# Reliable Infrastructure as Code for Decentralized Organizations

DISSERTATION

of the University of St. Gallen,
School of Management,
Economics, Law, Social Sciences,
International Affairs and Computer Science,
to obtain the title of
Doctor of Philosophy in Computer Science

submitted by
**Daniel Sokolowski**
from
Germany

Approved on the application of
**Prof. Guido Salvaneschi, Ph.D.**
and
**Prof. Dr. Jürgen Cito**

Dissertation no. 5462

Copy Blitz AG, St. Gallen, 2024

The University of St.Gallen, School of Management, Economics, Law, Social Sciences, International Affairs and Computer Science, hereby consents to the printing of the present dissertation, without hereby expressing any opinion on the views herein expressed.

St. Gallen, May 22, 2024

The President:

Prof. Dr. Manuel Ammann

# Acknowledgements

I thank my advisor, Guido Salvaneschi, for his support throughout my PhD journey, especially for challenging ideas and directions at many crucial points. This entire undertaking was only possible thanks to the freedom and plethora of opportunities you provided, enabling me to grow, exchange with others, and ideate, shape, work on, and scrap many ideas and projects, of which the most central outcomes constitute this thesis.

I thank Jürgen Cito for co-advising me since our first in-person meeting at the ESEC/FSE Doctoral Symposium 2022 in Singapore. Your feedback was invaluable in positioning and shaping my work, and I particularly thank you for opening doors to a new community through the Dagstuhl seminar 23082 [48] and CONFLANG workshops.

Further, I thank the sheer number of other people I exchanged and worked with during the past five years, providing me with endless insights and inspiration. First, this includes especially my PhD and PostDoc colleagues at the Programming Group (former: Reactive Programming Technology group) Benyamin Delshad, Nafise Eskandani, Mirko Köhler, Aditya Oak, Simon Schönwälder, David Spielmann, Felix Weirich, Pascal Weisenburger, and George Zakhour. Second, I appreciate the many supportive and inspiring interactions throughout my PhD with Mira Mezini and her Software Technology group at TU Darmstadt, where I especially want to highlight the frequent interactions with Ragnar Mogk, David Richter, and Matthias Eichholz. Third, it was a pleasure to work with the researchers with whom I collaborated and all the students I had the chance to supervise, especially those working on their theses with me. Many of them are truly brilliant. Lastly, I value the exchange with all the other people at TU Darmstadt, Uni St. Gallen, and many places beyond. Thanks also to the organizers of the retreats, conferences, workshops, etc. I attended; the pandemic emphasized how vital these networking and exchange occasions have been for me.

I especially thank Gudrun Harris, Claudia Roßmann, Eriane Breu, and Florence Hafner for handling many administrative burdens smoothly.

Finally, I thank my family, Soumya Shivananda, and close friends for their continuous unconditional support, including all the much-appreciated moments of distraction, and everyone else who made life joyful.

St. Gallen, May 22, 2024 Daniel Sokolowski

# Abstract

IT must be reliable for organizations to thrive and quickly adaptable for their swift reaction to their environment. Agility is vital, and DevOps achieves these goals by empowering independent cross-functional teams in decentralized organizations and automating the entire software pipeline. Infrastructure as Code (IaC) is the critical tool to automate software operations, including infrastructure provisioning, application deployment, and configuration. Beyond simple IaC scripts, developers implement IaC programs in programming languages like TypeScript and Python. Such IaC programs are software, and their reliability is crucial to the functionality and security of the deployed systems. Still, techniques for the rapid development of reliable IaC programs are missing, limiting organizations' agility. Specifically, developers lack automation for deployment coordination and updating and quality assurance tools for, e.g., testing and verification.

We surveyed 134 IT professionals, finding that coordination across deployments is commonly needed and often requires manual coordination, even though IT professionals believe automated coordination yields better agility. However, automated approaches are centralized, limiting team independence and agility in decentralized organizations. To solve this issue, we propose automating coordination across deployments in a decentralized fashion through *μ*S ([mjuːz] *"muse"*), a novel IaC solution. With *μ*S, teams have separate IaC programs, which express and jointly automate the correct order of operations across deployments. We further show how *μ*S enables safe updating through IaC programs, preventing updates from breaking distributed transactions or workflows.

Beyond automating the coordination of IaC programs, we address the reliability of IaC program code. To unblock studies, we built a dataset of 37 712 public IaC programs. In initial analyses, only a vanishing fraction implements tests. We identified that available testing techniques are either slow and resource-intensive or require excessive development effort. To solve this dilemma, we propose ACT, an extensible automated unit testing approach that enables testing IaC programs quickly in hundreds of configurations, often without writing additional testing code.

This dissertation studies the coordination and testing of IaC programs. Empirically motivated, we present *μ*S for safe deployment coordination and updating in decentralized setups and ACT for efficient testing of IaC programs. Our contributions nurture future research and reliable deployments in decentralized organizations, ensuring agility.

# Zusammenfassung

IT muss zuverlässig und anpassungsfähig sein, damit Unternehmen florieren und schnell auf ihre Umgebung reagieren können. Agilität ist entscheidend, und DevOps erreicht diese Ziele, indem es unabhängige, funktionsübergreifende Teams in dezentralen Organisationen stärkt und die gesamte Software-Pipeline automatisiert. Infrastructure as Code (IaC) ist das entscheidende Tool zur Automatisierung des Softwarebetriebs, einschließlich Deployment und Konfiguration. Mittlerweile implementieren Entwickler IaC-Programme in Programmiersprachen wie TypeScript und Python. Solche IaC-Programme sind Software und ihre Zuverlässigkeit ist entscheidend für die Funktionalität und Sicherheit der Systeme. Dennoch fehlen Techniken für die schnelle Entwicklung zuverlässiger IaC-Programme. Insbesondere mangelt es an Automatisierung für Koordinierung und Aktualisierung sowie an Qualitätssicherungswerkzeugen, z.B., zum Testen und Verifizieren.

Wir befragten 134 IT-Experten und stellten fest, dass Koordination über Deployments hinweg erforderlich und oft manuell ist, obwohl IT-Experten der Meinung sind, dass automatisierte Koordination zu besserer Agilität führt. Allerdings sind automatisierte Ansätze zentralisiert, was die Unabhängigkeit von Teams in dezentralen Organisationen einschränkt. Um dieses Problem zu lösen, koordinieren wir Deployments dezentral mit $\mu$S ([mjuːz] *“muse”*), einer neuartigen IaC-Lösung. Mit $\mu$S verfügen Teams über separate IaC-Programme, die den Koordinationsbedarf ausdrücken und gemeinsam automatisieren. Wir zeigen außerdem, wie $\mu$S sichere Aktualisierungen durch IaC-Programme ermöglicht, die verhindern, dass Aktualisierungen verteilte Transaktionen abbrechen.

Zudem befassen wir uns mit der Zuverlässigkeit des IaC-Programmcodes. Um Studien zu ermöglichen, erstellten wir einen Datensatz mit 37 712 öffentlichen IaC-Programmen und analysierten, dass nur für einen verschwindenden Anteil Tests implementiert wurden. Verfügbare Testtechniken sind entweder langsam und ressourcenintensiv oder sie erfordern übermäßigen Entwicklungsaufwand. Um dieses Dilemma zu lösen, präsentieren wir ACT, eine erweiterbare automatisierte Unit-Test-Lösung, die das schnelle Testen von IaC-Programmen in Hunderten Konfigurationen ermöglicht.

Diese Dissertation untersucht die Koordination und das Testen von IaC-Programmen. Wir stellen $\mu$S für die sichere Koordinierung von Deployments in dezentralen Umgebungen und ACT für das effiziente Testen von IaC-Programmen vor. Unsere Beiträge fördern zukünftige Forschung und die Agilität von dezentralen Organisationen.

# Table of Contents

# List of Algorithms

# List of Figures

# List of Listings

# List of Tables

# Chapter 1

# Introduction[1]

Software is pervasive in all industries and must adapt quickly to changing requirements while being stable and robust despite frequent updates. The past two decades have seen various IT approaches aiming at these goals, e.g., Unified Process, Scrum, and Extreme Programming, converging into DevOps [63, 70, 125]. The key focus of DevOps is to enable the frequent development of software updates and ensure reliable software operations, improving satisfaction through iterative feedback and higher velocity. As the base philosophy of modern IT organizations, DevOps inspired a range of practices with additional focus and insights, e.g., GitOps, MLOps, and DevSecOps.

The objectives of DevOps are well expressed and commonly measured through the core Software Delivery and Operational (SDO) performance metrics, also known as DevOps metrics. Forsgren et al. [79] developed them for their annual State of DevOps reports as (1) deployment frequency, (2) lead time between the development and deployment of changes, (3) required time to restore service on failure, and (4) the rate of failed changes. In recent reports, these key metrics are accompanied by a fifth one measuring availability and, since 2021, reliability. In contrast to outdated concepts, studies show that, in practice, throughput metrics (e.g., higher deployment frequency) correlate positively with service stability metrics (e.g., lower change failure rate) [79]. Achieving good SDO performance requires both organizational and technical innovation.

On an organizational level, DevOps aims to reduce the friction between software development and operations. Traditionally, software development and operations have been separated, where operations summarizes all activities after the development, including configuration, resource provisioning and deployment, monitoring, alarming, and reporting. Thereby, the separated silos had seemingly contradicting goals. While developers focused on changing requirements and software quality, aiming for minimal change re-

[1] Based on the authors' work in [225, 226, 228, 229, 232, 233, 234, 235]. [228] © 2023 IEEE. [229] © 2024 IEEE. [235] © 2023 IEEE.

sponse time, operators focused on stability and reliability, which are typically assumed to be threatened by frequent change. DevOps aims to mitigate this tension by strengthening the collaboration between development and operations staff, often by unifying both tasks in cross-functional teams, leading to a smoother workflow [167] and improving service quality [140] and change management performance [36]. Cross-functional teams are ideally independent of other teams, yielding decentralized organizations in which each team is responsible for all concerns of its software applications.

On a technical level, the premise for good SDO performance is a high degree of automation along the whole software pipeline [70, 111]. DevOps drives operations automation through Continuous Integration (CI) [112] and Infrastructure as Code (IaC) [162]. The latter automates managing the IT infrastructure with machine-readable files, i.e., code, replacing manual configuration via interactive configuration tools. Such IaC definitions allow versioning, debugging, updating, and reviewing of the infrastructure setup, reusing well-developed techniques from traditional application code. As a result, IaC enables faster, more reproducible software operations [97, 134, 140, 194, 208].

Early examples of IaC solutions include CFEngine [166], Puppet [189], Chef [180], and Ansible [203], which blend *imperative* infrastructure management with *declarative* concepts to varying extents. In purely declarative approaches, IaC scripts only describe a target state, and the system automatically derives the operations to achieve it, providing better stability and less maintenance [35, 72, 197]. Meanwhile, these systems are also called Configuration as Code (CaC) because they focus on configuring mutable infrastructure. In contrast, IaC solutions like Terraform [104], AWS CloudFormation [11], and Azure Resource Manager (ARM) [159] focus on provisioning immutable cloud infrastructure, e.g., serverless functions, containers, databases, and storage.

Typically, developers write IaC scripts in configuration languages like JSON and YAML, which is tedious for big and complex deployments and has led to DSLs like HCL [103] and Bicep [160]. Still, the setup of modern cloud applications is increasingly complex. Modern applications often comprise many small components, e.g., serverless functions, microservices, smaller databases, and blob storage. This trend transfers complexity from inside big monolithic applications to the composition of these components, resulting in long, structured IaC scripts.

In contrast, Programming Languages IaC (PL-IaC) solutions address complexity using general-purpose programming languages like Python and TypeScript instead of DSLs. With PL-IaC, developers write IaC programs—not IaC scripts. Using (imperative)

Listing 1.1: Static Website (SW): A Pulumi IaC program that deploys a static website on AWS S3, implemented in TypeScript (left) and Python (right).[2]

```typescript
import * as aws from "@pulumi/aws";

const bucket =
  new aws.s3.Bucket("website", {
    website: {
      indexDocument: "index.html",
    },
  });
new aws.s3.BucketObject("index", {
  bucket: bucket,
  content: "<!DOCTYPE html>Hello!",
  key: "index.html",
  contentType: "text/html",
});

export const url =
    bucket.websiteEndpoint;
```

```python
import pulumi
import pulumi_aws as aws

bucket = aws.s3.Bucket("website",
  website=aws.s3.BucketWebsiteArgs(
    index_document="index.html"
  )
)
aws.s3.BucketObject("index",
  bucket=bucket,
  content="<!DOCTYPE html>Hello!",
  key="index.html",
  content_type="text/html"
)

pulumi.export("url",
    bucket.website_endpoint)
```

programming languages provides developers with powerful abstractions they already know, making it easy to tackle the complexity of deployments. Still, PL-IaC solutions ensure the (imperative) IaC programs are declarative, i.e., developers only express the intended target state, not how to achieve it. For instance, Listing 1.1 shows the SW example, the deployment of a simple static website on AWS S3 in Pulumi TypeScript and Python. Both versions are simple imperative programs that define two resources in the declarative target state: the bucket and the bucket object hosted in it.

The industrial-strength PL-IaC solutions available today are Pulumi [186] and the Cloud Development Kits (CDKs) for Amazon Web Services (AWS CDK) [8] and Terraform (CDKTF) [101]. They have existed since 2018–2020 with quickly growing communities. Pulumi reported a ~10× growth from hundreds to 2 000 customers and tens of thousands to 150 000 end users from 2020 to 2023 [67, 68], and the total annual downloads of all solutions' core packages on NPM grew even faster from 11 M downloads to 146 M downloads in this time.[3] We expect this growth to continue rapidly and PL-IaC to become much more popular than it is today. This is driven by the increasing complexity of application setups and deployments and developers' excitement about tackling them with modern tools that leverage languages and ecosystems they already know.

[2] For brevity, we omit the bucket's ownership controls, public access block, and policy resources that are required to allow public access from the Internet.

[3] https://npm-stat.com/ for `aws-cdk-lib`, `@aws-cdk/core`, `cdktf`, and `@pulumi/pulumi`.

While addressing general IaC issues, we focus on PL-IaC because—beyond its growing popularity—it is a novel, powerful technique for complex deployments, promising easier reuse and better applicability of existing software engineering techniques and tools. Further, despite its growing popularity, PL-IaC is understudied. Previous research on IaC reliability is focused and widely limited to CaC solutions like Ansible, Chef, and Puppet.

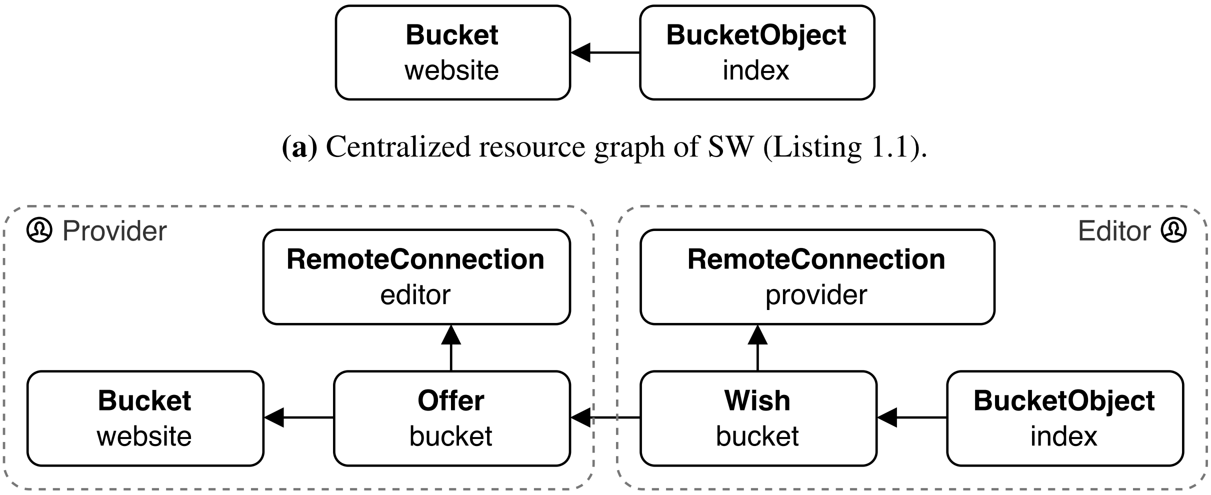
The reliability of IaC programs is imperative. In the best case, faulty IaC programs do not deploy an application at all or only partly, causing the system to be unavailable or malfunction. In the worst case, the faulty IaC program deploys the application so that it works correctly, but the error causes an insecure setup with vulnerabilities. Colloquially, *reliable* systems "just work." This aligns with Meyer's definition: Reliability is a more general term encompassing *correctness* and *robustness* [158]. Correctness describes that the program performs tasks as specified. Robustness describes that the program reacts appropriately to abnormal conditions. IaC programs must satisfy both—work as intended, even in a changed environment.

This dissertation contributes to reliable IaC for decentralized organizations by introducing new features for IaC programs, enabling decentralized organizations to address and safely automate their requirements, and by improving tool support, helping developers write correct IaC program code. We now introduce our work on safe coordination of IaC programs (Section 1.1) and our work on quality assurance tooling for IaC programs (Section 1.2). Section 1.3 summarizes the dissertation's contributions, and Section 1.4 the publications it comprises. Finally, Section 1.5 describes the outline of this document.

## 1.1 Coordination of Deployments

In declarative IaC, users define the target state of the infrastructure as a directed acyclic graph (DAG), the *resource graph*, where each node is a resource, e.g., a database, container, or network ACL entry, and arcs are dependencies between them, typically due to a contained-in or requires relationship [262]. These dependencies are transitive and order the deployment, i.e., if resource $R$ depends on $S$, $S$ must be deployed before $R$, and $R$ must not be deployed when $S$ is undeployed. This applies to PL-IaC, too. For instance, Listing 1.1 defines the resource graph in Figure 1.1a. Due to the dependency between the resources, the index must be deployed after and undeployed before the bucket.

Ideally, DevOps teams are independent. However, we observed that applications often depend on one another, leaving us to wonder whether such dependencies require teams to coordinate deployment times. For example, we consider a decentralized version



**(a)** Centralized resource graph of SW (Listing 1.1).

**(b)** Decentralized global resource graph of SW in µS (Listings 1.2 and 1.3).

**Figure 1.1:** Centralized and decentralized resource graph of SW.

of the SW example, which is split across two teams. The *provider* is responsible for the bucket and the *editor* for the page in it. To ensure that the index page is only deployed when the bucket is, both teams need to coordinate whenever the bucket is deployed, updated, or undeployed. This decreases the flexibility of the teams and wastes time due to synchronization at every deployment action. However, it also contradicts the core goals of modern software architecture paradigms like Microservices, where applications shall be independently deployable [142, 165]. Yet, do practitioners achieve such independence?

