Groups, AWS Lambda Functions, AWS Step Functions, and Amazon DynamoDB.
- **Scenario 1** - Cloud Services, AWS Services, IaC tools, AWS CDK, AWS CloudFormation, AWS EC2 Instances, AWS VPCs, Elastic Load Balancing, Auto Scaling Groups, AWS Lambda Functions, AWS Step Functions, and Amazon DynamoDB.
- **Scenario 2** - Cloud Services, AWS Services, IaC tools, AWS CDK, AWS CloudFormation, AWS EC2 Instances, AWS VPCs, Elastic Load Balancing, Auto Scaling Groups, AWS Lambda Functions, AWS Step Functions, Amazon DynamoDB, and Amazon CloudFront.

Using these values and crossing them with the obtained performance values throughout the three scenarios, we conclude there is no clear correlation between the reported experience for any given scenario and a participant's performance, as demonstrated in figures 7.8, 7.9 and 7.10. In figure 7.10 there is arguably a trend for better performance with increased experience when using the C2A tool's output, but due to the lack of data points, it remains inconclusive.

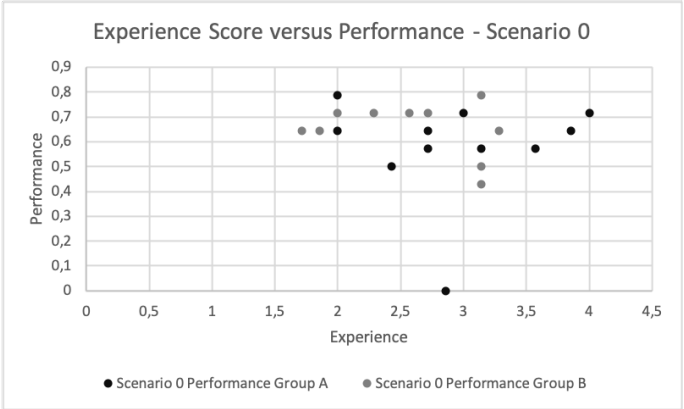

Figure 7.8: Scatter plot of reported experience against obtained performance scores - scenario 0

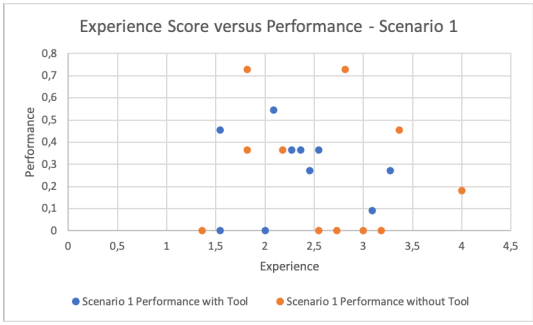

Figure 7.9: Scatter plot of reported experience against obtained performance scores - scenario 1

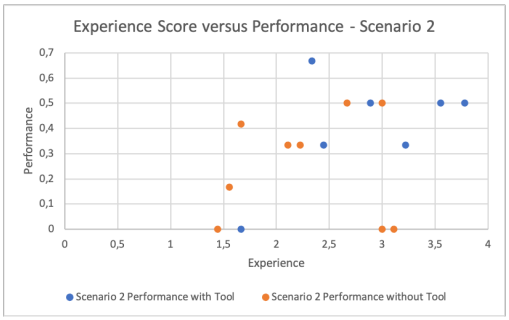

Figure 7.10: Scatter plot of reported experience against obtained performance scores - scenario 2

7.3 Observed Threats to Validity

The obtained results, as expected (see Section 7.1.4, p. 61), cannot fully validate all hypothesized effects. Based on the submitted responses, the following validity defects were observed:

- A relatively low number of valid responses was collected - 21 - decreasing the statistical significance of the drawn conclusion. This is especially relevant on the results of scenario 2, which were based on a limited number of 16 responses.
- Several people failed to identify any of the relevant changes in scenarios 1 (33%) and 2 (25%). These cases don't seem to correlate with the amount of time people dedicate to the review or whether they had access to the tool. It is, hence, impossible to extract relative improvements of using the tool based on them.
- The bias introduced in scenario 2 that was expected to decrease the performance gains of users with access to C2A's output was not significant. The tool was found to have provided

a positive improvement to scenario 2's scores, even surpassing the improvement obtained in scenario 1.