We surveyed 134 IT professionals in the Dependencies in DevOps Survey 2021 to assess the state of dependencies across application deployments and whether and how they lead to coordination requirements. We found that dependencies across applications are common and often constrain the order of their (un)deployment. To coordinate deployment times, practitioners commonly use manual coordination, e.g., via phone, email, or chat, which contradicts DevOps' automation paradigm, slows down evolution [36, 140], and is error-prone [167]. Hence, despite relying on manual coordination, practitioners believe that automated coordination promises better SDO performance—their goal.

We analyzed why deployment coordination is often manual and noticed that all automated coordination solutions are centralized. While such centralized solutions seem suitable for organizations with dedicated teams centrally maintaining state-of-the-art development and deployment platforms, e.g., at Meta [95], generally, such centralization hinders teams' independence, contradicting DevOps goals. Hence, we argue that a solution is missing that automates deployment coordination for organizations with cross-

Listing 1.2: The provider's μS IaC program of the decentralized SW example.

```
1.2.1 const editor = new RemoteConnection("editor");
1.2.2 const bucket = new aws.s3.Bucket("website", {
1.2.3   website: { indexDocument: "index.html" },
1.2.4 });
1.2.5 new Offer(editor, "bucket", bucket);
```

Listing 1.3: The editor's μS IaC program of the decentralized SW example.

```
1.3.1 const provider = new RemoteConnection("provider");
1.3.2 const wish = new Wish<aws.s3.Bucket>(provider, "bucket");
1.3.3 new aws.s3.BucketObject("index", {
1.3.4   bucket: wish.offer,
1.3.5   content: "<!DOCTYPE html>Hello!",
1.3.6   key: "index.html",
1.3.7   contentType: "text/html",
1.3.8 });
```

functional teams in a *decentralized* fashion. Such automation enables decentralized organizations to be compatible with DevOps goals in the presence of inevitable application dependencies across teams, improving their SDO performance.

To fill this gap, we propose μS ([mjuːz] *"muse"*), the first PL-IaC solution allowing the *expression* and *automation* of decentralized deployment coordination. With μS, every team has its own IaC program, which defines interfaces to other teams' deployments. Teams define a `RemoteConnection` for each connected deployment and `Offer` resources to provide values and resources to remote deployments. In the consuming deployments, teams define corresponding `Wish` resources. Together, offers and wishes define a contract expressing the assumptions about the deployments' connection on the providing and consuming sides. In our decentralized SW example, the provider and editor specify their connection (Lines 1.2.1 and 1.3.1). The provider offers their bucket (Lines 1.2.2 to 1.2.4) to the editor's deployment in Line 1.2.5. The editor specifies their expectation of the offer in Line 1.3.2, and uses the offered bucket via `wish.offer` in Line 1.3.4. Jointly, Listings 1.2 and 1.3 define the resource graph Figure 1.1b.

To automate the coordination of deployments across teams, μS deployments are continuously running processes, not one-off tasks like, e.g., in Pulumi, leveraging the information expressed in wishes and offers. In particular, μS ensures wishes and dependent resources are only deployed when the corresponding offer is deployed, and that changes are reactively propagated from offers to the corresponding wishes. This way, the editor can start their deployment independently and run it continuously. Whenever the provider starts or updates their deployment, the editor's deployment automatically deploys, updates, or undeploys the index page without manual intervention. Our solution guarantees the correct deployment and undeployment order for dependencies across deployments of different teams without introducing a central authority or requiring manual coordination. Thus, μS enables safe deployments in decentralized organizations.

So far, we addressed deployment coordination for coupled applications. Yet, μS' novel runtime enables reliable coordination of updates in environments where deployments are fully decoupled, as demanded, e.g., by Microservices architectures. In this case, independent deployments require fault tolerance, but distributed transactions spanning multiple separately deployed applications are still possible if the system detects transactions that failed due to an update and repeats them. However, if these transactions are frequent or take long, e.g., *workflows*, breaking and repeating them is infeasible due to the required additional resources and introduced delays. Safe Dynamic Software Updating (DSU) solves this issue by identifying *when* to update a component in a running system such that no transaction breaks. Unfortunately, research on safe DSU [24, 128, 152, 256] has not been accessible yet for workflows in decentralized organizations.

To close this gap, we introduce a new unified model for safe DSU for workflows and a modular information dissemination and control algorithm for safe DSU in decentralized organizations. Still, safe DSU requires the implementation of an extension for the components' orchestrators. However, alternatively, safe DSU can be implemented in μS IaC programs because they are long-running and can react to external signals, in contrast to previous IaC solutions. Implementing the mechanism in the IaC program is desirable from a reliability standpoint because it allows testing and holistic reasoning about the deployment and its behavior on the deployment's IaC program. Otherwise, the whole orchestrator and its safe DSU extension also have to be considered. Further, to reduce the performance impact of current safe DSU approaches, we propose and evaluate Essential Safety, an optimized safe DSU approach retaining strong update safety guarantees.

## 1.2 Reliability of IaC Programs

For reliable deployments, it is not enough that IaC solutions allow developers to express and automate all requirements in IaC programs, e.g., for coordination among deployments. Additionally, the developer's IaC program code must be reliable and correctly describe their requirements. Luckily, code quality is a well-explored problem for software in general and has been addressed with various code quality assurance techniques, including testing and verification. As developers write IaC programs in general-purpose programming languages, there is great potential to apply existing software engineering methods to them, and many tools available for these languages apply out-of-the-box. However, such an application implies that IaC programs have properties and problems similar to those of other software, and it does not leverage IaC-specific insights. Even worse, the similarities and differences between IaC programs and other software are generally neither studied nor well understood, and there are not even datasets to start these explorations. Yet, investigating these issues is crucial to transferring existing software engineering techniques and developing new ones optimized for PL-IaC.

To shed light on PL-IaC in practice and enable studies on real-world IaC programs, we built the open-source dataset PIPr [230], the first systematic dataset of IaC programs. PIPr comprises metadata of 37 712 IaC programs from 21 445 public GitHub repositories and shallow copies (i.e., without history) of the ones permitting redistribution. PIPr enables researchers to investigate PL-IaC in-depth and understand the similarities and differences of IaC programs compared to other software.

As initial analyses of PIPr, we inspected all IaC programs for their (1) programming languages, (2) testing techniques, and (3) licenses. Most interestingly, we found that only 25 % of the PL-IaC programs use testing, dropping to 1 % for general PL-IaC, which only Pulumi implements. This low share suggests an issue with current PL-IaC testing because testing is generally much more popular. For example, previous studies on public software projects on GitHub found that more than 50% use testing [154, 222]. Other researchers have found that testing IaC is an open research problem, too, and critical in practice. For example, Rahman et al. [194] urge in their mapping study of IaC research for more work on testing, and Guerriero et al. [97] found that declarativity and "impossible testing" are the most mentioned differences between IaC and traditional software in 44 semi-structured interviews with senior developers.

To understand why the share of IaC programs with tests is vanishingly small, we analyzed the current testing techniques for general PL-IaC solutions and noticed that they pose a dilemma: available integration testing techniques are notoriously slow and can cause high infrastructure costs. Unit testing is the only alternative without these issues, but insightful unit tests for IaC programs require high development effort. Every resource definition has to be replaced with a mock faithfully modeling the cloud resource and implementing configuration validation and generation logic. Mocking code easily becomes more complex than the IaC program under test itself, resulting in very few projects using systematic testing—despite testing being crucial for the high-velocity development of reliable software [111, 112].

To enable efficient testing of PL-IaC programs, we propose Automated Configuration Testing (ACT). ACT is an automated framework that allows developers to quickly unit-test PL-IaC programs in hundreds of configurations by combining ideas from property-based testing (PBT) [49, 78] and fuzzing [266]. ACT automatically mocks all resource definitions in the PL-IaC program and uses a generator that provides test input and a set of oracles to validate the resource configurations. ACT is open and provides a plugin mechanism for test generators and oracles. This way, developers and researchers can exchange plugins and, crucially, reuse them among IaC programs. Once the community has developed generally applicable generators and oracles, developers can take them off the shelf and test their IaC programs without writing additional code, minimizing the effort to test IaC programs. Further, ProTI enables developers to conveniently augment these generalized strategies with application-specific insights through ad-hoc specification syntax, enabling fine-tuning generators and oracles directly in the IaC program code.

We implemented ACT in the open-source testing tool ProTI [231] for Pulumi TypeScript with a default test generator and oracle leveraging type information from Pulumi package schemas. The evaluation on 6 081 Pulumi TypeScript programs from GitHub and generated artificial benchmarks shows that (1) ProTI can find bugs reliably *and* quickly compared to existing testing techniques, (2) ProTI can be applied to IaC programs without any changes, (3) ProTI finds bugs often within seconds or tens of seconds, and (4) ProTI can leverage existing generator and oracle tools through simple plugins.

## 1.3 Contributions

Enabling the expression and safe automation of coordination among deployments in a decentralized fashion and addressing code quality through automated unit testing are significant advancements for the reliability of IaC programs in decentralized organizations. In summary, this dissertation contributes the following:

1. Beyond describing the state of the art of PL-IaC and tools for IaC programs, we present a conceptual model of the deployment state evolution in IaC programs.

2. We surveyed 134 IT professionals, showing that dependencies among applications are common, they often constrain the order of application (un)deployment, and

that, in practice, developers resort to manual coordination, even though they believe automated coordination promises better SDO performance.

3. We propose μS, a novel approach for coordinated deployments across teams that neither requires centralization nor manual coordination, ensuring safe deployments in decentralized organizations.

4. We implemented μS as an open-source PL-IaC solution based on Pulumi TypeScript [240], extending the IaC language through new resource types and introducing a new runtime for long-running IaC programs.

5. We evaluated μS on a microservices application and artificial benchmarks, showing that it can effectively coordinate decentralized deployments with negligible coding and performance overhead and applies to existing decentralized IaC programs.

6. We show how deployment coordination, as enabled through μS, can be leveraged for safe Dynamic Software Updating (DSU) in decentralized organizations, i.e., ensuring deployment updates do not break distributed transactions (workflows).

7. We propose a formal model for safe DSU and a corresponding dissemination algorithm to enable safe DSU in decentralized organizations, capturing state-of-the-art DSU approaches and supporting asynchronous workflows.

8. We propose Essential Safety as a novel approach for safe DSU, which leverages the identification of whether an update introduces a semantic change, i.e., is essential.

9. We analytically compared Essential Safety to previous DSU approaches, showing that Version Consistency is a conservative over-approximation of Tranquility and Essential Safety, explaining their performance differences.

10. We evaluated our dissemination algorithm with all discussed safe DSU approaches by simulating 106 realistic collaborative BPMN workflows and analyzing eight monorepos. Essential Safety provides the best performance and, in practice, at least 60 % and often more than 90 % of the updates are non-essential changes.

11. We present PIPr [230], the first systematic PL-IaC dataset containing metadata of 37 712 IaC programs from 21 445 public GitHub repositories, including the code of 15 504 IaC programs whose licenses permit redistribution.

12. We analyzed the IaC programs in PIPr for their (1) programming language, (2) testing techniques, and (3) licenses, noticing that developers barely use testing.

13. We establish the testing dilemma of PL-IaC, explaining why developers rarely test IaC programs: they either have to resort to resource-intensive, slow integration testing or invest comparatively high effort to develop suitable unit tests.

14. We propose Automated Configuration Testing (ACT), a novel approach for efficient PL-IaC testing, embracing automation and reuse on a community level to minimize the development effort to unit-test IaC programs.

15. We implemented ACT in ProTI [231], an open-source automated testing tool for Pulumi TypeScript IaC programs that is extensible through plugins, and provide default type-based oracles and generators based on Pulumi package schemas.

16. We evaluated ProTI on all 6 081 Pulumi TypeScript programs in PIPr and artificial benchmarks, showing that ProTI applies to existing IaC programs, can efficiently find bugs, and can leverage existing tools as test generators and oracles.

## 1.4 Publications

This dissertation led to the following publications at peer-reviewed journals, conferences, and workshops. Their content is used verbatim in this dissertation as stated below.

[233] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. “Automating Serverless Deployments for DevOps Organizations”. In: *ESEC/FSE ’21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. 2021, pp. 57–69. DOI: 10.1145/3468264.3468575
[Used in Chapters 1 to 4]

[225] Daniel Sokolowski. “Deployment Coordination for Cross-functional DevOps Teams”. In: *ESEC/FSE ’21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. 2021, pp. 1630–1634. DOI: 10.1145/3468264.3473101
[Used in Chapters 1, 2 and 4]

[234] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. “Change Is the Only Constant: Dynamic Updates for Workflows”. In: *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. 2022, pp. 350–362. DOI: 10.1145/3510003.3510065
[Used in Chapters 1, 2 and 5]

[226] Daniel Sokolowski. “Infrastructure as Code for Dynamic Deployments”. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. 2022, pp. 1775–1779. DOI: 10.1145/3540250.3558912
[Used in Chapters 1 and 8]

[235] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. “Decentralizing Infrastructure as Code”. In: *IEEE Software* 40.1 (2023), pp. 50–55. DOI: 10.1109/MS.2022.3192968
[Used in Chapters 1 and 3]

[228] Daniel Sokolowski and Guido Salvaneschi. “Towards Reliable Infrastructure as Code”. In: *20th International Conference on Software Architecture, ICSA 2023 - Companion, L’Aquila, Italy, March 13-17, 2023*. 2023, pp. 318–321. DOI: 10.1109/ICSA-C57050.2023.00072
[Used in Chapters 1 and 8]

[232] Daniel Sokolowski, David Spielmann, and Guido Salvaneschi. “The PIPr Dataset of Public Infrastructure as Code Programs”. In: *21th IEEE/ACM International Conference on Mining Software Repositories, MSR 2024, Lisbon, Portugal, April 15-16, 2024*. 2024, pp. 498–503. DOI: 10.1145/3643991.3644888
[Used in Chapters 1, 6 and 8]

[229] Daniel Sokolowski, David Spielmann, and Guido Salvaneschi. “Automated Infrastructure as Code Program Testing”. In: *IEEE Transactions on Software Engineering* (2024). DOI: 10.1109/TSE.2024.3393070
[Used in Chapters 1, 2 and 7]

Beyond the publications constituting this dissertation, I (co)authored the following peer-reviewed articles during my doctoral studies.

[211] Guido Salvaneschi, Mirko Köhler, Daniel Sokolowski, Philipp Haller, Sebastian Erdweg, and Mira Mezini. "Language-integrated Privacy-aware Distributed Queries". In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), 167:1–167:30. DOI: 10.1145/3360593

[227] Daniel Sokolowski, Jan-Patrick Lehr, Christian H. Bischof, and Guido Salvaneschi. "Leveraging Hybrid Cloud HPC with Multitier Reactive Programming". In: *IEEE/ACM International Workshop on Interoperability of Supercomputing and Cloud Technologies, SuperCompCloud@SC 2020, Atlanta, GA, USA, November 11, 2020.* 2020, pp. 27–32. DOI: 10.1109/SUPERCOMPCLOUD51944.2020.00010

[250] Sebastian Sztwiertnia, Maximilian Grübel, Amine Chouchane, Daniel Sokolowski, Krishna Narasimhan, and Mira Mezini. "Impact of Programming Languages on Machine Learning Bugs". In: *AISTA 2021: Proceedings of the 1st ACM International Workshop on AI and Software Testing/Analysis, Virtual Event, Denmark, 12 July 2021.* 2021, pp. 9–12. DOI: 10.1145/3464968.3468408

[245] David Spielmann, Daniel Sokolowski, and Guido Salvaneschi. "Extensible Testing for Infrastructure as Code". In: *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH 2023, Cascais, Portugal, October 22-27, 2023.* 2023, pp. 58–60. DOI: 10.1145/3618305.3623607

## 1.5 Overview

The rest of this dissertation is structured as follows.

Chapter 2 summarizes core concepts and related research. We introduce a categorization for IaC approaches, our model of declarative IaC, and conceptualize existing PL-IaC solutions, focusing on deployment state evolution and available testing techniques. Then, we summarize previous research on IaC and other fields related to this dissertation.

Chapter 3 assesses the state of dependencies between and coordination of deployments in practice. We performed the Dependencies in DevOps Survey 2021 with 134 IT professionals and analyzed its results. The chapter describes the survey instrument and its construction and evaluation, the results, analysis, and threats to validity. We found that dependencies across deployments are common and require coordination, which is often manual, even though automated approaches promise better SDO performance.

Chapter 4 discusses coordinating deployments in decentralized organizations. We developed $\mu$S, a PL-IaC solution that enables expressing and safely automating coordination across separate IaC programs. $\mu$S introduces abstractions to express interfaces to other deployments in IaC programs. It safely automates coordination based on them, presenting a novel PL-IaC runtime for long-running IaC programs that react to external signals. The chapter motivates, discusses, and evaluates $\mu$S' design and implementation. We found that $\mu$S effectively automates coordination in decentralized setup, introduces negligible overhead, and can be applied to existing decentralized deployments.

Chapter 5 makes safe DSU accessible to decentralized organizations through IaC programs. We developed the novel safe DSU approach Essential Safety, a unified model, and a decentralized dissemination algorithm that, together with IaC solutions with support for coordination, e.g., $\mu$S, enables safe DSU in decentralized organizations. The chapter motivates, discusses, and evaluates our contributions. We found that safe DSU can practically be applied to workflows, that Essential Safety reduces performance impact compared to previous safe DSU approaches, especially in the presence of non-essential changes, and that non-essential changes are common in real systems.

Chapter 6 enables empirical studies on PL-IaC. We built PIPr, an open-source dataset of all 37 712 public IaC programs we found on GitHub in August 2022, and performed initial analyses. The chapter describes the dataset's motivation, construction, dissemination, threats to validity, and our initial analyses, including their results. We found that AWS CDK is the most popular IaC solution followed by Pulumi, TypeScript and Python are the most used programming languages, and that most projects do not implement tests; only 25 %, dropping to 1 % for Pulumi, the only general PL-IaC solution.

Chapter 7 advances testing IaC programs. We developed ACT, an extensible method for automating unit testing IaC programs. We implemented it for Pulumi TypeScript in ProTI, enabling efficient IaC program testing, often without writing any testing code. The chapter identifies why IaC program developers do not write tests and describes, discusses, and evaluates ACT and ProTI. We found that ProTI can find bugs effectively and quickly, applies to real-world IaC programs, and allows the integration of existing test generation and oracle tools through plugins.

Chapter 8 summarizes this dissertation and outlines future research directions. We see perspectives in further empirical studies, work on safely automating additional requirements with IaC, and advanced quality assurance techniques for IaC programs.

## Chapter 2

# Fundamental Concepts and Related Work[1]

This chapter introduces the IaC concepts this dissertation uses and summarizes related research. We categorize IaC approaches and introduce the core concepts of declarative IaC. Building upon these basic concepts, we introduce PL-IaC in depth, including the available solutions, their behavior and differences, and testing techniques because PL-IaC is the IaC approach we focus on throughout this dissertation. Lastly, we provide an overview of work related to our contributions in the remaining chapters.

Section 2.1 introduces the IaC concepts relevant to this dissertation, and Section 2.2 provides a detailed insight into PL-IaC. After the concepts and technology, we summarize related work on IaC in Section 2.3 and other related research in Section 2.4.

## 2.1 Infrastructure as Code Concepts

In this section, we provide a high-level categorization for IaC approaches and introduce core concepts of declarative IaC, setting foundations for PL-IaC and beyond.

### 2.1.1 Categorization of Infrastructure as Code Approaches

Many IaC approaches have been proposed and implemented in recent decades. For orientation on how PL-IaC, the IaC approach we focus on in this dissertation, relates to other solutions, we now describe three axes of categorizations for IaC solutions.

**Code-centric vs. Model-driven IaC** *Code-centric* IaC approaches emphasize text-based user input, i.e., code, for defining deployment configurations. Within code-centric approaches, solutions can be distinguished by the supported languages, i.e., configuration

[1] Based on the authors' work in [225, 229, 233, 234]. [229] © 2024 IEEE.

languages (e.g., JSON and YAML), custom DSLs, or general-purpose programming languages. In contrast, *model-driven* approaches focus on receiving structured information, i.e., components and their relationships with properties. These are often represented in notations that are more conducive to graphical interfaces than text editing. Despite their differences, both approaches are technically complementary. For instance, modeling languages typically have a text-based notation, too, and tools for code-centric IaC may provide visual representations of the configuration expressed in the code.

**Provisioning-focused vs. Configuration-focused IaC** *Configuration-focused* IaC, or Configuration as Code (CaC), focuses on (re)configuring mutable infrastructure. Examples are CFEngine, Puppet, Chef, and Ansible, which are typically used for tasks like installing packages and managing files on existing servers. In contrast, *provisioning-focused* IaC solutions specialize in provisioning and managing immutable infrastructure, i.e., idiomatically, resources are created and only configured once, and updates replace them with a new resource. Examples are Pulumi, Terraform, AWS CloudFormation, and Azure Resource Manager, which typically manage volatile cloud infrastructure.

The distinction between these two groups, however, is not absolute. Provisioning-focused solutions may also mutate existing infrastructure under certain conditions during updates. Conversely, configuration-focused tools sometimes possess provisioning capabilities; for example, Ansible can provision VMs, too. Further, both kinds complement each other. For instance, a provisioning-focused tool like Pulumi may provision VMs, whose state a CaC tool like Chef then manages.

**Declarative vs. Imperative IaC** With *imperative* (also: *procedural*) IaC, developers describe the actions to perform, e.g., creating the file `/tmp/demo.txt`. In contrast, with *declarative* IaC approaches, developers describe the target state (also: desired state), e.g., the file `/tmp/demo.txt` exists. The IaC solution's deployment engine compares the target state with the infrastructure, and automatically derives and executes the required actions. If the file is missing, it is created; if it exists already, nothing is performed.

Declarative solutions often lead to more robust deployments [35, 72, 197], largely due to the flexibility provided by their deployment engines compensating environmental differences to a certain degree. In contrast, imperative solutions rely less on the deployment engine's capabilities and offer more direct control, for example, by executing bash scripts. However, achieving idempotency and convergence of infrastructure configuration—both highly desirable properties for reliable IaC—is generally more straightforward with declarative solutions. While imperative approaches can also attain these properties, they typically require more careful scripting and management.

These dimensions categorize IaC solutions; however, the practical solutions typically have nuances blurring their assignment. For instance, CFEngine and Ansible are configuration-focused and code-centric, but whether they are imperative or declarative is debatable. CFEngine is, at its core, declarative in defining the desired state, but imperative scripts are required to achieve it. Ansible's playbooks are declarative but have imperative features, e.g., users explicitly provide the order in which tasks are executed. In this dissertation, we focus on PL-IaC, which is code-centric, provisioning-focused, and declarative, supporting IaC programs written in general-purpose programming languages.

### 2.1.2 Abstractions of Declarative Infrastructure as Code

We now introduce abstractions for the core concepts of declarative IaC, where developers specify the target state of the deployment, not how to achieve it. Across all common declarative IaC solutions, the data structure describing deployment target states is similar, which is confirmed by Wurster et al. [262], who consolidated the representations in the Essential Deployment Metamodel (EDMM). In this dissertation, we describe a deployment as a *resource graph*.

**Definition 2.1** (Resource Graph)**.** A *resource graph* $G = (R, A)$ is a directed acyclic graph (DAG) of the set of nodes $R$, representing resources, and the set of arcs $A \subseteq R \times R$, representing dependencies between resources. Each resource $r = (\alpha, \tau, C) \in R$ has a unique identifier $\alpha$, type $\tau$, and configuration $C^\tau = K^\tau \to V^\tau$, which is a key-value mapping of the resource's properties. A resource's type $\tau$ specifies the property keys $K^\tau$ and types of the values $V^\tau$.

Any deployment entity is a resource in the resource graph, e.g., networks, servers, container services, load balancers, or network security policies. The arcs between resources in the resource graph are *dependencies*.

**Definition 2.2** (Resource Dependencies)**.** Given a resource graph $G = (R, A)$, a resource $r \in R$ *depends on* resource $r' \in R$ if and only if there is a path $P \subseteq A$ from $r$ to $r'$, i.e., a sequence of arcs leading from $r$ to $r'$ in $G$. Given $r$ depends on $r'$, $r'$ is a *direct dependency* of $r$ if and only if $(r, r') \in A$ or an *indirect dependency* if and only if $(r, r') \notin A$.

Dependencies are transitive and constrain the order in which resources may be deployed and undeployed, often for *requires* or *hosted-by* relationships. If $r$ depends on $r'$, $r'$ must be deployed before $r$. Symmetrically, $r$ must be undeployed before $r'$ can be undeployed. For instance, if a container is deployed into a cluster, both are resources, and the container depends on the cluster because the cluster must exist before the container can be deployed, and the container must not exist anymore when the cluster is safely undeployed. The (un)deployment order is always decidable because the resource graphs must be acyclic. We ignore all relationships between resources that do not impose these constraints in the resource graph. Hence, we model dependencies to ensure *dependency availability* while a resource is deployed.

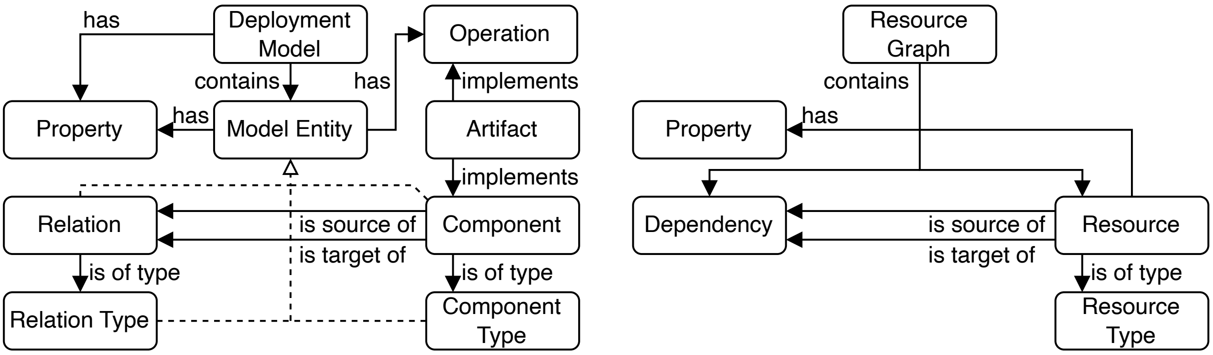
**Definition 2.3** (Dependency Availability)**.** Given a resource graph $G = (R, A)$, the dependencies of resource $r \in R$ are *available* if and only if all its resource dependencies $R' = \{r' \in R \mid r \text{ depends on } r'\}$ are deployed.

We already presented simple examples of resource graphs in Figure 1.1. However, realistic resource graphs can become quite big and precise from a user's perspective. To reduce complexity, *compound resources* can represent a subgraph of the resource graph as a single node, collapsing the subgraph in a simplified resource graph for developers.

**Definition 2.4** (Compound Resource)**.** Given a resource graph $G = (R, A)$, a *compound resource* $r_c \subseteq R$ provides an abstract view on $G$, collapsing all resources $r \in r_c$ into a single node $r_c$ in the simplified resource graph $G_S = (R_S, A_S)$. Its nodes are the compound resource $r_c$ and all resources in $R$ that are not in $r_c$, i.e., $R_S = \{r_c\} \cup R \setminus r_c$. Its arcs are all direct dependencies in $G$ not involving resources in $r_c$, and all dependencies involving a resource in $r_c$ on one side, replacing the member of $r_c$ with $r_c$, i.e., $A_S = \{(r, r') \in A \mid r \notin r_c \wedge r' \notin r_c\} \cup \{(r, r_c) \in R_S \times \{r_c\} \mid \exists r' \in r_c.\ (r, r') \in A\} \cup \{(r_c, r') \in \{r_c\} \times R_S \mid \exists r \in r_c.\ (r, r') \in A\}$. Multiple compound resources, e.g., $r_c$ and $r_c'$, can be applied in the same simplified resource graph if they do not overlap, i.e., $r_c \cap r_c' = \emptyset$.

Compound resources are high-level abstractions encapsulating common patterns, e.g., best practices, making it easier for developers to use them in the target state. For instance, users may use a compound resource for a web server, defining a single node in a simplified resource graph for which they only configure the IP address and port. Like this, they add an entire subgraph to the resource graph containing the required network, computing, and software resources with configuration for the web server.

Resource graphs, our core model of deployments in declarative IaC, simplifies the EDMM [262] (Figure 2.1a). We show the subset of the EDMM comprising resource



**(a)** Essential Deployment Metamodel.

**(b)** Resource graph model of this work.

**Figure 2.1:** Side-by-side comparison of the Essential Deployment Metamodel (EDMM) by Wurster et al. [262] with our resource graph model.

graphs with changed names in Figure 2.1b, using the same entity placement as in Figure 2.1a for easy side-by-side comparison. We call deployment models resource graphs, components resources, and relations dependencies, and the only entities with properties are resources. We only have a single dependency type expressing required availability (enforced dependency availability); therefore, we do not need dependency types in the model. Further, resource types are provided by the IaC solution and are not part of the resource graph. Finally, we do not need artifacts and operations. We assume that the IaC solution can perform the required operations, i.e., create, read, update, delete, and list (CRUDL), for each resource type it provides.

## 2.2 Programming Languages Infrastructure as Code

We now introduce PL-IaC, declarative IaC, where developers describe the deployments' target state in IaC programs written in general-purpose programming languages like TypeScript or Python. The available industrial-grade PL-IaC solutions are Pulumi [186], AWS CDK [8], and CDKTF [101]. Our examples focus on Pulumi TypeScript because Pulumi is the only established PL-IaC solution implementing general PL-IaC, while AWS CDK and CDKTF only implement a limited form, as described in Section 2.2.3. TypeScript is the most popular language for IaC programs (cf. Section 6.4.1).

We illustrate the high-level architecture of PL-IaC solutions in Figure 2.2. Developers implement an IaC program, which instructs the *deployment engine* about the target state of the deployment. The deployment engine knows the state of the infrastructure, i.e., the cloud, and performs the actions required to change the infrastructure towards the target state. The IaC program, in turn, receives information about the deployment's state from

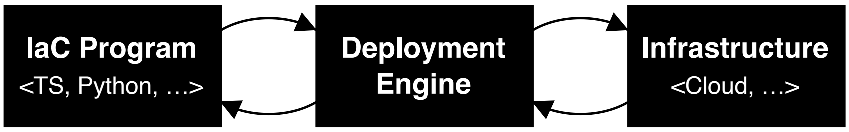


**Figure 2.2:** High-level architecture of PL-IaC solutions [229] © 2024 IEEE.

the deployment engine, including post-deployment information of resources, e.g., an ID or IP address assigned by the cloud on resource creation. The IaC program can use this post-deployment information to configure further resources in the target state.

Figure 2.3 shows the PL-IaC architecture in more detail. At the core of the IaC solution is the deployment engine, which maintains the current state of the deployments. Further, it controls the clouds, i.e., collections of resources controllable through a CRUDL API, allowing the creation, reading, updating, deletion and listing of resources. For each cloud, e.g., AWS, Azure, and Google Cloud, there is a provider that is typically developed by the community. Providers connect clouds with the deployment engine by providing a plugin implementing the cloud-specific control actions. Further, providers implement cloud-specific SDKs, offering the resource types of the clouds to IaC program developers. Developers implement IaC programs in general-purpose programming languages to deploy applications, importing the IaC solution's SDK for utility functions and the cloud-specific provider SDKs to define resources of the provider's resource types in the target state. Each execution of the IaC program defines a target state, which instructs the deployment engine. The deployment engine derives and executes the required actions to achieve the target state in the clouds and updates the current state accordingly.

Pulumi implements this architecture. Its users write IaC programs and run them using the Pulumi CLI. A deployment of an IaC program with the deployment's state is a *stack*, and multiple stacks can be instantiated from the same IaC program. The CLI transparently runs Pulumi's deployment engine and the IaC program in parallel, automatically installing and running the needed provider plugins distributed as *Pulumi packages* in a central registry. Pulumi's deployment engine maintains the current state of stacks and persists them across executions in the configured backend, which may be Pulumi's cloud service, cloud storage like AWS S3, or just a local file.

### 2.2.1 IaC Programs

With PL-IaC, developers write IaC programs to describe their deployment's target state, i.e., resource graph. Examples are Listings 1.1 to 1.3 and Listing 2.1, which is less

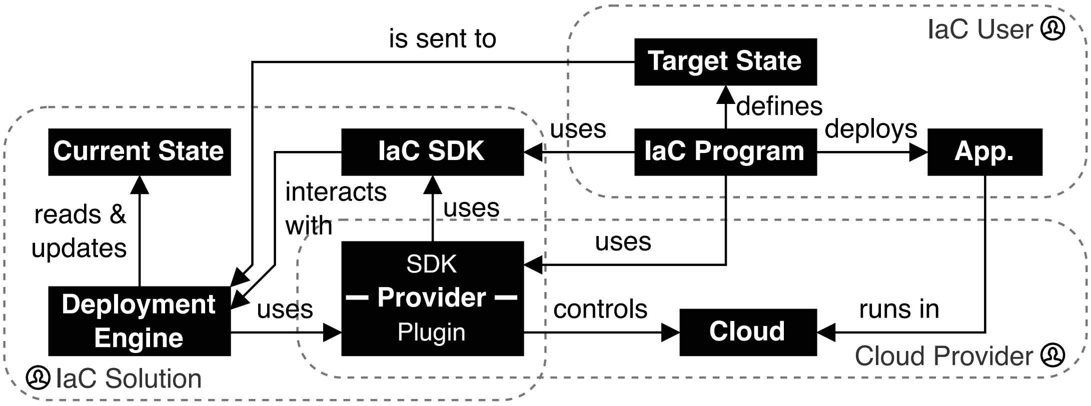


**Figure 2.3:** Entities and relationships in PL-IaC solutions [229] © 2024 IEEE.

realistic but demonstrates various features to define the resource graph. Developers can use all programming language features; however, the resource graph is only defined by the means we explain now and exemplify right after on Listing 2.1.

Providers export a class for each resource type they offer in their SDK. In IaC programs, developers use the SDKs and define a resource (i.e., a node in the resource graph) by instantiating an object of the resource type's class. The resource's *input configuration* is provided as an argument to the constructor. After the deployment engine deploys a resource, its post-deployment *output configuration* is available as properties on the resource's object. Developers explicitly define a dependency from a resource $r$ to a resource $r'$ (i.e., an arc from node $r$ to $r'$ in the resource graph) by referencing $r'$ or one of its output properties in the input configuration of $r$. Alternatively, such a dependency can be defined implicitly by instantiating $r$ in a program part that depends on an output property of $r'$. Defined resources, their properties, and dependencies are immutable. Thus, the target state grows monotonically throughout the IaC program execution. Defined resources and dependencies can neither be changed nor removed, ensuring declarativity.

The Pulumi TypeScript program in Listing 2.1 uses these features to define the resource graph in Figure 2.4. The program first imports Pulumi's SDK in Line 2.1.1 and the SDK of its provider for the AWS cloud Line 2.1.2. Provider SDKs export resource types as subclasses inheriting from Pulumi's `Resource` class. Listing 2.1 instantiates objects of the resource type for AWS S3 Buckets, `aws.s3.Bucket`, a class inheriting from Pulumi's `Resource`, in Lines 2.1.4 to 2.1.8, each adding a respective node to the resource graph. The buckets A and B have no dependencies, while D to E depend on bucket A. In Lines 2.1.6 and 2.1.7, these dependencies are explicitly defined by using the `dependsOn` option, which is standard for Pulumi resource type classes, and by using an

Listing 2.1: Examples of defining dependencies in a Pulumi TypeScript IaC program.

```
 2.1.1 import * as pulumi from "@pulumi/pulumi";
 2.1.2 import * as aws from "@pulumi/aws";
 2.1.3
 2.1.4 const bucketA = new aws.s3.Bucket("A");
 2.1.5 new aws.s3.Bucket("B");
 2.1.6 new aws.s3.Bucket("C", undefined, { dependsOn: bucketA });
 2.1.7 new aws.s3.Bucket("D", { versioning: bucketA.versioning });
 2.1.8 bucketA.id.apply(() => new aws.s3.Bucket("E"));
 2.1.9
2.1.10 export const id = bucketA.id;
```

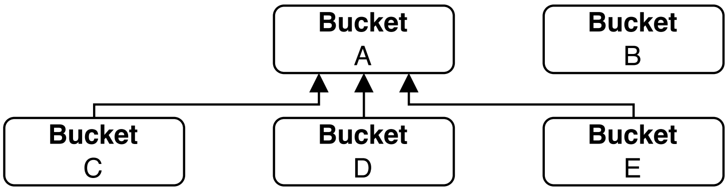


**Figure 2.4:** Resource graph of Listing 2.1.

output property of bucket A to configure an input property of bucket D. In Line 2.1.8, the dependency is implicitly defined by instantiating the resource's object in a code block that depends on an output property of bucket A; Pulumi even overlooks the dependency and misses it in its internal representation. However, for deployment, Pulumi still ensures the correct deployment order, i.e., dependency availability for bucket E, because it only executes the apply callback once the output property of bucket A is available, delaying the definition and deployment of bucket E until bucket A is deployed. Line 2.1.10 exports an output property of bucket A, making its value available for import in another deployment using Pulumi's stack references [187].