- The total time spent increased as users had access to the tool, which was expected. However, the amount of time users spent on the code review itself increased in scenario 1 when having access to the tool. It is, therefore, unclear how much the observed increase in performance was due to the tool versus the extra amount of time spent in code review.
- Multiple participants attempted to expose flaws in the codebase that were not part of the set of changes introduced, decreasing the overall time dedicated to the changes and, consequently, the number of relevant detected changes. This lowers the amount of data points on a per-change basis, further decreasing the ability to take on their type.

Chapter 8

Conclusions and Future Work

This dissertation attempted to study a solution for safely introducing infrastructure changes in an Infrastructure-as-Code (IaC) codebase, particularly for substantially large ones, such as when including external libraries or updating already included ones. It is currently very hard to review and analyze the security risk of such changes efficiently, as reviews become very time-consuming and may be unable to detect subtle vulnerabilities in resources, configurations, or permissions.

8.1 Answering Research Questions

8.1.1 ERQ1. Can code flaws introduced in IaC code changes be avoided by performing peer-review?

Analysis of the results suggests that code flaws introduced in IaC code changes can be avoided by performing peer-review, but not completely (ERQ1. in Section 5.3 (p. 31)). In none of the three scenarios presented to industry professionals in this study, was there a total identification of all security threats or functionality breaking changes by any individual. This remains true when using the proposed solution.

8.1.2 ERQ2. How can the peer-review process in IaC be facilitated to minimize undetected code flaws?

8.1.2.1 ERQ2.1. What effect do the summary and automated classification of changes during code review have on the percentage of accurately detected changes? What effect does it have on both security-threatening and functionality-breaking changes?

The gathered results also show a slight overall improvement in the number of accurately detected security threatening and functionality breaking changes when developers are given access to the changes in a summarized, structured and classified manner (see ERQ2.1. in Section 5.3 (p. 31)).

This is most observable when changes: are hidden in dependency updates (only detected by participants with access to the proposed solution), implicitly result in stateful resource replacements (49,1% more likely to be detected), or consist of bugs or small changes introduced when creating abstractions (on average twice as likely to be detected - 201,2% averaged across 4 different changes).

8.1.2.2 ERQ2.2. What effect do the summary and automated classification of changes have on how confident developers are in detecting all impactful changes? What effect does it have on both security-threatening and functionality-breaking changes?

Despite the positive results regarding the quality of the reviews, the proposed solution did not significantly impact the average developer confidence in the review outcome (see ERQ2.2. in Section 5.3 (p. 31)). However, it did impact the distribution of such reported confidence values. This effect varied according to the presented set of changes, either converging or diverging the reported confidence scores, but remained consistent across security-related and functionality-breaking changes.

8.1.2.3 ERQ2.3. What effect does the extra information provided by the summary and classification of changes have on how much time the code review takes? Does having changes pre-approved and risk-sorted decrease the amount of time spent? Does it decrease the amount of effort allocated to non-impactful changes?

The proposed solution also created an average 57% increase in the amount of time spent reviewing the set of changes. Although this can be partially explained by the fact that users had no previous experience with the interface, the average reported time spent in traditional code review also increased, possibly indicating skepticism or confusion introduced by the presented solution (see ERQ2.3. in Section 5.3 (p. 31)).

Time spent on non-impactful information decreased 15% on average in scenario 1 (where C2A showed the most benefit), but increased 14% in scenario 2. There is, therefore, no sufficient data to determine this effect, but indicates that the C2A tool may be better suited for evaluating specific types of scenarios.

One particular change was only identified by one participant, who did not have access to C2A's output, due to requiring very specific knowledge to be detected (see Section 7.2.3, p. 63). Although its detection was not included in the change classification configuration used to produce the change summary, this situation revealed a case where such configuration could be easily added and highlighted in future deployments.

The results obtained presented lower increases in the number of accurately detected relevant changes than expected, which is assumed to be due to the relatively small amount of optimizations regarding user experience and navigation in the produced change summary (see Section 8.4, p. 72).

8.1.3 ERQ3. Is it possible to automate the change review process in IaC? Can it be fully automated in a CI/CD pipeline?

The developed reference implementation lacks the maturity to determine whether the change review process in IaC can be fully automated in a CI/CD pipeline (see ERQ3. in Section 5.3 (p. 31)). The developed change classification language should allow change review automation (see Section 6.8, p. 52), but its applicability to real-world use is yet to be evaluated, making this a good path to pursue as future work (see Section 8.4, p. 72).

8.2 Hypothesis

We attempted to validate the following hypothesis:

> *The productivity and security of declarative IaC code deployments can be increased by: a) automating the approval or rejection of specific infrastructure changes in a continuous deployment pipeline; b) generating a comprehensive summary of upcoming infrastructure modifications, suitable for manual review of large changes. These methods provide the highest impact when changes are hard to detect in regular code reviews or introduced in third-party code.*

Automated creation of an interactive summary of effective infrastructure changes was proposed, featuring user-configurable highlighting of impactful modifications, condensing of repetitive information, and pre-approval of modifications that do not require human review.

The reference implementation did not allow to confirm point *a)* of the hypothesis or conclude productivity impacts. It did, however, confirm point b) regarding security, as generating a comprehensive summary of upcoming infrastructure modifications, suitable for manual review of large changes, did have a positive effect on detecting impact on security functionality in declarative IaC code deployments.

8.3 Process Overview

Designing the proposed solution involved extensive literature research, performed in the relevant fields of continuous deployment of IaC code, analysis of infrastructure changes, visualization of changes for review and usage of custom rules in continuous deployment. Furthermore, existing IaC tools and products were studied and compared, depicting the current approaches for reviewing and ensuring safe deployments of large sets of changes.

Complementing the current state of the art, interviews were conducted with industry professionals of different backgrounds. This provided information about the adopted practices in IaC continuous deployment workflows, as well as allowed gathering feedback on the proposed solution to estimate its feasibility and applicability (see Chapter 4, p. 21).

To tackle this barrier in the wide adoption of IaC code sharing, we proposed a dedicated visualization of changes, optimized to highlight unwanted changes and collapse less relevant ones.

In addition, a user-defined set of rules was proposed to automatically skip the manual review of specific changes - when integrated in a continuous deployment pipeline -, allowing for faster human infrastructure deployment approval and safe automatable deployments.

For the validation of the mentioned proposal, a reference implementation was developed, which consisted of a proof-of-concept tool that allows both human review of large infrastructure changes, as well as their classification and approval based on user-defined rules.

8.4 Future Work

Although this study has given useful insights into the impact of change analysis and its automation for declarative Infrastructure-as-Code development workflows, it remains a largely unexplored field, leaving room for future studies.

The developed C2A tool can be further optimized for large adoption by implementing the following improvements:

- Improving effectiveness and intuitiveness of the user interface by gathering further developer feedback on existing flaws and potential improvements of the displayed information .
- Adding the distinction between *sets* and *arrays* of property values in Components since, contrary to set reordering, array reordering may impact the resulting infrastructure.
- Providing the change analysis report under a Command Line Interface (CLI) for faster feedback on local development environments.
- Further increasing user-friendly syntax on the change classification rules language.
- Increasing support for AWS CDK and AWS CloudFormation features including, but not limited to: resolving nested stack templates (see Section 2.4.2.2, p. 11); supporting YAML templates (see Section 2.4.2.3, p. 12); interpreting intrinsic function expressions (see Section 2.4.2.1, p. 11); using AWS CDK code information not currently included in the synthesized CloudFormation template (e.g. asset changes, construct file locations) and checking configuration drift of the target deployment environment.
- Adding support for other declarative IaC platforms.

Regarding the direct influence of this approach on the performance and safety of development workflows, the following aspects should also be studied:

- Integrating approval with CI/CD pipelines and evaluate the applicability of automating the change review process
- Studying the suitability and experience of defining change classification rules in real-world development workflows.

- Using historical change approval data to classify future changes automatically.
- Adding mechanisms for implementing new rules based on the produced change analysis reports, possibly providing a rule generator based on the characteristics of a given displayed change.
- Comparing applicability and impact of using this tool on the three idealized stages of development (local, code-review, and pre-deployment) (see Section 6.6, p. 46)
- Providing textual descriptions of the rules applied to offer context on displayed changes
- Matching Components based on the entire incoming *InfraModel* subgraph (the subgraph obtained by following all incoming relationships), either based on dependency relationships or structural ones. This could potentially be done by comparing Components while performing a topological traversal of both *InfraModel* versions.
- Grouping and displaying similar sets of related changes by performing subgraph clone detection after change detection.