### 2.2.2 Deployment State Evolution

We described how IaC programs define deployments as resource graphs. Our informal explanations introduced (1) that the state evolves monotonically throughout the execution of an IaC program and (2) that there are multiple configurations for each resource. We formally model the deployment state and its evolution to enable a firm understanding of these facts. We also show how the model maps to the resource graph from Definition 2.1 during the execution of a PL-IaC program in Figure 2.5.

Like in Definition 2.1, a resource $r \in R$ has a unique identifier $\alpha$ and type $\tau$. However, instead of a single configuration, resources are configured in three stages: (1) the user's

| Resource | $r \;= (\alpha, \tau) \quad \in R$ |
|---|---|
| Input Configuration | $C_I^\tau = K_I^\tau \to V_I^\tau \in C_I$ |
| Target Configuration | $C_T^\tau = K_T^\tau \to V_T^\tau \in C_T$ |
| Output Configuration | $C_O^\tau = K_O^\tau \to V_O^\tau \in C_O$ |

| Deployment State | $D \;= (R_T, R_O, A)$ |
|---|---|
| Target State Configs. | $R_T = R \rightharpoonup C_T$ |
| Output State Configs. | $R_O = R \rightharpoonup C_O$ |
| Dependencies | $A \;\subseteq R \times R$ |

| IaC Program Trace | $T \;= (S, O) \triangleright T \mid \boxdot$ |
|---|---|
| Statements | $S \;= \mathbb{R}\big(\alpha, \tau, C_I^\tau\big) \mid \ldots$ |
| Value Origins | $O \;= V \to \mathscr{P}(R)$ |

| Auxiliary Functions | |
|---|---|
| `prepare :` | $C_I \to C_T$ |
| `register :` | $C_T \to C_O$ |

$$\frac{D \vdash T \longrightarrow D'}{D \vdash \ldots, O \triangleright T \longrightarrow D'} \;\textsc{Other} \qquad\qquad D \vdash \boxdot \longrightarrow D \;\textsc{End}$$

$$\frac{\begin{array}{c} D = (R_T, R_O, A) \qquad r = (\alpha, \tau) \qquad r \notin R_T \wedge r \notin R_O \qquad R'_T, R'_O, A' \vdash T \longrightarrow D' \\ C_T^\tau = \texttt{prepare}(C_I^\tau) \qquad C_O^\tau = \texttt{register}(C_T^\tau) \qquad R'_T = R_T \cup (r, C_T^\tau) \qquad R'_O = R_O \cup (r, C_O^\tau) \\ A' = A \cup \big\{ \big(r, r'\big) \mid k \in K_I^\tau \wedge v \in C_I^\tau(k) \wedge r' \in O(v) \big\} \qquad \forall k \in K_I^\tau . O(C_I^\tau(k)) \subseteq \mathrm{dom}(R_O) \end{array}}{D \vdash \mathbb{R}\,(\alpha, \tau, C_I^\tau)\,, O \triangleright T \longrightarrow D'} \;\textsc{ResDef}$$

**Figure 2.5:** PL-IaC deployment state evolution on an IaC program execution trace $T$.

input configuration in the IaC program $C_I^\tau$, (2) the derived target configuration $C_T^\tau$, and (3) the output configuration after deployment $C_O^\tau$. These configurations are mappings from keys $K$ to values $V$, where the set of valid keys and values is defined for each stage by the resource's type $\tau$. The deployment state $D$ is the triple $(R_T, R_O, A)$. $R_T$ and $R_O$ are partial mappings that assign resources $r \in R$ their respective target configuration $C_T^\tau$ and observed post-deployment output configurations $C_O^\tau$. $A$ is a relation defining the dependencies between resources in $D$, i.e., $(r, r') \in A$ means $r$ depends on $r$'. Hence, the deployment state is a resource graph $G$ with nodes $R$ and arcs $A$ with three configurations for each resource. $G$ with $C_I^\tau$ defines the user's input in the IaC program, $G$ with $C_T^\tau$ the target state, and $G$ with $C_O^\tau$ the post-deployment resource graph, respectively.

We model the execution of PL-IaC program as a trace $T$, a sequence of pairs $(S, O)$ terminated by $\boxdot$, where $S$ is a statement and $O$ a mapping that assigns each value $v \in V$ that may be used in a resource's configuration to the set of resources it depends on. Effectively, $O$ tracks information flow, e.g., $O(v) = \emptyset$ means $v$ is independent from any deployed resource, and $O(v') = \{r, r'\}$ means that $v$' is computed based on the state of the resources $r$ and $r$'.

The relation $\longrightarrow$ describes the evolution of the deployment state $D$ by denoting how a trace $T$ transfers $D$ to the new deployment state $D'$ as $D \vdash T \longrightarrow D'$. For the first trace pair, $D$ is empty, i.e., $(\emptyset \rightharpoonup C_T, \emptyset \rightharpoonup C_O, \emptyset)$, and $D'$ is the final deployment state. The only statement that changes the deployment state is $\mathbb{R}(\alpha, \tau, C_I^\tau)$, which defines a new resource with unique identifier $\alpha$, type $\tau$, and input configuration $C_I^\tau$. As defined by ResDef, $C_I^\tau$ is converted to the resource's target configuration $C_T^\tau$ using the auxiliary function `prepare`, which then is turned into the output resource configuration $C_O^\tau$ using `register`. Both derived configurations are added to the deployment state and the resource's dependencies, identified through information flow from another resource to a value in the input configuration. Any other statement type ('...') does not change the deployment state (cf. Other).

This model is expressive enough for IaC programs in general-purpose programming languages, including concurrency, because the deployment state evolution is commutative and monotonic as long as trace $T$ is valid, i.e., each resource is only defined once, after all resources its values depend on as captured by $O$. Further, definition and deployment of a resource are instant, i.e., in the same step, in our model as defined by ResDef. We present this simplification because the more realistic alternative does not provide additional insight and is trivial to achieve: adding the defined resource to the post-deployment state is moved from ResDef to a new rule, which adds a resource that is in the target state to the post-deployment state on a corresponding, to-introduce trace element that signals that the cloud confirmed the deployment of the resource.

The model naturally maps to Pulumi programs. A resource definition, e.g., `new aws.s3.Bucket("A")`, is a statement $\mathbb{R}(\alpha, \tau, C_I^\tau)$, in this example $\mathbb{R}(\mathrm{A}, \mathrm{aws.s3.Bucket}, \emptyset)$. All other statements are subsumed by '....' The resource class constructors implement `prepare` and invoke `register`, registering the new resource in Pulumi's deployment engine and receiving its post-deployment state, exposing it through output properties of the class. To detect the resource dependencies $A$, our model uses information flow captured in $O$. Pulumi tracks this information flow explicitly by wrapping observed resource state and derived values in future-like `Output` values. `Output` tracks the resources its wrapped value depends on. Yet, this only works if dependencies are explicitly defined, like in Lines 2.1.6 and 2.1.7. If there is implicit information flow, like for Line 2.1.8 through defining the resource in an `apply`-callback depending on another resource's output, Pulumi fails to capture the dependency. As explained in Section 2.2.1, such implicit dependencies are missing in Pulumi's representation, but dependency availability is enforced for them, which is why we capture them in our model.

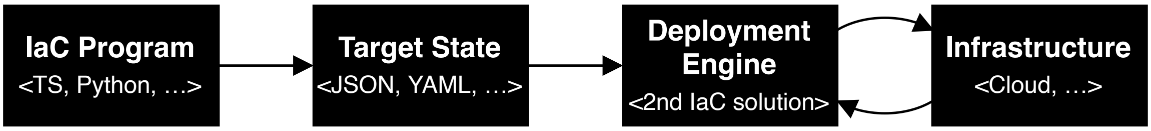


**Figure 2.6:** High-level architecture of two-phase PL-IaC solutions.

### 2.2.3 Limitations of Two-phase PL-IaC

To understand why we focus on PL-IaC solutions like Pulumi, we now describe the difference to AWS CDK and CDKTF, the only other industrial-grade PL-IaC solutions. Both only implement a limited form of PL-IaC, which we call *two-phase PL-IaC*.

With general PL-IaC solutions like Pulumi, IaC programs receive post-deployment state, i.e., output configuration of resources, and can process the values in the general-purpose language to configure further resources. In contrast, two-phase PL-IaC solutions like AWS CDK and CDKTF prohibit IaC programs from accessing the post-deployment state. Two-phase PL-IaC solutions execute the IaC program to generate the target state as a JSON file, which they provide to the deployment engine, i.e., AWS CloudFormation or Terraform, deploying it in a separate second step. This exchange is unidirectional, as visible in Figure 2.6, which, compared to the general PL-IaC architecture in Figure 2.2, prohibits information flow from the deployment engine to the IaC program.

Due to this approach, two-phase PL-IaC can only perform computation on post-deployment resource state that the deployment engine's DSL can express—practically limited to referencing values, string interpolation, and simple value processing. Yet, using an expressive general-purpose programming language to process the externally generated state is the reason for using general-purpose languages in IaC programs in the first place. Accordingly, two-phase PL-IaC only provides a subset of PL-IaC's capabilities. AWS CDK code can be embedded into Pulumi programs but not vice versa [109].

Our model for the deployment state evolution model (Section 2.2.2) covers two-phase PL-IaC, too. `register` is the identity function in this case: two-phase PL-IaC programs do not interact with the deployment engine during execution. Hence, each resource's target configuration $R_T$ is its post-deployment configuration $R_O$ in the IaC program.

### 2.2.4 Testing IaC Programs

Because software developers make mistakes, quality assurance techniques have been developed to detect and prevent errors before they get into production. We now provide an overview of the existing testing techniques for IaC programs.

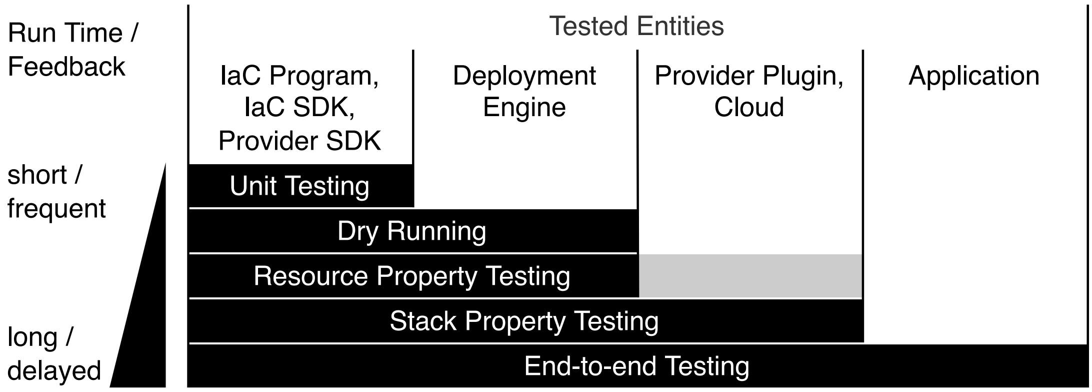


**Figure 2.7:** Pyramid of PL-IaC testing techniques. Covered entities, and typical relative run time and resulting feedback cycle frequency [229] © 2024 IEEE.

Figure 2.7 shows the PL-IaC testing techniques available for Pulumi programs [188], ordered top-to-bottom by time consumption. *Unit Testing* IaC programs is like in traditional software: IaC users run (parts of) the program with a unit testing framework, mock objects with side effects, i.e., every resource definition in an IaC program, and add checks. Even with runtime mocking—like supported by Pulumi—developers still have to provide the mocking logic. *Dry Running* simply executes the IaC program without executing deployment actions, providing a quick indication of whether the program terminates and a preview of the target state. Yet, the preview of dry running is incomplete, neither supporting specific checks nor ensuring sufficient coverage. Dry running does not execute code paths that depend on values available only after a resource was created. *Resource Property Testing* and *Stack Property Testing*, e.g., with CrossGuard [184], solve these issues by performing the deployment, making them integration testing techniques. They check resource configurations against policies before deployment and the final observed post-deployment state. *End-to-end Testing*, e.g., Pulumi's integration testing framework [182], runs the IaC program and validates the resulting deployment—not only its observed state.

Testing two-phase PL-IaC, i.e., for AWS CDK and CDKTF, is simpler than general PL-IaC and does not require mocking, as two-phase PL-IaC programs do not interact with the deployment engine. Instead, by design, they generate the *whole* target state before interacting with the deployment engine; at this point, it is simple to implement unit tests with assertions on the target state. Both CDKs' offer domain-specific assertions to facilitate implementing such checks. Additionally, both CDKs' unit testing offers snapshot testing, a regression testing technique where the generated target state is compared to the one of a previous execution, identifying changes and preventing unexpected updates. Lastly, CDKTF offers Terraform compatibility checks, which are needed because escape hatches in CDKTF programs can lead to a target state incompatible with its deployment engine, Terraform.

## 2.3 Research on Infrastructure as Code

We now provide an overview of research on IaC. Most works in the last years either focused on CaC, i.e., Puppet, Chef, and Ansible, summarized in Section 2.3.1, or model-driven IaC, especially in the TOSCA ecosystem, discussed in Section 2.3.2. Before these, we summarize the remaining publications, which are either more general or closer to PL-IaC because they address code-centric, provisioning-focused IaC.

**Surveys** Rahman et al. [194] performed a systematic mapping study on IaC and found that CaC tools are well-studied, but defects and security issues in applying IaC require more research. Guerriero et al. [97] investigated how practitioners adopt and develop IaC, tool support for IaC, and practitioners' requirements on IaC development, maintenance, and evolution. They identified a strong need for research on IaC testing and maintenance techniques. Chiari et al. [47] surveyed work on static analysis for IaC, almost exclusively finding work for CaC. Rahman et al. [198] analyzed 2 K Kubernetes manifests, finding more than 1 K misconfigurations. These studies identify a lack of research on quality assurance techniques for IaC, which we address in this dissertation.

Begoug et al. [26] studied more than 110 K posts on Stack Overflow about IaC, finding that file management and server configuration are the most discussed, while CI/CD pipelines and templating seem to be the most challenging concerns.

**Pre-deployment Analysis** Lepiller et al. [141] noticed that changing a declarative deployment from one safe configuration to another may exhibit unsafe intermediate configurations, called sniping vulnerabilities, which may be present only briefly. They proposed Häyhä, a tool using dataflow analysis to identify sniping vulnerabilities, and evaluated it on AWS CloudFormation templates. Cauli et al. [41] employed description logic modeling and inference to find security vulnerabilities in AWS CloudFormation templates. The approach was extended with a language for mutable actions to verify the impact of updating AWS CloudFormation templates before deploying them [42].

These advanced techniques precisely model and analyze the semantics of the cloud infrastructure services. This dissertation does not cover verification and cloud semantics; however, their techniques can be leveraged conceptually as oracles for ACT with ProTI.

### 2.3.1 Configuration-focused Approaches

We now summarize recent work on CaC, i.e., Puppet, Chef, and Ansible. The publications do not address decentralized coordination or safe DSU. Especially defect prediction, idempotency testing, and code smells have been studied; however, most insights do not apply to PL-IaC, and in the remaining cases, such generalization is questionable.

**Idempotency and Convergence Testing** Hummer et al. [113] proposed an idempotency testing approach for Chef scripts, which Ikeshita et al. [116] augmented with verification techniques to minimize the size of the required test suite. Shambaugh et al. [220] proposed Rehearsal to verify the determinacy and idempotency of Puppet scripts. Hanappi et al. [99] used transition graphs to assess whether Puppet scripts reliably converge.

IaC programs are idempotent and converge by design because they are declarative and deploy in one batch, which also applies to individual deployment rounds of $\mu$S.

**Code Smells** Sharma et al. [221] were the first to identify code smells in Puppet scripts, which later studies confirmed for Chef [217]. Rahman and Williams [201] identified source code properties correlating with defects in Puppet scripts, such as hard-coded strings. They further recognized security smells and proposed linters for Puppet, Ansible, and Chef [196, 197, 200]. Saavedra and Ferreira [207] introduced GLITCH for linters on a CaC-solution-agnostic intermediate representation. Reis et al. [204] found that such linters are too imprecise but can be improved through user feedback. Opdebeeck et al. [173] analyzed and proposed variable-precedence-related code smells in Ansible scripts. Rahman and Sharma [199] analyzed previously proposed code smells in Puppet scripts of Mozilla, Openstack, and Wikimedia and derived recommendations for practitioners. Finally, Opdebeeck et al. [172] applied program dependence graph analysis to Ansible scripts, motivating control- and data-flow analysis for IaC security smell detection techniques.

The precision of code smells is limited, i.e., false alarms are common, and the applicability and usefulness of these smells for IaC program developers are unknown.

**Defect Prediction and Error Detection** Dalla Palma et al. [58, 59, 60] proposed various quality metrics and an AI defect prediction framework for Ansible scripts. Quattrocchi and Tamburri [190] explored predictive maintenance of Ansible scripts, applying machine learning to a continuously evolving “fluid” dataset. Borovits et al. [32] proposed FindICI, an AI-based tool to identify linguistic inconsistency between documentation, comments, and code in Ansible scripts. Chen et al. [46] developed an approach to detect errors in

IaC scripts based on code feature models from historical commits and evaluated it on Puppet scripts. Bent et al. [30] proposed a model for the quality of Puppet IaC scripts, respective metrics, and a tool to measure it, evaluated with experts. Sotiropoulos et al. [243] developed a trace analysis to detect missing dependencies and notifiers in Puppet scripts, whose absence may cause them to be stale. Rahman et al. [192] proposed a taxonomy of eight defects in IaC scripts and developed a detection tool evaluated with Puppet scripts and developers, and Rahman and Parnin [195] used information flow analysis to track the propagation of security issues.

While these prediction and detection techniques could also be implemented for IaC programs, it is unclear whether current evaluation results generalize to them.

**Other Empirical Studies** Jiang and Adams [120] found in an early investigation on the co-evolution of Puppet and Chef IaC scripts in 265 OpenStack projects that IaC scripts are often changed, especially by testing staff. Rahman et al. [191, 193] performed a grey literature review to identify practices for secret management in CaC and identified development practices correlated with defects in Puppet scripts. These anti-patterns are unfocused contributions, contributions from developers in a silo or with a low amount of contributions in the project, as well as when too many people change a script and when the highest-volume contributor does not author all changes. Kumara et al. [134] surveyed CaC best practices in grey literature and compared them with findings in academic literature. Based on their Andromeda dataset [171], Opdebeeck et al. [174, 175] analyzed semantic versioning in the Ansible Galaxy ecosystem, finding that versions are inconsistently incremented. Hassan and Rahman [106] studied bugs in open-source Ansible test scripts.