Appendix A

Keywords Used in Literature Research

This appendix contains the keywords ultimately used in the literature research and review, as depicted in Section 3.1 (p. 15).

- **Semantic Analysis of Infrastructure Changes** - Infrastructure-as-Code; Cloud Infrastructure; Semantics; Code Review; Analysis; Configuration Management; Code Changes; Version; Code Differences; Impact Analysis.
- **Infrastructure Representation** - Cloud Infrastructure; Representation; Modeling; Architecture; Specification; Definition; DevOps; Standard; Metamodel; Automation.
- **Identification of Changes** - Code Changes; Code Differences; Code Diffing; Graph Diffing; Graph Changes; Graph Matching; Cloud Infrastructure; Infrastructure-as-Code; Clone Detection; Graph Comparison.
- **Visualization of Changes for Review** - Code Changes; Visualization; User Interface; Code Review; Semantics; Display; Optimization; Analysis; Code Diffing; Code Comparison; Code Differences.
- **Custom Rules in Continuous Deployment** - Configuration; Continuous Deployment; CI/CD; Domain Specific Language; Validation; Rules Language; DevOps; Policy-as-Code.
- **Continuous Deployment in Infrastructure-as-Code** - Configuration; Continuous Deployment; CI/CD; Validation; DevOps; Infrastructure-as-Code; Cloud; Infrastructure; Provisioning; Orchestration; Management.
- **Relevant Features of Current IaC Tools** - Infrastructure-as-Code; Tools; DevOps; Compliance; Changes; Automation; Diff; Impact; Analysis; Check; Policy-as-Code; Pulumi; AWS CDK; AWS CloudFormation; Ansible; Chef; Terraform; Puppet.

Appendix B

Preliminary Studies - Interview Questions

B.1 Current Experience

The following are the interview questions and justifications for gathering information on the interviewee's experience.

1. **Give a short overview of your business and product.**

   Infrastructure requirements and approaches vary across different businesses and products. While some require frequent deployments, others may accumulate several changes onto a single deployment. Security requirements may also vary across this spectrum, along with automated deployment processes and strategies. In short, this information allows us to match specific expectations and requirements to their business contexts and assess whether the needs collected in the following questions are generalized or context-specific.

2. **Do you use cloud services? Which ones? For how long?**

   IaC helps manage and deploy cloud resources. If the interviewee does not have experience with it, IaC related questions will not be relevant. Different providers may also result in different experiences with IaC, which may be reflected in the following questions' answers.

3. **Could you draw and describe the complexity of the infrastructure maintained by your product?**

   Higher complexity and diverse infrastructure approaches provide insight into a wide range of use cases. Asserting these use cases enables a better estimation and prioritization of the critical aspects to consider for the proposed solution.

4. **Give a short overview of your experience regarding IaC. Does your company use IaC? What IaC technology do you currently use? For how long have you used it? Have you or your company used other IaC solutions before? Which ones? What made you change?**

   Different experiences and levels of expertise may have a wide range of use cases relevant for this study. Asking the interviewees about their experience will allow us to evaluate the proposed solution's effectiveness within their work.

5. **Can you describe your infrastructure deployment process? Are your deployments automatic? Do you use CI/CD pipelines? How frequent are your deployments?**

   The review of infrastructure changes and their approval is often a part of CI/CD tasks. Since we intend to simplify and automate certain aspects of this process, it is essential to determine how it is currently handled in order to design a solution that can improve upon it consistently while fitting the strategies currently adopted by the interviewees.

6. **Have you previously used third-party IaC code in your project? If so, how was its risk assessed?**

   Third-party IaC code may introduce or modify resources in ways that may not be clear to the user. Understanding how the interviewees currently assess the risk in such situations proves crucial to designing solutions for automatically detecting and processing potential security risks or unintended side effects.

7. **How much time do you usually spend analyzing the infrastructure changes before approving each new deployment? How much of this time is spent on interpreting the changes or repetitive information? How is the approval responsibility managed? Who is responsible for approving the deployment? How is it handled when the responsible person is not available?**

   In order to successfully simplify the user's experience and speed up the analysis of the changes, it is essential to determine what aspects decrease productivity and to what extent. Understanding this allows for evaluating and establishing strategies that could be used to tackle such problems from a user experience standpoint.

8. **Are there any frequent changes in your project whose approval could be automated?**

   This question attempts to assess the applicability of rule-based automatic approval or refusal behaviors as a replacement for manual approval of specific changes in order to establish its relevance for a cleaner and faster deployment workflow.

9. **Is your infrastructure deployed in multiple stages, regions, or clouds? If so, would you benefit from automatic approval of infrastructure changes that have already been approved for other environments? In which contexts (stages, regions, clouds)?**

   Through these questions, we attempt to understand how many interviewees would benefit from automatic history-based approvals and by how much, specifically in the case of multiple stages, regions, or clouds, as it is the most intuitive application of this feature.

B.2 Approach Discussion

The following are the interview questions and justifications for gathering feedback from the interviewee about the proposed solution.

1. **Of the planned features, which would have the most impact on your current IaC workflow?**

   The following 3 questions allow us to determine the relative relevance of each feature in order to better plan the implementation of the proposed solution.

2. **On the following scale, how beneficial would collapsing repetitive information on the diff tool be for your current IaC workflow?**

   Not beneficial - slightly beneficial - moderately beneficial - greatly beneficial

3. **On the following scale, how beneficial would automating specific change approvals be for your current IaC workflow?**

   Not beneficial - slightly beneficial - moderately beneficial - greatly beneficial

4. **On the following scale, how beneficial would automatic approval of changes based on past approvals be for your current IaC workflow?**

   Not beneficial - slightly beneficial - moderately beneficial - greatly beneficial

5. **Can you give examples of rules that would be particularly relevant for your project?**

   Understanding which types of rules would be particularly useful for the interviewees to implement enables the solution to be developed with a broader and structured initial list of the feasible rule types.

6. **What other simplifications would you expect to see in a more user-friendly diff tool?**

   This question aims to collect suggestions for the user experience features since interviewees can offer different perspectives and may have very distinct experiences.

7. **What security threats could you want to detect and prevent in IaC, particularly when re-using community-made code?**

   This question allows us to determine specific scenarios of potential security threats in order to better plan solutions for their detection and treatment.

8. **Would context provided by code annotations aid in interpreting the diff tool?**

   This dissertation might also consider the viability of using code annotations to provide context in infrastructure changes. Asking the interviewees about such a possibility will allow evaluating its convenience and usefulness.

Appendix C

C2A - Change Rules Language Grammar

```
Rules = Rule[]

Rule = { // all fields are optional
    "description": STRING, // textual context for this rule
    "let": {
        Identifier: Query
    },
    "effect": {
        "target": STRING, // identifier of a change,
        "risk": "high"|"low"|"unknown",
        "action": "approve"|"reject"|"none",
    }
    "where": Condition[],
    "then": Rules
}

Identifier = STRING // alfanumeric string with non-numeric first character

Query = PropertyAccess | ComplexQuery

ComplexQuery = ComponentShortQuery | GenericQuery

ComponentShortQuery = {
    STRING: STRING, //Component Type and Component Subtype
    "where": Condition[]
}

GenericQuery = {
    "Component"|"Relationship"|"Property"|"Change": {
        STRING: STRING // filters entities with key-value pairs that match the
                          ones provided
```

```
    },
    "where": Condition[]
}

PropertyAccess = Indentifier(.Accessor)+

Accessor = STRING // alfanumeric string or wildcard "*"

Condition = Operand Operator Operand

Operand = Literal | PropertyAccess

Literal = number | string | boolean

Operator =
    "->", "<-", // has dependency relationship (to/from respectively)
    ">>", "<<", // has structural relationship (to/from respectively)
    "appliesTo" // [change] applies to entity,
    "==", "!=",">",">=","<","<=", // standard scalar operators
```

Appendix D

Final Survey Content

D.1 Background

The performed survey includes background questions to determine the experience level of the participant on the technologies used in the three scenarios. Responders are requested to classify each technology on the following scale: Not experienced; Somewhat experienced; Fairly experienced; Very experienced; Extremely experienced.