It is unclear how these insights generalize to PL-IaC solutions and their ecosystems.

### 2.3.2 Model-driven Approaches

Modeling languages express a system's structure (e.g., components and relations) following a consistent set of codified rules. Various IaC modeling languages and related techniques have been developed, clearly focusing in the last years on TOSCA [168], an OASIS standard for modeling cloud applications and their management.

**TOSCA** The Topology and Orchestration Specification for Cloud Applications (TOSCA) describes application topologies as a graph of components and relationships (e.g., "hosted-on" or "connected-to"). Operational tasks are either declaratively derived from the topology or explicitly described as management plans using workflow languages like BPMN or BPEL. Bellendorf and Mann [27] provided a survey on cloud orchestration methodologies using TOSCA, its language extensions, and tools for manipulating TOSCA models. Wettinger et al. [260, 261] applied TOSCA to DevOps, using TOSCA as a metamodel to integrate heterogeneous automation artifacts. Various works automatically generate (BPMN) management workflows based on TOSCA models [38, 100]. The Essential Deployment Metamodel (EDMM) [262] is the least denominator metamodel of popular declarative deployment technologies to ensure that metamodel instances easily map to such technologies. TOSCA Light [264] is an EDMM-compliant subset of TOSCA, whose models can be deployed with 13 popular deployment technologies using the TOSCA Lightning [263] toolchain.

**Other Languages** Baresi et al. [25] developed KATENA, a deployment framework for blockchain applications based on TOSCA. Sandobalin et al. [214] compared the effectiveness of Argon [213], their model-driven IaC solution, with Ansible as a code-centric comparison in a user study with 67 computer science students. The results indicate model-driven solutions are more effective for novice users on simple deployments. However, it is unclear whether these results generalize to (1) realistic deployments, (2) widespread IaC modeling, i.e., TOSCA, and (3) experienced users.

This dissertation focuses on PL-IaC, which is not model-driven but code-centric and declarative, i.e., PL-IaC does not involve imperative operations (e.g., management workflows) in user input. The works above do not address decentralized coordination, safe DSU, or quality assurance techniques.

## 2.4 Further Related Research

In this section, we provide an overview of research that is not directly on IaC but is still related to and relevant to our contributions in this dissertation. We focus on work on system description and automation, updates and system changes, and software quality assurance techniques.

### 2.4.1 System Description and Automation

Describing a system's structure in a language and leveraging this information for automation is not unique to IaC and has been researched from various angles. We now summarize such research close to our work on describing and automating deployment in IaC: architecture description languages and resource orchestration.

**Architecture Description Languages** Architecture description languages (ADL) describe the component-level structure of an application. According to the classification of Medvidovic and Taylor [156], ADLs explicitly model components and their connection with the respective configurations, and they require tools for development and evolution. ArchJava [4] defines the component architecture of a system within the programming language. Components are special Java objects that define *ports*, i.e., the interfaces connecting components. In these interfaces, methods are *provided* and *required* to bind the provided method of a single connected port or *broadcasts*, binding the provided methods of multiple connected ports. The language enforces communication integrity. The ORS language and runtime [129] treat services as first-class composition units, separating the application from infrastructure concerns. It features a sub-language to define the deployment and to allow dynamic system changes. Terra and Valente [252, 253] proposed a domain-specific dependency constraint language to restrict structural dependencies in object-oriented software architectures. Acceptable and unacceptable dependencies are statically enforced to avoid architectural erosion.

ADLs can be used to verify that an application's architecture complies with its specification. Descriptive IaC solutions are similar to ADLs because they define the system's architecture as resources (components) and their dependencies (relations). However, ADLs do not provide an executable specification, constructing the system from the specification, which is required for deployments. Moreover, they do not cover mechanisms to coordinate decentralized deployments.

**Resource Orchestration** Weerasiri et al. [259] provided an overview of resource orchestration for the cloud. Ranjan et al. [202] summarized the programming of resource orchestration operations. Various centralized orchestration solutions for virtualized containers exist, e.g., Kubernetes, Kubernetes Federation, Mesos, and Docker Swarm [37]. DOCMA [121] is an orchestrator for IoT applications that is distributed and decentralized. However, the applications globally define all resources in their scope, requiring a centralized view of the system. COPE [145] is a distributed policy enforcement engine that enforces orchestration policies expressing constraints and service-level agreements.

Orchestrators manage resources, i.e., create, update, and delete them, which is also required in IaC. Declarative IaC is often used to configure orchestrators, which perform their deployment tasks. In contrast to current IaC, orchestrators can implement dynamic behavior, e.g., auto-scaling and fail-over. $\mu$S is an IaC solution implementing dynamic mechanisms to coordinate deployments. Generalizing $\mu$S' core idea to dynamic IaC, as we outline in Section 8.2, will enable developers to implement dynamic mechanisms of orchestrators in IaC programs, potentially even replacing orchestrators.

### 2.4.2 Updates and System Changes

Updating software is necessary to introduce new features and fix bugs and security issues. However, updating with low interruption and without causing errors is challenging, especially in bigger component-based systems. We now summarize related work close to our work on enabling safe DSU for workflows in decentralized organizations: DSU, dynamic code replacement, reconfiguration, and updating continuous delivery and workflows.

**DSU** Islam et al. [117] surveyed runtime software patching research and developed a taxonomy, providing a comprehensive overview. Older but specifically focused on DSU is the survey of Seifzadeh et al. [218]. Opaque-box safe DSU approaches [24, 128, 152, 256] only analyze the interaction of components to identify when it is safe to update a component, not the component's internals. In contrast, transparent box approaches [98, 247] leverage formal models of the programs to identify points in time when it is safe to update. While transparent-box methods allow more fine-grained analyses, they rely on strong assumptions on the implementation technology, making them hard to apply to distributed systems in practice, where components are implemented using heterogeneous paradigms, languages, and technologies.

Our work is closely related to the opaque-box approaches mentioned above. We compare our approach Essential Safety with them and enable their component-level application for workflows in decentralized organizations.

**Dynamic Code Replacement** Complementary to our work, there are approaches for updating running components. Updating a running program's code was investigated already in the 1970s [76]. Later, Erlang [18] was one of the first programming languages to enable *hot swapping*, i.e., modules can be replaced at run time (the new version is loaded when the next invocation occurs), and programmers can specify state transfer between modules. A similar solution for dynamic code replacement is also available in the Ada programming language [254]. More recently, dynamic code replacement on the Java Virtual Machine has been supported in Jvolve [249] and in the DCE VMs [265] as a modification to the Java HotSpot VM. These approaches focus on the technical realization of code replacement while the system is running and assume that developers correctly handle transferring the state of components across updates. Another line of work focuses on ensuring that state transformations are correct, e.g., using type systems [108]. Gu et al.

[96] replayed the sequence of invocations performed on the old object on the new one to ensure they reach the same state.

In this work, we focus on safely updating entire components, not parts of their code. While the entire system continues running, we stop and restart components.

**Reconfiguration** Dynamic software reconfiguration is about changing the configuration of a software product at run time while the system is operational. Researchers focused on reconfiguration models, ensuring the preservation of consistency properties and minimizing system disruption [31]. Adapta [210] is a reflective middleware for self-adaptive, component-based applications. It aims to decouple the application logic from the code that handles the adaptation, and it requires run-time monitoring and triggering mechanisms. Software reconfiguration has been applied to distributed execution, where remote system nodes interpret reconfiguration scripts [29]. Software self-adaptivity is a research line on switching the behavior of applications at run time, for example, using meta-programming or reconfiguration of component-based systems [155].

Coullon et al. [55] provided an overview of research on reconfiguration of component-based systems, focusing on verification. Concerto [44] is a model for analyzing and coordinating safe and efficient reconfiguration in deployments. It is inspired by Aeolus [62] and similar to Madeus [43], which only models deployments without reconfiguration.

We do not analyze reconfiguration and use a minimalistic lifecycle with only two states for resources, deployed and undeployed. Leveraging more precise lifecycles in IaC and safe DSU, e.g., by integrating existing reconfiguration models and formal approaches, promises performance improvements and less disruption without hampering safety.

**Updates in Continuous Delivery** Shahin et al. [219] systematically reviewed research on continuous delivery (CD), providing an overview. Lwakatare et al. [151] investigated CD implementations in five different development contexts. Laukkanen et al. [139] provided an overview of CD adoption problems and showed that most research focuses on issues of developers, but developers usually consider release and software update problems as external factors. Updating components in CD poses a problem in real-world settings, according to semi-structured interviews by Claps et al. [50] at Atlassian. At least 7 out of 10 interviews highlight that *seamless upgrades* are hard to implement in large systems and consume significant amounts of resources. Gallaba et al. [81] inferred dependencies between components using build execution tracing to accelerate CI/CD pipelines—information that could be used to identify non-essential changes.

CI/CD pipelines execute IaC scripts or programs to perform deployments. With $\mu$S, the CI/CD pipeline only updates the IaC program, which then performs changes *continuously* according to $\mu$S' safety protocol (Section 4.6). Generalizing solutions $\mu$S to dynamic IaC, which we describe as a future perspective in Section 8.2, will allow the implementation of complex CI/CD mechanisms directly in IaC programs, eliminating the boundary between dynamic mechanisms (CI/CD) and the deployment (IaC), enabling holistic analysis and reasoning for reliability aspects. In contrast to our contributions to safe DSU, CI/CD pipelines are typically independent of system monitoring, ignoring whether an update is performed in a safe time interval.

**Updating Workflows** Researchers have examined how existing workflows can be modeled to support changes while they are executed. Casati et al. [40] addressed the problem by defining a set of transformation rules and dividing the state space into parts terminated or handled by different process definitions. Geiger et al. [83, 84] presented a detailed review of the current state and evolution of BPMN 2.0 support and implementation, finding a lack of standard compliance in current implementations.

Our work on safe DSU focuses on updating components in workflows, not the workflows. We interpret workflow updates as updating the workflow engine component.

**DSU in Practice** Safe DSU relies on complex workarounds, avoiding the need for safe update intervals today. Cloud vendors and deployment platforms, e.g., Kubernetes [130], provide variations of blue-green [80] and canary [215] deployment strategies. Parallel change [216] is a pattern for safe interface updates that replaces unsafe changes with a sequence of safe ones. These solutions provide safe DSU for software where the components for the application logic are stateless and a (transactional) database holds the state. However, this hypothesis does not always apply, e.g., in workflows involving components belonging to various authorities. Using a central database is infeasible in such a case—a codified principle in microservice architectures [165].

In many scenarios, e.g., Web applications for social networks, it is accepted that updates may break multi-request transactions—retry is cheap, but it hampers user experience. In other scenarios, retry is not acceptable because it requires too much time or resources; therefore, safe DSU is needed to minimize the updates' impact.

Safe updating today, if applied at all, decomposes unsafe changes into sequences of safe changes, requiring discipline and introducing complexity to the systems. These strategies complement Essential Safety and other safe DSU approaches we discuss, which do not have these disadvantages but require monitoring and smarter orchestration.

### 2.4.3 Software Quality Assurance Techniques

Software quality, especially correctness, is a ubiquitous concern, and researchers have actively studied quality assurance techniques. We now summarize related work in the closest fields to our work on automated testing for IaC programs: automated testing, infrastructure verification, and automated mocking.

**Automated Testing** Fuzz testing (fuzzing) discovers software vulnerabilities, typically by treating the program as a closed box and testing it for hangs and crashes. Yet, input-value-generation-guided approaches exist; for example, grammar-based fuzzing is an active research field [110, 246, 257]. Li et al. [143] and Zeller et al. [266] provided an overview of state-of-the-art fuzzing techniques. Property-based testing (PBT) [49, 78] is a related approach, where code is exercised on randomly generated tests, and results are checked against invariants—the properties.

Various works investigate effective PBT test generators. Lampropoulos et al. proposed Luck, a language for PBT generators [135], and coverage-guided PBT [136], which mitigates PBT's inability to generate inputs for sparse preconditions. Löscher and Sagonas introduced targeted PBT [149] and automated it [148] using search-based techniques to guide the generation. Kuhn et al. [132] found that most bugs are caused by the interaction of only a few parameters, motivating combinatorial testing [133], which Goldstein et al. [87] applied to PBT generators by modifying the random generator distributions. On the intersection with formal methods, Paraskevopoulou et al. [178] integrated PBT into a proof assistant to verify tests, and Lampropoulos et al. [137] compiled logical conditions (inductive relations) to generators and to their soundness and completeness proofs. De Angelis et al. [61] leveraged symbolic execution and constraint logic programming to automatically derive generators.

Test oracles are imperative in automated testing to decide whether a generated test passes or fails. Differential testing compares test outputs across implementations, assuming different outputs indicate a bug [75]. Homomorphic testing oracles encode how an input-input relationship affects the output-output relationship [45], and intramorphic testing oracles, recently proposed by Rigger and Su [205], encode how a system-system relationship maps to the output-output relationship for fixed inputs. Jahangirova et al. [119] proposed an approach to iteratively improve imperfect oracles. Further, there are machine-learning-based approaches, e.g., Tsimpourlas et al. [255] used supervised learning on traces, Dinella et al. [64] used a transformer-based approach on existing unit tests, and Ibrahimzada et al. [114] learned oracles without ground-truth, assuming differences in the input-output correlation for correct and incorrect behavior.

ACT is fuzzing and PBT for PL-IaC programs. For ProTI, type-based generators and oracles, prototypes demonstrating third-party tool integration, and an ad-hoc specification syntax are available. The approaches above can be integrated or implemented in ProTI plugins to use them for IaC programs.

**Infrastructure Verification** Ensuring infrastructure correctness has been extensively studied. AWS investigated automatically verifying infrastructure properties [21, 33, 51], leading to at least two automated services in production: AWS Tiros verifies reachability queries on virtual networks [19], and AWS Zelkova performs access verification on role-based AWS IAM policies [20]. These solutions verify already deployed setups, but their techniques should be applicable pre-deployment on IaC programs, which encode the infrastructure's configuration. Such pre-deployment infrastructure verification could also leverage more foundational techniques. For example, Alloy [118] is a language and analysis tool to verify the structural properties of software. Ahrens et al. [3] developed a proof system for invariants on reconfigurable distributed systems. Evangelidis et al. [74] proposed probabilistic verification of performance properties of rule-based auto-scaling policies. Lastly, Abu Jabal et al. [2] gave a comprehensive overview of techniques for policy verification, focused on access control and network management.

Program verification remains an open challenge, requiring significant manual effort or being limited to specific properties. Augmenting ACT with automated verification of domain-specific properties, e.g., network access constraints, is a promising direction, orthogonal to the automated testing contributions in this dissertation, but a promising direction for future research, as we outline in Section 8.2.

**Automated Mocking** In a study on mocking in open-source systems, Spadini et al. [244] found that developers mock components that are difficult to handle and that mocking code increases the coupling between system and test code, motivating mock synthesis. Taneja et al. [251] proposed MODA, using an efficient, SQL-aware mock and advanced test generation techniques to automatically test database applications. Solms and Marshall [242] automatically generated mocks from explicit component contracts. Various works synthesize mocks from interaction traces of components [77, 122, 209]. In contrast, Zhu et al.'s StubCoder [267] synthesizes mocks for regression testing solely from the tests' code, without running the mocked component.

Mocking resource definitions in IaC programs is trivial because PL-IaC solutions provide an interface to intercept them, eliminating the need for advanced mocking techniques. Yet, the mocks' test generation and validation logic are complex. ACT encapsulates them into plugins, enabling the integration of mocking techniques from literature into ProTI.

## Chapter 3

# Dependencies and Coordination Between Deployments[1]

In this chapter, we present the Dependencie in DevOps Survey 2021. The survey sheds light on dependencies between deployments in practice and related coordination across deployments, showing that application dependencies often constrain the order of deployment operations across teams. Such order is often coordinated manually, even though automation promises better SDO performance. This highlights a research gap in available approaches for automated deployment coordination, which we address in Chapter 4.

We motivate and outline the survey's goals in Section 3.1, describe its design in Section 3.2, and the execution in Section 3.3. Section 3.4 presents the results that we analyze in Section 3.5 before discussing the core insights in Section 3.6. Section 3.7 describes the study's threats to validity, and Section 3.8 concludes.

## 3.1 Motivation and Research Questions

DevOps encourages cross-functional teams, which contrasts the previous silo-based approach with separate teams for, e.g., development, operations, and testing [70]. Such teams combine competence, interest, and responsibility for a single application, preventing unclear, shared responsibilities across them. An application is jointly designed, developed, tested, and operated within a cross-functional team, reducing friction and enabling fast feedback between activities.

Building teams around applications—not roles—aims at team independence in a decentralized organization. Applications, however, are usually not fully isolated but interact with other applications on which they depend. For example, in a microservices webshop, an order service may require an authentication service to verify the user's

[1] Based on the authors' work in [233, 235, 236]. [235] © 2023 IEEE.

permission to view or place orders. If different teams maintain and operate these services, we wondered whether and how their operations are decoupled. Do such dependencies constrain the order of the applications' (un)deployments? If so, how do the responsible teams maintain their independence as DevOps prescribes? Based on these uncertainties, we formulated the following research questions.

**RQ 3.1** How many dependencies on other applications do applications have in practice?

**RQ 3.2** Do inter-application dependencies constrain the order of their (un)deployment?

**RQ 3.3** How is the order of deployments coordinated across teams in organizations?

**RQ 3.4** Do practitioners believe that automated coordination of deployments promises better SDO performance than manual coordination?

**RQ 3.5** Does the organization's SDO performance influence the answers to RQ 3.1, RQ 3.2, RQ 3.3, and RQ 3.4?

**RQ 3.6** Do the demographics, i.e., experience, department, company size, region, and industry, influence the observed SDO performance or answers to RQ 3.1, RQ 3.2, RQ 3.3, and RQ 3.4?

To answer these questions and understand the issues related to deployment coordination, we conducted a cross-sectional, self-administered, online questionnaire survey with 134 IT professionals about software dependencies in their organization. While various empirical studies on the application of DevOps in practice exist [36, 57, 70, 79, 151], the state of application dependencies and their impact on the order of deployments have not been assessed yet. Such insight indicates whether teams may deploy their applications independently or whether they need to coordinate. Further, in case coordination is required, we do not yet have insight into how such coordination is accomplished. The raw data and a technical report are published [236] and licensed under CC BY 4.0.

## 3.2 Survey Design

To answer our research questions, we chose a cross-sectional, self-administered, online questionnaire as the survey instrument because it is suitable for collecting quantitative statistics from a large population sample [127]. We refined the research questions to the research hypotheses shown in Table 3.1 on the variables in Table 3.2. The first two numbers in the hypotheses' labels match the research questions they belong to, e.g., RH 3.3.2 relates to RQ 3.3.