The technologies in question are:

- AWS Cloud Services
- AWS Services
- IaC tools
- AWS CDK
- AWS CloudFormation
- AWS EC2 Instances
- AWS VPCs
- AWS Elastic Load Balancing
- AWS Auto Scaling Groups
- AWS Lambdas
- AWSStep Functions
- AWS DynamoDB
- AWS CloudFront

D.2 Scenario 0

In scenario 0 the participants are only presented with the following question:

> "In this survey you will have to review Pull Requests. The following question aims to provide you some context before moving forward.
>
> Please review the PR and identify in the text box below all changes that could be considered: security threats to the infrastructure functionality-breaking changes. As this question is only meant to provide context for the following ones, please try to identify as many changes as you can for 2 minutes and move on to the next question."

Below this question, a hyperlink is provided to the corresponding GitHub PR.

From this answer, the detected changes are checked against the changes identified in Section 7.1.1.1 (p. 58).

D.3 Scenarios 1 and 2

A small introduction is provided in scenarios 1 and 2 before requesting review, respectively:

- "Consider a CDK project designed to deploy a containerized web app. This project has a new Pull Request, which you'll have to review."
- "Consider a CDK project designed to deploy a serverless web app. This project has a new Pull Request, which you'll have to review."

In both scenarios, the participants start by doing reviews similarly to scenario 0. The following question requests this review, recommending spending under 10 minutes:

> "Please review the PR and identify in the text box below all changes that could be considered: security threats to the infrastructure functionality-breaking changes. To keep this survey under the advertised duration, we recommend you limit your feedback to the most relevant changes to the infrastructure you can find under 10 minutes."

For both user groups, a hyperlink is provided to the corresponding GitHub PR. In addition, a hyperlink to the proof-of-concept's user interface (see Section 6.6, p. 46) is also provided if the participant is on the experimental group for that scenario. (see Section 7.1.3.2, p. 61).

Similarly to scenario 0, from this answer, the detected changes are checked against the changes identified in Section 7.1.1.2 (p. 58) and Section 7.1.1.3 (p. 59).

In addition, for each scenario, there are multiple follow-up questions:

- "Please rate your confidence on the two following aspects after having reviewed the code changes" - answered on the following scale: Not at all confident; Somewhat confident; Reasonably confident;Very confident; Extremely confident.

- "How confident are you that you found all potential security threats to the infrastructure?"
- "How confident are you that you found all changes that break infrastructure functionality?"
- "How much time would you say you spent analyzing information that does not affect your ability to approve a PR?"
- "Roughly how many minutes did you spend..."

  "looking for problems in the Github PR?"

  (If the participant is on the scenario's experimental group) "looking for problems in the Change Analysis tool?"

  "describing the problems you found?"
- (If the participant is on the scenario's experimental group) "Was there any change you were unable to detect in the code review that you were able to detect in the Tool? (Feel free to leave any other feedback you might have about the tool)"

Appendix E

Identified Changes

Change	Detection without C2A (%)	
Removed SQS Queue Encryption	20.00	95.24
Give all S3 permissions to Lambda	19.50	92.86
Listing Lambda Functions in Lambda	13.00	61.90
Missing permissions to List Lambdas	10.00	47.62
Removed sqs:SendMessage permission from lambda	8.50	40.48
Set Lambda Timeout to 4s	6.00	28.57
Added Dead Letter Queue (DLQ)	4.00	19.05
Added AWS SDK to project and lambda	4.00	19.05

Table E.1: Scenario 0 - Change Detection Frequency

Change	Time detected without C2A (%)	Times detected with C2A (%)	Detection rate without C2A (%)	Detection rate with C2A (%)	Improvement with C2A (%)
PutItem' action was removed in DynamoDB Endpoint permissions	3.00	5.50	27.27	61.11	124.07
Added sort key to DynamoDB Table schema definition	4.00	4.50	36.36	50.00	37.50
DynamoDB table replaced	3.00	3.00	27.27	37.50	37.50
Allow SSH Logic flipped	4.00	1.00	36.36	11.11	-69.44
Narrower resources in ListTables and DescribeTable permissions	1.00	2.00	9.09	22.22	144.44
ListTables permission cannot have narrower resources	1.00	0.00	9.09	0.00	-100.00

Table E.2: Scenario 1 - Change Detection Frequency

Change	Time detected without C2A (%)	Times detected with C2A (%)	Detection rate without C2A (%)	Detection rate with C2A (%)	Improvement with C2A (%)
Renamed DynamoDB Table will get replaced	4.00	5.00	44.44	71.43	60.71
Step Function invokes SendPush Lambda	1.50	3.00	16.67	42.86	157.14
External Library CloudFormation Origin changed to URL	0.00	1.50	0.00	21.43	Inf
Restrict Lambdas resource access to only the users dynamoDB Table	4.00	1.00	44.44	14.29	-67.86

Table E.3: Scenario 2 - Change Detection Frequency

References

[1] Ansible. Available at https://www.ansible.com/. Accessed: 2021-01-30.

[2] Aws cdk. Available at https://aws.amazon.com/cdk/. Accessed: 2021-01-30.

[3] Aws cdk toolkit commands. Available at https://docs.aws.amazon.com/cdk/latest/guide/cli.html#cli-ref. Accessed: 2021-01-30.

[4] Aws cloudformation. Available at https://aws.amazon.com/cloudformation/. Accessed: 2021-01-30.

[5] Aws cloudformation documentation. Available at https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/. Accessed: 2021-01-30.

[6] Aws cloudformation guard repository. Available at https://github.com/aws-cloudformation/cloudformation-guard. Accessed: 2021-01-30.

[7] Chef. Available at https://docs.chef.io/chef_overview/. Accessed: 2021-01-30.

[8] Chef compliance. Available at https://www.chef.io/products/chef-compliance. Accessed: 2021-01-30.

[9] Chef provisioning. Available at https://github.com/chef-boneyard/chef-provisioning. Accessed: 2021-01-30.

[10] Cloudformation template specification used in aws cdk. Available at https://github.com/aws/aws-cdk/tree/master/packages/%40aws-cdk/cfnspec. Accessed: 2021-05-30.

[11] Cypher general purpose graph querying language. Available at https://neo4j.com/developer/cypher/. Accessed: 2021-05-31.

[12] Github pull request reviews. Available at https://docs.github.com/en/github/collaborating-with-pull-requests/reviewing-changes-in-pull-requests/about-pull-request-reviews. Accessed: 2021-06-25.

[13] Oasis tosca - frequently asked questions. Available at https://www.oasis-open.org/committees/tosca/faq.php. Accessed: 2021-01-30.

[14] Pulumi. Available at https://www.pulumi.com/. Accessed: 2021-01-30.

[15] Pulumi's preview command. Available at https://www.pulumi.com/docs/reference/cli/pulumi_preview/. Accessed: 2021-01-30.

[16] Puppet. Available at https://puppet.com/. Accessed: 2021-01-30.

[17] Puppet. Available at https://github.com/puppetlabs/puppetlabs-aws. Accessed: 2021-01-30.

[18] Puppet impact analysis. Available at https://puppet.com/docs/continuous-delivery/4.x/impact_analysis.html. Accessed: 2021-01-30.

[19] Terraform. Available at https://www.terraform.io/. Accessed: 2021-01-30.

[20] Terraform sentinel. Available at https://www.terraform.io/docs/cloud/sentinel/index.html. Accessed: 2021-01-30.

[21] Terraform's plan command. Available at https://www.terraform.io/docs/cli/commands/plan.html. Accessed: 2021-01-30.

[22] Yaml tags. Available at https://yaml.org/spec/1.2/spec.html#id2761292. Accessed: 2021-05-30.

[23] Anca Apostu, Florina Puican, Geanina Ularu, George Suciu, Gyorgy Todoran, et al. Study on advantages and disadvantages of cloud computing–the advantages of telemetry applications in the cloud. *Recent advances in applied computer science and digital services*, 2103, 2013.

[24] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721, 2013.

[25] Salman Baset, Sahil Suneja, Nilton Bila, Ozan Tuncer, and Canturk Isci. Usable declarative configuration specification and validation for applications, systems, and cloud. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Industrial Track*, Middleware '17, page 29–35, New York, NY, USA, 2017. Association for Computing Machinery.