**Table 3.1:** Research hypotheses of the Dependencies in DevOps Survey 2021.

| | Hypothesis Variables: | Independent | Dependent |
|---|---|---|---|
| RH 3.1.1 | The majority of primary applications depend on other applications. | V 3.1 | |
| RH 3.2.1.1 | The majority states dependencies may imply deployment order. | V 3.2 | |
| RH 3.2.1.2 | The majority states dependencies may imply undeployment order. | V 3.3 | |
| RH 3.2.2.1 | The agreement that dependencies imply deployment order positively correlates with the number of dependencies. | V 3.1 | V 3.2 |
| RH 3.2.2.2 | The agreement that dependencies imply undeployment order positively correlates with the number of dependencies. | V 3.1 | V 3.3 |
| RH 3.2.3 | The agreement that dependencies imply deployment order positively correlates with the agreement that dependencies imply undeployment order. | V 3.2 | V 3.3 |
| RH 3.3.1 | The majority relies on manual coordination to enforce deployment orders. | V 3.4 | |
| RH 3.3.2 | The agreement that dependencies imply deployment order positively correlates with reliance on manual coordination. | V 3.2 | V 3.4 |
| RH 3.4.1 | The majority believes manual coordination does not promise better SDO performance than when no coordination is required. | V 3.5 | |
| RH 3.4.2 | The majority believes automated coordination does not promise better SDO performance than when no coordination is required. | V 3.6 | |
| RH 3.4.3 | The majority believes automated coordination promises better SDO performance than manual coordination. | V 3.5,<br>V 3.6 | |
| RH 3.5.(1,…,5) | The SDO performance correlates (with the number of dependencies, negatively with the agreement that dependencies imply deployment order, negatively with the agreement that dependencies imply undeployment order, positively with the use of automated coordination, positively with the agreement that automated coordination promises better SDO performance than manual coordination). | V 3.7 | (V 3.1,<br>…,<br>V 3.4,<br>{V 3.5,<br>V 3.6}) |
| RH 3.6.(1,3).X<br>$X = (1,\dots,6)$ | The (experience, company size) correlates with (the number of dependencies, agreement that dependencies imply deployment order, agreement that dependencies imply undeployment order, order enforcement approach, agreement that automated coordination promises better SDO performance than manual coordination, SDO performance). | (V 3.8,<br>V 3.10) | (V 3.1,<br>…,<br>V 3.4,<br>{V 3.5,<br>V 3.6},<br>V 3.7) |
| RH 3.6.(2,4,5).X<br>$X = (1,\dots,6)$ | The (department, region, industry) influences (the number of dependencies, agreement that dependencies imply deployment order, agreement that dependencies imply undeployment order, order enforcement approach, agreement that automated coordination promises better SDO performance than manual coordination, SDO performance). | (V 3.9,<br>V 3.11,<br>V 3.12) | (V 3.1,<br>…,<br>V 3.4,<br>{V 3.5,<br>V 3.6},<br>V 3.7) |

**Table 3.2:** Variables of the Dependencies in DevOps Survey 2021.

| | Description | Survey Questions |
|---|---|---|
| V 3.1 | Number of application dependencies. | SQ A.2.1 |
| V 3.2 | Agreement that dependencies imply deployment order. | SQ A.2.2 |
| V 3.3 | Agreement that dependencies imply undeployment order. | SQ A.2.3 |
| V 3.4 | Deployment order enforcement approach. | SQ A.2.4 and SQ A.2.5 |
| V 3.5 | Agreement that manual coordination promises better SDO performance than when no coordination is required. | SQ A.3.1.1, SQ A.3.2.1, SQ A.3.3.1 and SQ A.3.4.1 |
| V 3.6 | Agreement that automated coordination promises better SDO performance than when no coordination is required. | SQ A.3.1.2, SQ A.3.2.2, SQ A.3.3.2 and SQ A.3.4.2 |
| V 3.7 | The organization's SDO performance. | SQ A.1.1 to SQ A.1.4 (SQ A.1) |
| V 3.8 | Years of professional experience. | SQ A.4.1 |
| V 3.9 | Department. | SQ A.4.2 |
| V 3.10 | Company size. | SQ A.4.3 |
| V 3.11 | Region. | SQ A.4.4 |
| V 3.12 | Industry. | SQ A.4.5 |

The target audience of the questionnaire is IT professionals with knowledge about the deployment and the dependencies of the application they work on. Participants (1) had to work in a company or company-like environment, e.g., an IT department at a university or freelancing, (2) their company had to develop and/or operate software, and (3) they must not have been only a *user* of this software. The size of this population can only be estimated, and it is roughly 55.3 million IT professionals worldwide [115].

### 3.2.1 Instrument Design

Based on the research questions and hypotheses, we designed the questionnaire in Appendix A and implemented it in Google Forms [90]. In the survey, the participants were asked to answer all questions in the context of the *primary application* they were working on. This ensures clarity if participants are involved with multiple applications and that participants answer the questions for relevant applications—assuming that applications to which more time is dedicated have higher relevance, i.e., they are no toy projects. The term primary applications had been used in other DevOps surveys to frame such questions before, e.g., by Forsgren et al. [79].

The first section of the survey measures the organization's SDO performance (V 3.7). We reuse the measurement instrument by Forsgren et al. [79] (SQ A.1.1 to SQ A.1.4), showing how well the DevOps goals are achieved. It comprises (1) deployment frequency, (2) lead time for changes (delay between development and production), (3) time to restore service on failure, and (4) change fail rate. To the best of our knowledge, it is the furthest developed instrument to measure the SDO performance of an organization, and it was validated over multiple years in the DORA DevOps reports.

The second section assesses the practices in the participant's environment, i.e., the presence of inter-application dependencies (V 3.1), whether such dependencies constrain (un)deployment order (V 3.2 and V 3.3), and how they (would) coordinate deployment order (V 3.4). For V 3.1, SQ A.2.1 uses a discrete answer scale instead of a continuous numerical input to make answering easier for the participants. While the answers are less precise, they still convey the relevant magnitude of the number of dependencies. For V 3.2 and V 3.3, SQ A.2.2 and SQ A.2.3 use standard agreement Likert scales. For V 3.4, we chose a very reduced set of options in SQ A.2.4, containing only the two extremes and one step in between, as even this simplification already required the respondents to think about the question. To keep the chance to learn and discover solutions used by more motivated respondents, we added the optional open-ended question SQ A.2.5, allowing respondents to skip the question to prevent abortion due to being overstrained.

The third section of the questionnaire is about the IT professional's beliefs about how manual and automated coordination impact the core SDO performance metrics compared to no need for coordination. To measure V 3.5 and V 3.6, we altered the validated instrument for SDO performance assessment of Forsgren et al. [79]. SQ A.3.1 to SQ A.3.4 present standard Likert scales, assessing the respondents' belief that the two extremes (manual coordination and ideal automated coordination) improve or worsen the respective SDO performance metric compared to a scenario where no coordination is required. To ease the introduction for the respondents, the question wording is very close to SQ A.1.1 to SQ A.1.4, which they already know. Moreover, the scenario is provided in textual form as well as in a side-by-side comparing graphic (SI A.3). We chose no coordination as a baseline because it represents the ideal situation from an SDO performance perspective, and all sorts of coordination are expected to be at most that performant. Accordingly, an unjustified belief in automation can be identified.

The last section records the demographics of the respondents, i.e., SQ A.4.1 to SQ A.4.5 directly measure V 3.8 to V 3.12. The choices are taken from Forsgren et al. [79] and, for the ordinal questions, presented in descending order proportional to the number of answers in their survey. This way, we reused their years of survey experience, and our survey targeted the same audience as theirs did. Finally, we followed the best practice of allowing respondents to comment on the study after filling it out (SQ A.4.6).

The questions are ordered according to Bourque and Fielder [34]: SQ A.2.1 to SQ A.3.4—the survey's core questions—are logically ordered with increasing complexity. We noticed early that it was easy for participants to answer the SDO performance questions SQ A.1.1 to SQ A.1.4 about why they were at the beginning of the survey. The demographic questions (SQ A.4.1 to SQ A.4.6) are located at the end.

### 3.2.2 Instrument Evaluation

We evaluated the questionnaire in two focus groups, moderated discussion groups where all participants conducted the survey and then discussed its objectives, clarity, and required improvements [127]. The first meeting was between two authors and six researchers from the Software Technology Group at the Technical University of Darmstadt, who are experts in (empirical) research, software development, and engineering. However, they did not belong to the target audience. Based on the feedback, we added clarifications and improved questions, redesigned SQ A.3.1 to SQ A.3.4, and changed SQ A.2.4 and SQ A.2.5 to be closer to the research objective.

We presented the updated version to the second focus group, comprised of one author and five persons within the target audience: four software developers of various experience levels and one manager from the University of St. Gallen IT service provider. They developed and operated a heterogeneous landscape of IT systems and were transitioning to a DevOps organization at the time of the meeting. Based on the feedback, only minor clarifications and improvements, e.g., adding the figure in SI A.3, were performed as the participants understood and answered the questionnaire as intended, even when they individually faced doubts and ambiguities in some places.

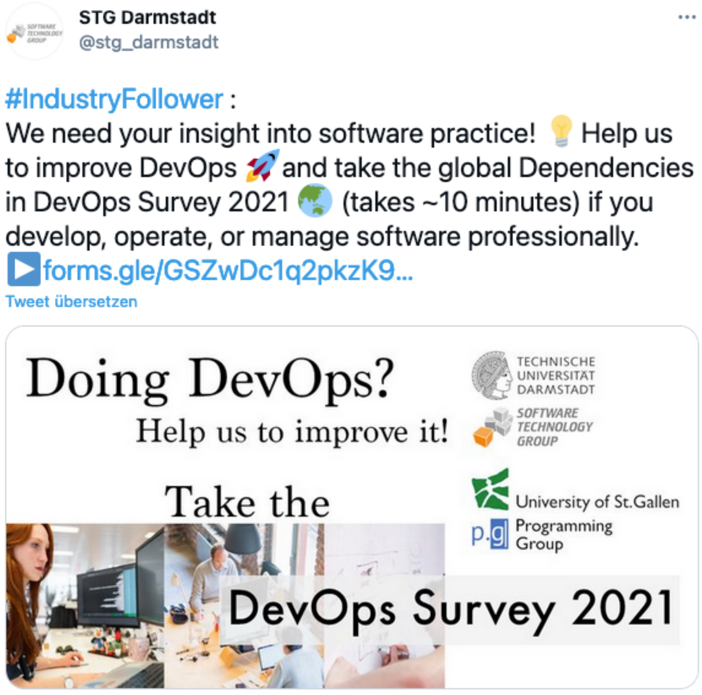
As a last evaluation step, we started advertising the survey slowly on only a few distribution channels, treating it as a pilot study [123, 127]. After one week and the first 30 responses, we assessed the data collected and comments received from participants. No need for changes was identified during this pilot phase.

## 3.3 Execution

The whole population of IT professionals in the target audience cannot be determined. Hence, we could only use non-probabilistic sampling methods [123]. Participants were advertised via social media, mailing lists, and personal contacts using snowball sampling [89], where participants were asked to invite additional participants from their

**Table 3.3:** Distribution channels of the Dependencies in DevOps Survey 2021.

| Channel | Responses |
|---|---|
| DevOps Chat Slack | 3 |
| Devops Weekly Newsletter | 24 |
| Facebook | 3 |
| LinkedIn | 7 |
| Reddit | 2 |
| Twitter | 6 |
| SB1 | 27 |
| SB2 | 16 |
| SB3 | 20 |
| SB4 | 14 |
| SB5 | 12 |
| *Total* | 134 |



STG Darmstadt
@stg_darmstadt

#IndustryFollower :
We need your insight into software practice! 💡 Help us to improve DevOps 🚀 and take the global Dependencies in DevOps Survey 2021 🌏 (takes ~10 minutes) if you develop, operate, or manage software professionally.
▶️forms.gle/GSZwDc1q2pkzK9...
Tweet übersetzen

**Figure 3.1:** Advertisement for the Dependencies in DevOps Survey 2021 on Twitter.

network. The resulting sample cannot be assessed for representativity due to the unknown properties of the whole population.

The questionnaire was advertised starting at the end of January 2021 for 12 weeks via social media, a mailing list, and our networks. Table 3.3 shows the distribution channels and number of responses. Advertisements on social media were usually supported with a graphic and the sentence, “We need your insight into software practice! Help us to improve DevOps and take the global Dependencies in DevOps Survey 2021 (takes 10 minutes) if you develop, operate, or manage software professionally”. For instance, Figure 3.1 shows an advertisement on Twitter. Every snowball (SB) personally advertised participants. SB 1 and SB 2 are personal contacts of two survey authors, SB 3 is all personal advertisement by colleagues of the authors, and participants of SB 4 and SB 5 were advertised by two friends of the authors who work in the industry in positions spanning multiple companies and contexts and performed advertisement in their network. In total, we received responses from 134 participants, of which 66 % were found through personal advertisement via snowball sampling in the authors' network, 18 % through a DevOps newsletter, and the remaining 16 % via social media. For all channels, the response rates are unknown and assumed to be very low for social media, rather low for newsletters, and the highest for personal advertisements.

We used a separate copy of the same Google Forms form for every distribution channel, each storing responses in a separate Google Sheets [91] spreadsheet and enabling us to identify through which channel a response was submitted. During the campaign, we used this information to assess the response rate of individual actions to improve the advertisement and focus on distribution channels with better response rates. This setup and procedure was presented to the central data privacy team of the Technical University of Darmstadt, which confirmed its legal compliance, i.e., with the GDPR. The imprint and data privacy policy were hosted on the private website of one of the authors.

We downloaded the responses for each channel and augmented them with their channel ID. We then manually analyzed the data of each channel for anomalies or spam attacks. In this step, the only anomaly we noticed was that, in total, 3 responses were duplicated, each with multiple hours between the two submissions. We assumed that these are accidental resubmissions, which could occur in some web browsers when a user reopens the tab of a submitted Google Forms form. We removed the duplicate responses, merged all left-over responses into a single dataset, and dropped the responses' submission time.

We coded the responses using the codings in Appendix B, introducing a new field for each question with the suffix "Code." We introduced the fields SQ A.3.1D, SQ A.3.2D, SQ A.3.3D, and SQ A.3.4D, which are the differences between the subquestions' codings of SQ A.3.1 to SQ A.3.4, e.g., SQ A.3.1D = SQ A.3.1.2Code − SQ A.3.1.1Code. Positive values indicate that the respondent assumes that automated coordination performs better compared to manual coordination regarding the given SDO metric. The additional field SQ A.3D is the sum of SQ A.3.1D to SQ A.3.4D. Additionally, we applied the SDO performance clustering from Forsgren et al. [79] in a new field SQ A.1. We assigned each response to the cluster with the mean that has the minimal Euclidean distance to the respondent's answers to SQ A.1.1 to SQ A.1.4. The resulting dataset was exported in CSV format and is published as attachment of the study's technical report [236].

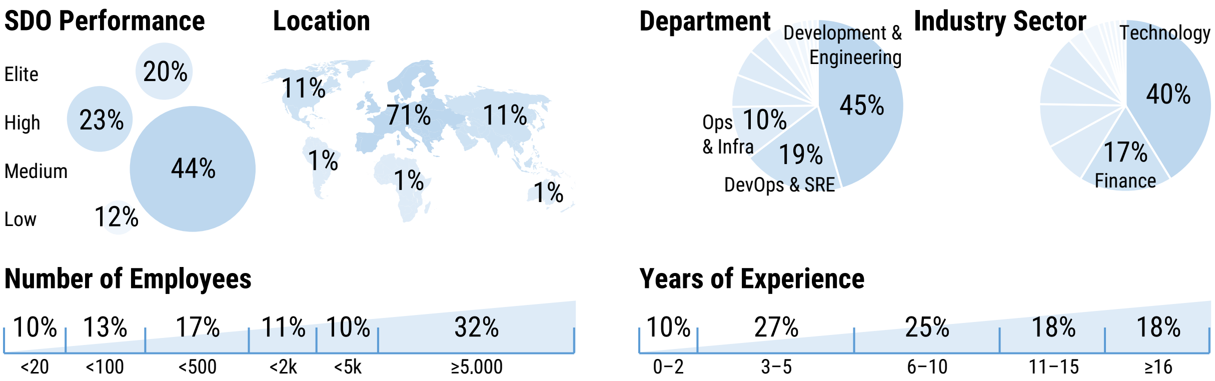


**Figure 3.2:** Respondent demographics of the Dependencies in DevOps Survey 2021 [235] © 2023 IEEE.

## 3.4 Results

We received 134 individual responses and present the results and selected plots; the raw results and detailed plots can be found in the technical report [236].

Figure 3.2 summarizes the respondents' and their organizations' demographics. Most participants were located in Europe and had software development or operations tasks. We reached respondents with various professional experiences in organizations that cover a broad spectrum of industry sectors and sizes. The organizations had varying SDO performance. 44 % had medium SDO performance, the biggest group, each one-fifth had high and elite performance, and the remaining 12 % had low SDO performance.

Figure 3.3a shows that for 89 % of the respondents, the primary application required another application to provide full functionality; two to five dependencies were very common. For a fifth of the respondents, the primary application required more than ten other applications. 87 % of the participants stated, to different extents, that such dependencies may constrain the order of the applications' deployments (Figure 3.3b). For undeployment, fewer respondents agreed, yet more than two-thirds confirmed that dependencies may constrain the order of undeployment. Accordingly, dependencies among applications are ubiquitous and often constrain the order in which applications can be safely (un)deployed.