[26] Tobias Baum, Kurt Schneider, and Alberto Bacchelli. On the optimal order of reading source code changes for review. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 329–340. IEEE, 2017.

[27] Tiago Boldt Pereira de Sousa. *Engineering Software for the Cloud: A Pattern Language*. PhD thesis, Faculty of Engineering, University of Porto, 2020.

[28] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano. Devops. *IEEE Software*, 33(3):94–100, 2016.

[29] Johann Eder and Karl Wiggisser. A dag comparison algorithm and its application to temporal data warehousing. In John F. Roddick, V. Richard Benjamins, Samira Si-said Cherfi, Roger Chiang, Christophe Claramunt, Ramez A. Elmasri, Fabio Grandi, Hyoil Han, Martin Hepp, Miltiadis D. Lytras, Vojislav B. Mišić, Geert Poels, Il-Yeol Song, Juan Trujillo, and Christelle Vangenot, editors, *Advances in Conceptual Modeling - Theory and Practice*, pages 217–226, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[30] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*, pages 321–330, 2008.

[31] David MacKenzie, Paul Eggert, and Richard Stallman. Comparing and merging files with gnu diff and patch. *Network Theory Ltd*, 4, 2002.

[32] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[33] M. Menarini, Y. Yan, and W. G. Griswold. Semantics-assisted code review: An efficient tool chain and a user study. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 554–565, 2017.

[34] S. Meyer, P. Healy, T. Lynn, and J. Morrison. Quality assurance for open source software configuration management. In *2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 454–461, 2013.

[35] Kief Morris. *Infrastructure as Code*. O'Reilly Media, 2020.

[36] Kathryn E Newcomer, Harry P Hatry, and Joseph S Wholey. Conducting semi-structured interviews. *Handbook of practical program evaluation*, 492, 2015.

[37] R. Opdebeeck, A. Zerouali, C. Velázquez-Rodríguez, and C. D. Roover. Does infrastructure as code adhere to semantic versioning? an analysis of ansible role evolution. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 238–248, 2020.

[38] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Complete and accurate clone detection in graph-based models. In *2009 IEEE 31st International Conference on Software Engineering*, pages 276–286, 2009.

[39] Tiago Sousa, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. A survey on the adoption of patterns for engineering software for the cloud. *IEEE Transactions on Software Engineering*, pages 1–1, 2021.

[40] Tiago Boldt Sousa, Ademar Aguiar, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. Engineering software for the cloud: patterns and sequences. In *Proceedings of the 11th Latin-American Conference on Pattern Languages of Programming*, pages 1–8, 2016.

[41] Tiago Boldt Sousa, Filipe Figueiredo Correia, and Hugo Sereno Ferreira. Patterns for software orchestration on the cloud. In *Proceedings of the 22nd Conference on Pattern Languages of Programs*, pages 1–12, 2015.

[42] Tiago Boldt Sousa, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. Overview of a pattern language for engineering software for the cloud. In *Proceedings of the 25th Conference on Pattern Languages of Programs*, pages 1–9, 2018.

[43] Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. Engineering software for the cloud: Messaging systems and logging. In *Proceedings of the 22Nd European Conference on Pattern Languages of Programs*, pages 1–14, 2017.

[44] Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. Engineering software for the cloud: Automated recovery and scheduler. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, pages 1–8, 2018.

[45] Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. Engineering software for the cloud: External monitoring and failure injection. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, pages 1–8, 2018.

[46] OASIS Standard. Topology and orchestration specification for cloud applications version 1.0, 2013.

[47] Luis M Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition, 2008.

[48] Johannes Wettinger, Uwe Breitenbücher, and Frank Leymann. Standards-based devops automation and integration using tosca. In *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 59–68. IEEE, 2014.

[49] Johannes Wettinger, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. Streamlining devops automation for cloud applications using tosca as standardized metamodel. *Future Generation Computer Systems*, 56:317 – 332, 2016.

[50] Michael Wurster, Uwe Breitenbücher, Michael Falkenthal, Christoph Krieger, Frank Leymann, Karoline Saatkamp, and Jacopo Soldani. The essential deployment metamodel: a systematic review of deployment automation technologies. *SICS Software-Intensive Cyber-Physical Systems*, 35(1):63–75, 2020.