Figure 3.3c illustrates that only one-fourth of the respondents coordinated the order of (un)deployment operations in a fully automated way; 76 % relied on manual coordination, e.g., via phone, chat, and mail. Through the free text field, some respondents provided the specific tools they used, which we summarize in Table 3.4. Further, one person mentioned that they avoid dependencies requiring coordination, another stated that their applications

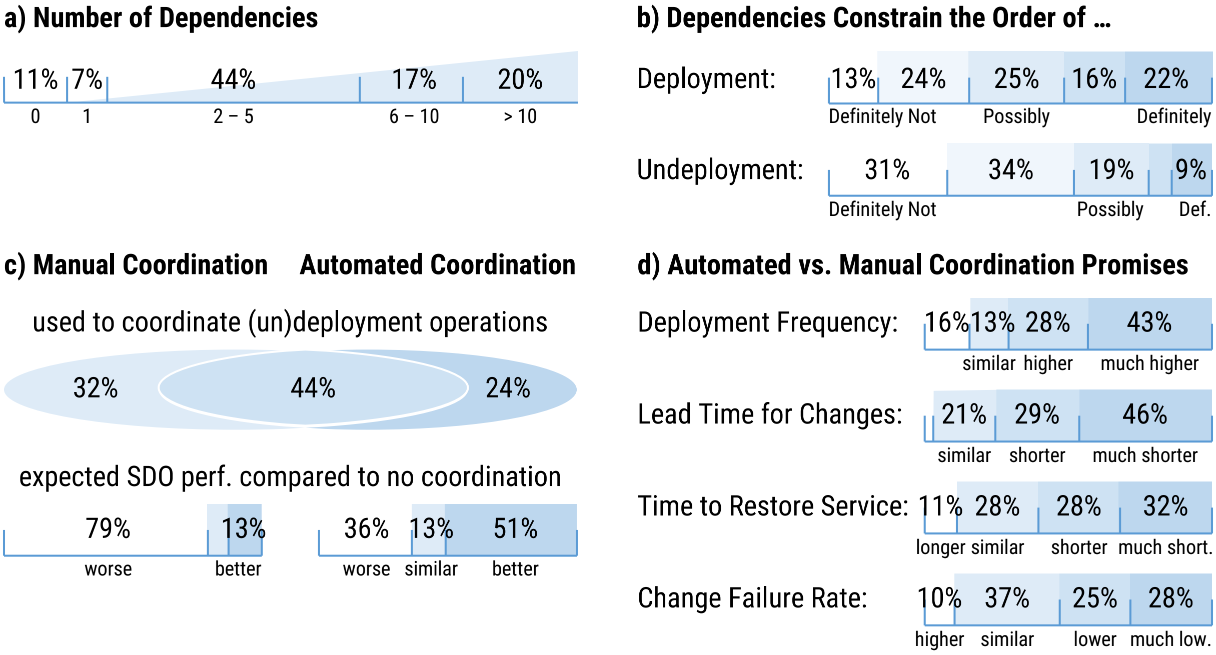


**Figure 3.3:** Quantitative results of the Dependencies in DevOps Survey 2021 [235] © 2023 IEEE.

**Table 3.4:** Tools IT professionals use to coordinate deployment orders.

| Category (# mentions) | Concrete Tools (# mentions) |
|---|---|
| CI/CD (17) | Jenkins (5), GitLab (4), Azure DevOps (3), Bamboo (2), ServiceNow (1), TeamCity (1), other (1) |
| Calling (2) | Jabber (1), other (1) |
| IaC (4) | Ansible (3), Puppet (1) |
| Messaging (11) | Microsoft Teams (3), Slack (3), Mattermost (2), other chat (2), email (1) |
| Other (16) | Jira (5), custom solution (4), Helm (2), Rundeck (2), feature toggles (1), Git (1), native deployment kits (1) |

do not require coordination, and a third one noted that they delegate deployments with coordination needs to more senior developers.

Most (79 %) believed manual coordination yields worse SDO performance than no coordination, while 51 % expected better SDO performance with automated coordination. In Figure 3.3d, we show, for each SDO performance metric, whether automated coordination was expected to yield better performance than manual coordination. For each metric, the majority of respondents expect better performance.

A few respondents shared insights in the final free text field. One person explained that their project started addressing DevOps maturity after more than one year and was interested in knowing how to achieve DevOps maturity earlier and whether it is desirable.

Another respondent stated that not every organization wants and is ready for continuous delivery. It was mentioned that some organizations may have continuous delivery for staging environments but still require manual intervention for production deployments. Lastly, one person asked for simpler language, and two persons raised concerns about the design of the third section about developers' beliefs on how automated and manual deployment coordination compare to no coordination.

## 3.5 Analysis

We now analyze the hypotheses from Table 3.1. For all hypotheses, their negation is the null hypothesis, and a confidence level of 95 % is applied. The error margin is 9 % because we received 134 individual responses within the population of size 55.3 million. Table 3.5 shows the effect sizes of all 14 accepted hypotheses. All other hypotheses from Table 3.1 were rejected.

For majority statement hypotheses, the share of supporting responses must be higher than $59\,\% = 50\,\% + \textit{error margin}$ to accept the hypothesis. All correlation hypotheses test between two variables with ordinal scales. We apply both Spearman's $\rho$ and Kendall's $\tau$ rank correlation methods, which lead to agreeing results for all tested hypotheses. For RH 3.6.1.1 to RH 3.6.1.6 and RH 3.6.3.1 to RH 3.6.3.6 we excluded responses that answered "I prefer not to answer." For the hypotheses about influence, we assessed whether there are significantly different answers to the ordinal variable between the categories of the categorical variable using the Kruskal-Wallis test. For the accepted hypotheses, we then performed a pairwise comparison using the Mann-Whitney U test to identify between which categories significantly different answers exist. Two influence statement hypotheses were accepted, and for both, no difference could be confirmed between the categories.

## 3.6 Core Insights

The results and analysis of our study lead to the following research insights, answering the research questions of this chapter.

**RI 3.1** Most applications depend on other applications (RH 3.1.1). Only 11 % of the participants' primary applications did not require another application for their correct operation, while 2–5 dependencies are common (44 %). 20 % depended on more than 10 other applications (Figure 3.3a). This answers RQ 3.1: dependencies among applications are very common.

**Table 3.5:** Accepted hypotheses tests of the Dependencies in DevOps Survey 2021. All other hypotheses tests from Table 3.1 are rejected.

| Majority | Support | Correlation | Kendall | | Spearman | | Influence | $H$ | $p$ |
|---|---|---|---|---|---|---|---|---|---|
| Hypothesis | | Hypothesis | $\tau$ | $p$ | $\rho$ | $p$ | Hypothesis | | |
| RH 3.1.1 | 88.8 % | RH 3.2.3 | 0.29 | 0.0 % | 0.33 | 0.0 % | RH 3.6.4.3 | 9.94 | 4.1 % |
| RH 3.2.1.1 | 87.3 % | RH 3.3.2 | -0.14 | 2.5 % | -0.17 | 2.2 % | RH 3.6.5.6 | 8.42 | 3.8 % |
| RH 3.2.1.2 | 68.7 % | RH 3.5.2 | -0.21 | 0.2 % | -0.24 | 0.2 % | | | |
| RH 3.3.1 | 76.1 % | RH 3.5.5 | 0.12 | 4.1 % | 0.15 | 4.3 % | | | |
| RH 3.4.1 | 87.3 % | RH 3.6.1.3 | -0.19 | 0.7 % | -0.24 | 0.5 % | | | |
| RH 3.4.3 | 81.3 % | RH 3.6.3.4 | 0.20 | 0.6 % | 0.25 | 0.5 % | | | |

**RI 3.2** Dependencies between applications constrain the order of their deployment (RH 3.2.1.1) and undeployment (RH 3.2.1.2). Only 13 % of the participants stated that dependencies do not constrain the deployment order, while 22 % answered that they definitely do; the answers in between mean that some dependencies imply a deployment order (Figure 3.3b). Participants with a higher likelihood that dependencies constrain the order of deployment were also more likely to face such constraints for undeployment (RH 3.2.3). Further, better SDO performance correlates with a lower likelihood of such constraints for deployment (RH 3.5.2), i.e., dependencies less often constrain deployment order in more developed DevOps organizations. This correlation is not significant for the order of undeployment, which is less likely to be constrained by dependencies. Lastly, the respondents' experience and their region influenced the likelihood that undeployment is constrained (RH 3.6.1.3 and RH 3.6.4.3); however, as most participants came from Europe, we suspect that the regional insight may be an artifact of the small samples of participants in other regions. In summary, our survey answers RQ 3.2, indicating that, in practice, dependencies constrain deployment order—even though this contradicts the widespread goal of idealistic loose coupling as promoted by, e.g., service-based architectures [65].

**RI 3.3** Deployments across teams need manual coordination (RH 3.3.1). 76 % of the participants relied on manual coordination to ensure the correct deployment order across teams, and 32 % did not support deployment coordination with automation at all (Figure 3.3c). The more respondents relied on manual coordination, the stronger they agreed that dependencies constrain the deployment order (RH 3.3.2). Also, the company size influenced the choice of automation over manual approaches (RH 3.6.3.4). Hence, the answer to RQ 3.3 is that manual coordination is widespread.

**RI 3.4** IT professionals think that automated coordination has better SDO performance than manual coordination (RH 3.4.3). This belief was stronger the better the SDO performance of their organization was (RH 3.5.5). We also found that they believe manual coordination yields worse SDO performance than when they would not need to coordinate at all (RH 3.4.1). Surprisingly, this correlation could not be confirmed between automated coordination and no coordination (RH 3.4.2 was rejected). Overall, 79 % of the respondents expected better SDO performance for automated coordination than when no coordination is needed. We expected worse or similar performance because any coordination introduces overhead. From our perspective, this is an unjustified belief, but it shows how much practitioners value automation. Answering RQ 3.4, we found that the respondents think automated coordination leads to better SDO performance than manual coordination.

These research insights encompass the answers to RQ 3.5 and RQ 3.6. Further, the organization's industry influenced its SDO performance level (RH 3.6.5.6).

## 3.7 Threats to Validity

We now discuss threats to the internal and external validity of the survey.

**Internal Validity** Some questions rely heavily on the term *primary application*. This choice was inspired by Forsgren et al. [79] to guide respondents who are in contact with many different applications. Still, some respondents found this narrowing hard, e.g., when focusing more on abstract frameworks or libraries. The choice of *no coordination* as a comparison reference in SQ A.3.1 to SQ A.3.4 introduced complexity to these questions and was reported by respondents to require precise reading, indicating that less careful respondents may have made mistakes. The suitability of the SDO clustering taken from Forsgren et al. [79] is at the core of big parts of the analysis.

Later questions may have influenced the answers to some questions in the survey that detail and narrow automated coordination, especially in SI A.3. We asked the participants to fill the questionnaire in order; however, it was not enforced.

Potentially, responses could have been submitted from people who were not within the target audience. However, we monitored the responses and analyzed them manually. We did not identify obvious forms of such abuse. Moreover, we got feedback from multiple invited participants who did not fit into the target audience anymore that they did not complete the survey, e.g., because they do not work with software applications anymore or felt too unsure about the required operational knowledge.

Lastly, the survey may suffer from the Hawthorne effect, i.e., participants answered in a way that lets them look more positive by having better SDO performance and favoring automation stronger than they do.

**External Validity** Relying on non-probabilistic snowball sampling and the sample size of only 134 due to a low response rate—although typical for self-administered surveys [34]—is a threat. We believe that our sample is representative as its demographics (SQ A.4.1 to SQ A.4.5) are similar to the bigger study of Forsgren et al. [79]—apart from the concentration on European participants—and for our major questions, almost no significant differences between different demographic groups could be confirmed (RH 3.6.2.1 to RH 3.6.2.6, RH 3.6.4.1 to RH 3.6.4.6 and RH 3.6.5.1 to RH 3.6.5.6).

## 3.8 Conclusion

We performed the Dependencies in DevOps Survey 2021 with 134 to assess the state of dependencies between applications and related coordination of deployments in practice. Our survey confirms that applications depend on each other (RI 3.1), often constraining their deployment and undeployment order (RI 3.2). In most cases, manual coordination is used to ensure the correct deployment order across teams (RI 3.3)—even though IT professionals assume better SDO performance with automated coordination solutions (RI 3.4). These results indicate the need to automate the deployment coordination among teams, which we address in the next chapter.

## Chapter 4

# Automated Decentralized Deployment Coordination[1]

In this chapter, we propose $\mu$S, a novel PL-IaC solution that automates deployment coordination in a decentralized fashion. $\mu$S is the first automation approach to coordinate deployments requiring neither a centralized authority nor manual intervention, ensuring correct deployment orders across separate IaC programs. $\mu$S addresses the automation research gap that we found in Chapter 3 and is the technological basis for Chapter 5, where it enables reliable safe DSU in decentralized organizations.

Section 4.1 summarizes the use cases and approaches for coordinating deployments across teams, establishing the lack of decentralized automation. We close this gap by proposing $\mu$S, enabling developers to express inter-deployment coordination in IaC programs (Section 4.2) and automating the deployment coordination in a decentralized fashion (Section 4.3), enabling teams to execute their deployments independently and asynchronously without manual coordination or centralization. Section 4.4 details the implementation. Section 4.5 evaluates $\mu$S for effectiveness, performance, and applicability to existing IaC Programs. Finally, Section 4.6 discusses $\mu$S' properties and limitations, and Section 4.7 concludes.

## 4.1 Coordinating Deployments in Decentralized Organizations

With modern service-based architectures, systems comprise many separate applications that jointly provide the system's functionality. Ideally, each team independently deploys its applications in DevOps organizations with cross-functional teams. However, in the previous chapter, we identified that the correct operation of an application often depends on the availability of other applications. If these dependencies span across

[1] Based on the authors' work in [225, 233].

teams, their deployments must be coordinated to ensure dependency availability, i.e., applications are only deployed when their dependencies are. Unfortunately, existing IaC solutions to orchestrate multiple deployments are centralized and cannot ensure that dependencies between applications in deployments of different teams are satisfied. Hence, such dependencies require decentralized organizations to fall back to manual coordination, i.e., teams must communicate out-of-band, e.g., via phone, chat, or email. This approach is problematic because it (1) hampers agility and slows down software evolution [36, 140] and (2) is error-prone because of potential miscommunication [167].

To illustrate deployment coordination techniques available for decentralized DevOps organizations, we consider the TeaStore [126], a case study of a microservices application in online retailing. Kistowski et al. developed it for benchmarking and modeling service-based software, and we use it as a running example in this chapter. According to our survey results in Chapter 3, TeaStore is representative. TeaStore consists of six applications (Figure 4.1): (1) *WebUI* is publicly accessed through a *Load Balancer*, (2) *Image* hosts images, (3) *Auth* handles authentication, (4) *Recommender* provides product recommendations, (5) *Persistence* is the storage backend for all applications and backed by a *Database* (DB), and (6) *Registry* lists all application instances for load balancing. All applications are horizontally scalable except for Registry, which is a singleton instance. All components reside in a single virtual private cloud (VPC). They run as serverless container applications (as offered, e.g., by AWS Fargate [13]) within the same container cluster, and the DB is serverless, too (as offered, e.g., by AWS RDS [12]).

TeaCorp, TeaStore's fictional company, adopts DevOps: Each application in Figure 4.1 is developed and operated by a dedicated team managing the application's code, infrastructure, and deployment. Additionally, the WebUI team manages the load balancer, the Persistence team the DB, and the Registry team maintains the shared VPC and cluster.

### 4.1.1 Use Cases for Coordination

The arrows in Figure 4.1 are dependencies between applications, e.g., Persistence depends on DB and Registry. Apart from Registry, every TeaStore application requires two to five (in the case of WebUI) other applications for its correct operation. Such dependencies are not limited to applications but may refer to any infrastructure entity, e.g., Persistence also depends on the cluster and DB. Dependencies between applications constrain the order of their (un)deployment because each application requires that its dependencies are satisfied

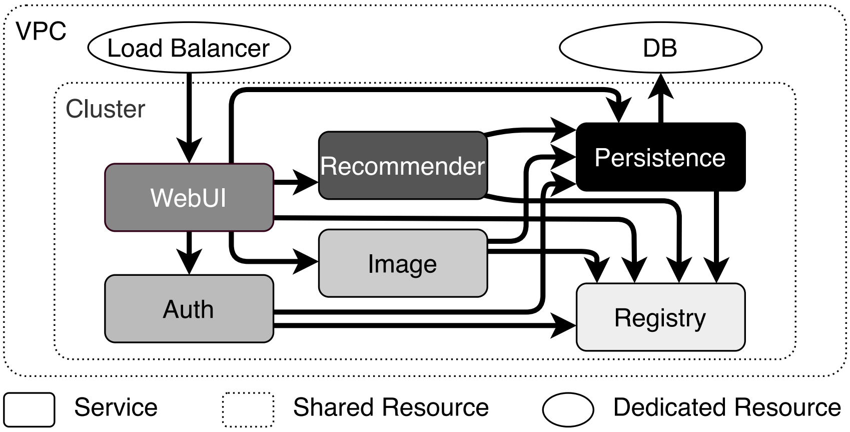


**Figure 4.1:** Applications and their dependencies of the TeaStore. Each application is managed by a dedicated team at TeaCorp.

for correct operation. Further, information is propagated along the dependencies, yielding three use cases for deployment coordination across teams in decentralized organizations.

**Asynchronous Deployment Across Teams** Before deploying an application safely, teams must ensure all its dependencies are deployed. For instance, before Persistence can be deployed, its team must coordinate with Registry's team to ensure it is up and running. For decoupled operations, ideally, the team should be able to start the deployment of Persistence whenever they want, and the application is asynchronously deployed once Registry is available.

**Safe Undeployment Across Teams** Before undeploying an application safely, teams must ensure all applications depending on it are undeployed first. For instance, before Persistence can be undeployed, its team has to coordinate with the Registry team to ensure Registry is undeployed first. For decoupled operations, ideally, the team should be able to start the undeployment of Persistence, which automatically triggers the undeployment of Registry and commences undeploying Persistence after Registry was undeployed.

**Reactive Updates Across Teams** Configuration changes must be transported across the teams' deployments. For instance, if the TCP/IP port of Registry is changed, its team has to communicate that to the Persistence team so that they can adjust their deployment. For decoupled operations, ideally, the Registry team should be able to change their deployment, and the update is automatically propagated and performed in the deployment of Persistence and the other applications that depend on Registry.

### 4.1.2 Available Coordination Approaches

Available solutions to coordinate deployments in such decentralized setups require manual coordination or centralization, which our survey in the previous chapter confirms. All tools respondents used for coordination either supported manual coordination, e.g., chat, emails, and issue tracking, or only supported centralization, e.g., various continuous integration platforms (Table 3.4). This also applies to previous research, e.g., on resource orchestration or infrastructure modeling, where approaches either focus on centralization or do not provide an executable solution.

**Manual Coordination** Most commonly, deployments are coordinated manually, as confirmed by our survey in Chapter 3. Current IaC solutions provide limited support for this approach: each team independently maintains a script for deploying its applications without guaranteeing that dependencies are satisfied across teams. As a result, the teams in TeaCorp have to manually coordinate tasks and communicate synchronously, e.g., via chat or phone, to plan the deployment together. More advanced IaC solutions still do not fully solve this issue. For example, with Pulumi stack references [187], a TeaCorp team can access the exported deployment state from other teams and ensure that an application is only deployed when its dependencies are met. Yet, they cannot coordinate the undeployment order of applications in the same way, potentially leaving the system in an inconsistent state. Also, teams still have to manually coordinate their operations to deploy and undeploy in the correct order. Manual coordination contradicts DevOps's aim of a high degree of automation. It is error-prone, inflexible, time-consuming, and, thus, likely to impact the organization's agility and SDO performance negatively.

**Automated Coordination** Automated coordination promises better SDO performance (RI 3.4). Yet, existing automated solutions are centralized: all teams delegate the deployment of their applications to a single operations team, which ensures that dependencies are satisfied without manual communication. To apply this methodology to TeaStore, a central operations team should maintain the infrastructure for all other teams, i.e., the whole company, ensuring correct deployment and undeployment order. Unfortunately, such a centralized solution separates development and operations, contradicting DevOps' *"you build it, you run it"* principle. It will likely reduce the SDO performance, as communication across teams is required for (1) all changes and (2) application improvements based on operational insights.

### 4.1.3 Automated Decentralized Coordination with $\mu$S

Beyond previous work, we strive to provide decentralized and automated coordination for dependencies among deployments. We propose $\mu$S ([mjuːz] *"muse"*), a PL-IaC approach that solves the issue for decentralized organizations. With $\mu$S, teams independently specify their deployments in separate IaC programs—like they do it today, e.g., with AWS CDK or Pulumi. In contrast to these solutions, however, $\mu$S provides a mechanism to satisfy dependencies across deployments without manual coordination. With $\mu$S, developers explicitly define the resources they require from and provide to other deployments in their deployments' IaC programs. To automate deployment coordination, the execution of each team's IaC program is a continuously running process of the $\mu$S runtime and not—as common today—a one-off task. The separate IaC programs communicate and ensure that resources are only deployed when their dependencies are satisfied. This way, $\mu$S automates the coordination use cases from Section 4.1.1:

**Asynchronous Deployment Across Teams** $\mu$S enables teams to start their deployments independently and deploys resources asynchronously once their dependencies are satisfied. For example, the Persistence team can start their deployment anytime, and $\mu$S delays deploying Persistence right until Registry is deployed by its team.

**Safe Undeployment Across Teams** $\mu$S ensures that all resources depending on a resource $R$ are not deployed anymore when $R$ is undeployed, i.e., if $R$ shall be undeployed, the undeployment of all dependent resources is triggered, and $R$ is only undeployed after their undeployment is completed. For instance, when the Persistence team undeploys its application, $\mu$S automatically undeploys Registry before.

**Reactive Updates Across Teams** $\mu$S automatically transports configuration changes across the teams' deployments and triggers reactive updates. For example, $\mu$S automatically propagates a TCP/IP port change in the Registry deployment and updates the applications using Registry without the manual intervention of their teams.

$\mu$S neither introduces a central authority nor requires manual coordination. Thus, $\mu$S enables safe deployments in decentralized organizations, including modern DevOps organizations. On a technical level, we enhance current PL-IaC solutions by (1) enabling developers to express coordination requirements in IaC programs (Section 4.2) and (2) safely automating coordination at run time by making IaC programs long-running and reactive (Section 4.3).

## 4.2 Expressing Coordination in IaC Programs

Declarative IaC solutions describe the target state of a deployment as a DAG, the resource graph (Definition 2.1). On it, IaC solutions ensure dependency availability (Definition 2.3): a resource is only deployed when all its dependencies are deployed. Resource graphs encode the information to enforce dependency availability *within* a single deployment. To extend this *across* deployments, crucially, µS connects independent resource graphs by *inter-deployment dependencies*, i.e., arcs between nodes in different resource graphs. These arcs specify dependency and, thus, deployment order between resources in independent deployments. An inter-deployment dependency is set up through an *offer* and a *wish* resource. An offer in the resource graph allows another resource graph, the *beneficiary*, to depend on it. The beneficiary defines a wish referencing the offer to introduce the inter-deployment dependency. Like all resources, offers, and wishes can be connected to additional resources in their resource graphs, enabling transitive dependencies among resources across separate deployments. Inter-deployment dependencies may not introduce cyclic dependencies to retain the decidability of the deployment order.

For example, at TeaCorp, each team maintains a separate resource graph, modeling the team's resources and dependencies. Figure 4.2 shows the Auth team's resource graph with all resources required to run the Auth service. The resources depend on three other deployments, expressed by the wishes *cluster*, *service* and *vpc* from Registry, *service* from Persistence, and *securityGroup* from WebUI. Lastly, the Auth deployment allows other deployments' resources to depend on its offers; *securityGroup* for Persistence and Registry, and *service* for WebUI. The combination of all resource graphs through inter-deployment dependencies forms the *global resource graph*, which can be used for global reasoning. However, such a central view is never reified at TeaCorp as it would require centralized access to the resource graphs of all teams.

### 4.2.1 µS IaC Programs

With PL-IaC solutions, e.g., Pulumi, developers describe their deployments by writing IaC programs (cf. Section 2.2). µS builds upon Pulumi TypeScript, retaining all TypeScript features in IaC programs, including OOP abstractions like classes and inheritance, and full compatibility with existing Pulumi TypeScript IaC programs.

Listing 4.1 shows the Auth team's IaC program for Figure 4.2. Lines 4.1.29 to 4.1.46 define the Auth service's node in the resource graph, which requires Persistence through `dependsOn` in Line 4.1.45, creating an arc from the Auth node to the node representing

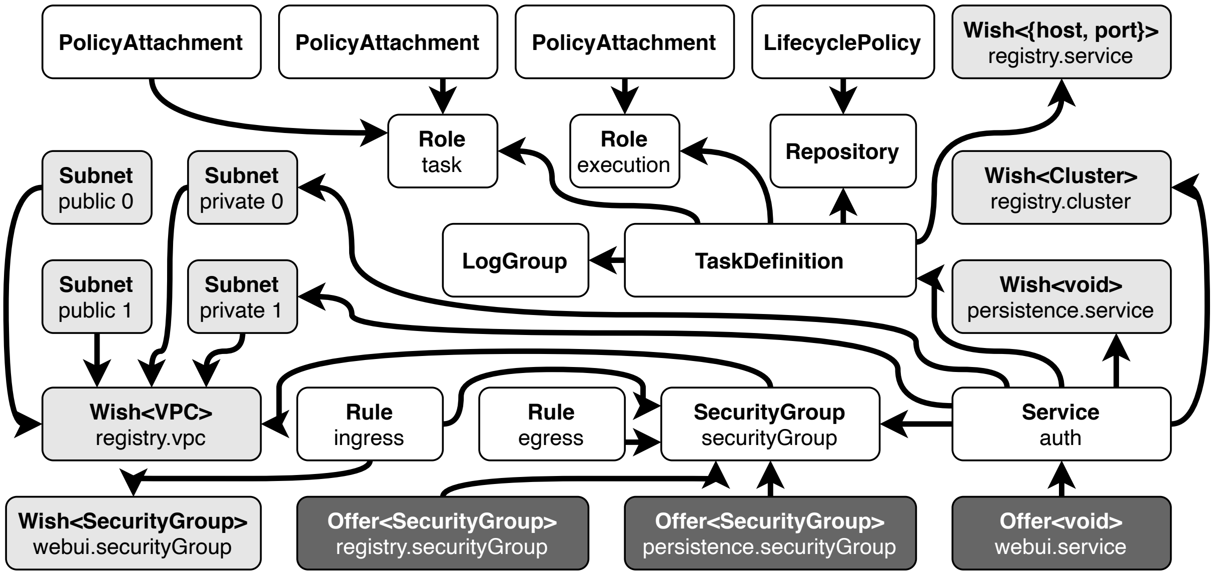


**Figure 4.2:** The Auth team deployment's resource graph.

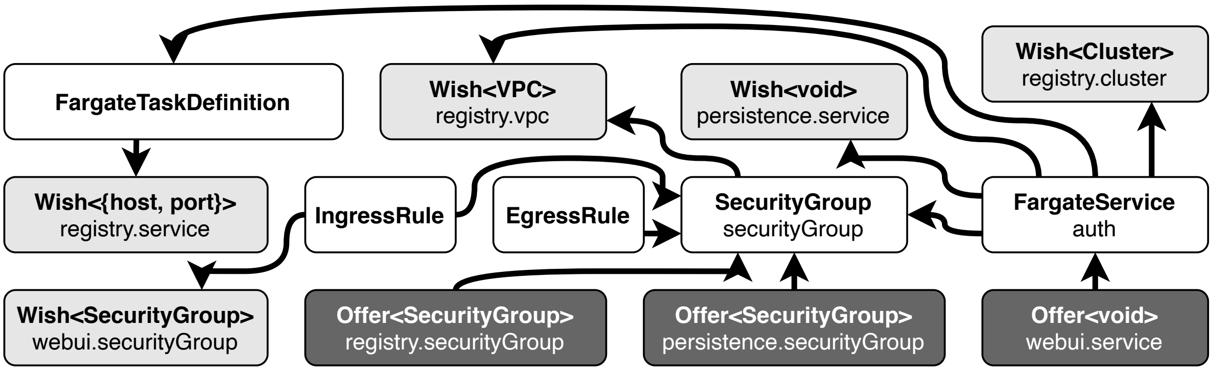


**Figure 4.3:** Simplified developer view on Figure 4.2.

Persistence in the resource graph. Further, Auth depends on the cluster (Line 4.1.30), the VPC (Line 4.1.31), the Docker image (Line 4.1.34), the Registry (Lines 4.1.38 and 4.1.39), and the security group (Line 4.1.43).

$\mu$S leverages compound resources (Definition 2.4) to provide developers with a simplified view. For example, the resource graph in Figure 4.2 is simplified to the developer view in Figure 4.3 (described in Listing 4.1) because `Remote`, `SecurityGroup`, and `FargateService` are compound resources: (1) the vpc and the four subnets are collapsed to `registry.vpc` and (2) nine resources (policies, roles, repository, logging group, and task definition) are collapsed to `FargateTaskDefinition`.

[2] AWS resource interfaces inherited from Pulumi. Arguably better alternative typing would be possible.
[3] `` `${...}` `` used to convert `Output<number>` to expected `Input<string>`.

Listing 4.1: The Auth team's µS IaC program.

```
import * as aws from "@pulumi/aws";
import * as awsx from "@pulumi/awsx";
import * as pulumi from "@pulumi/pulumi";
import { Offer, Remote } from "@mjuz/core/resources";

interface RegistryWishes {
  vpc: aws.ec2.Vpc;
  cluster: aws.ecs.Cluster;
  service: { host: string; port: number };
};
const registry = new Remote<RegistryWishes>(registryKey);
const webui = new Remote<{ securityGroup: aws.ec2.SecurityGroup }>(uiKey);
const persistence = new Remote<{ service: void }>(persKey);

const securityGroup = new awsx.ec2.SecurityGroup[2]("auth", {
  vpc: registry.wishes.vpc,
  egress: [anywhereViaTcp],
});
securityGroup.createIngressRule[2]("webui-inbound", {
  location: {
    sourceSecurityGroupId: webui.wishes.securityGroup.id,
  },
  ports: new awsx.ec2.TcpPorts(8080),
});
[registry, persistence].foreach(
  (remote) => new Offer(remote, "securityGroup", securityGroup)
);

const auth = new awsx.ecs.FargateService[2]("auth", {
  cluster: registry.wishes.cluster,
  subnets: registry.wishes.vpc.getSubnetsIds("private"),
  taskDefinitionArgs: {
    container: {
      image: Image.fromPath("auth", "../auth-service"),
      cpu: 2048 /* 2 cores */, memory: 4048 /* 4 GB */,
      portMappings: [{ containerPort: servicePort }],
      environment: [
        { name: "REG_HOST", value: registry.wishes.service.host },
        { name: "REG_PORT", value: `${registry.wishes.service.port}`[3] },
      ]
    }
  },
  securityGroups: [securityGroup],
}, {
  dependsOn: [persistence.wishes.service],
});

new Offer(webui, "service", undefined, { dependsOn: auth });
```

Listing 4.2: `service` offer for Auth in the Registry team's μS IaC program.

```
new Offer(auth, "service", {
  host: registryHost,
  port: registryPort,
}, { dependsOn: registry });
```

### 4.2.2 Connecting IaC Programs through Offers and Wishes

To define inter-deployment dependencies using offers and wishes, μS IaC programs reference remote IaC programs by *remote connections*. For example, Lines 4.1.11 to 4.1.13 define `Remote` resources from the Auth team's deployment to the IaC programs deploying Registry, WebUI, and Persistence.

The type parameter of a `Remote` object defines wishes from remote IaC programs, mapping the names of the expected offers to their expected value types. For instance, Auth defines three wishes from Registry's IaC program in Line 4.1.11 through the type in Lines 4.1.6 to 4.1.10, specifying `vpc`, `cluster`, and `service`. Further, Auth defines a wish for a security group as `securityGroup` from WebUI's IaC program (Line 4.1.12) and an empty wish `service` from Persistence's deployment (Line 4.1.13),

Developers access wishes that are satisfied by an offer of a remote deployment via the `wishes` property of the `Remote` object, which maps the wish's name to the *wish resource*, i.e., the resource that fulfills the wish. A wish resource is a proxy to the values provided by an offer in the remote IaC program. A wish's type may refer to a resource; e.g., `vpc` in Line 4.1.16 is a VPC resource. For other object types, the wish resource has a correspondingly typed output property per field, e.g., `host` and `port` of Registry's `service` offer (Lines 4.1.38 and 4.1.39). For other types, the wish resource provides the typed value as `value` or, in the case of type `void`, has no output property, like for Persistence's `service` offer (Line 4.1.45).

Offers to remote deployments are defined by instantiating an `Offer` object. They are configured with the beneficiary's remote connection, a unique name among the offers to that remote deployment, and the content to be offered. In Line 4.1.48, an empty offer depending on the Auth's `FargateService` is offered to WebUI as `service`. Lines 4.1.25 to 4.1.27 provide the security group as separate `securityGroup` offers to Registry and Persistence. Listing 4.2 shows the `service` offer in the Registry team's IaC program, an object with host and port fields that depends on the Registry's service resource.

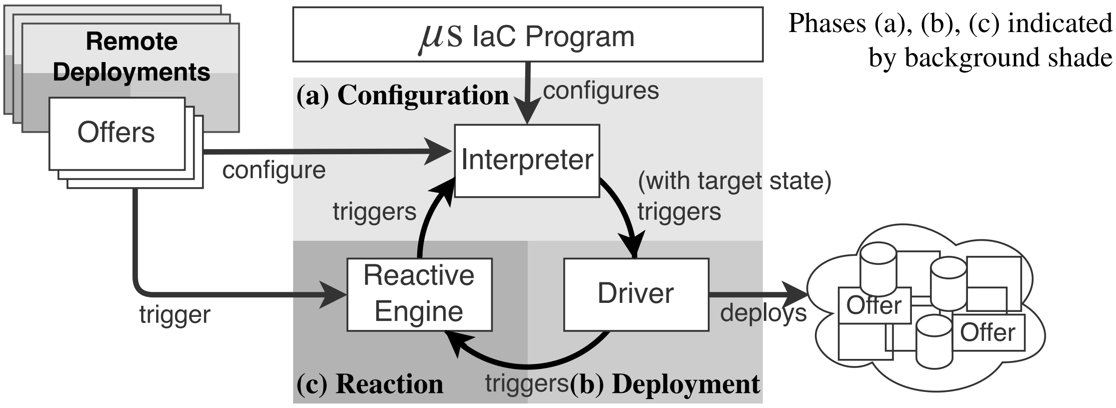


**Figure 4.4:** μS deployment architecture.

### 4.2.3 Deployment Compatibility

A wish is satisfiable if it corresponds to an offer in another deployment's IaC program. If a resource depends on an unsatisfiable wish, it will never be deployed—as if it was not defined. As unsatisfiable wishes threaten availability, μS allows checking the compatibility with connected deployments before the deployment is executed, i.e., whether all wishes are satisfiable by the connected deployments.

μS can generate offer excerpts for each remote connection specified in the IaC program for deployment compatibility checks. An excerpt describes all offers for a particular remote deployment with their types. These excerpts can then be used to validate the wishes in the remote deployment's IaC program, checking that all wished offers exist and that their types are subtypes of the wished types.

## 4.3 Automating Coordination Across IaC Programs

A deployment is a process of the μS runtime executing a μS IaC program. Each deployment can be connected to remote deployments, constituting a distributed system in which the coordination across the IaC programs is safely coordinated based on the defined inter-deployment dependencies expressed in offers and wishes. The deployment iteratively runs through three phases, illustrated in Figure 4.4 and described in the following.

### 4.3.1 Configuration Phase

Each μS IaC program is executed in the *interpreter* of its own μS runtime process during the *configuration phase* (Figure 4.4a). The interpreter receives the IaC program and the offers to the deployment as input from remote deployments and generates the *target state*, i.e., the resource graph, including the resource's configurations. For each resource, the configuration comprises only simple values and outputs of resources on which it directly depends. The outputs' values are available upon deploying the resource because the dependency implies that the resources associated with the outputs are already deployed.

Wish resources are configured by the provided values of the corresponding offers. If a wish is unsatisfied, i.e., the corresponding offer is not deployed, all resources (transitively) depending on the unsatisfied wish are removed from the resource graph, ensuring that they are not in the target state and, thus, not deployed.

### 4.3.2 Deployment Phase

In the *deployment phase*, the *driver* updates the infrastructure based on the target state and the current state (Figure 4.4b). The current state is the resource graph of the currently deployed resources. It also contains the resource configurations, which are fully resolved to values—in contrast to the target state. The current state is initially empty, and the driver updates it according to operations that have been performed. It is saved in persistent storage, from where it is read in consecutive runs to initialize the current state.

The driver implements the CRUDL operations (create, read, update, delete, list) for all supported resources. Reading a resource accesses its configuration from the infrastructure and updates the configuration and the output values in the current state. Creating a resource deploys it in the infrastructure and adds it with its configuration, dependencies, and output values to the current state. Updating a resource updates its configuration in the infrastructure and updates its configuration, dependencies, and output values in the current state. Deleting a resource undeploys it from the infrastructure and removes it with its configuration and dependencies from the current state. Listing resources accesses their configuration from the infrastructure. During deployment, the driver first reads the deployment's current state. Then, these rules are executed in parallel for all resources:

1. A resource is deleted if it is in the current state but not in the target state and if no resource depends on it.

2. A resource is created if it is in the target state but not in the current state and if all its dependencies are in the current state with the same resource configuration defined in the target state.

3. A resource is updated if it is in both the target state and the current state but with different configurations or dependencies. All its dependencies in the target state must exist in the current state and have the same configuration as in the target state.

The driver applies these rules iteratively until the current state and the target state match. Only then are all resource outputs resolved to values in the current state. If the target state is acyclic, *termination* is guaranteed. At any time, dependency availability is ensured, i.e., a resource is only deployed when all its dependencies are, too.

Wishes and offers are treated like any other resource in the deployment procedure, except there is no entity associated with them in the infrastructure. For wishes, the deployment operations, thus, reduce to the changes in the current state. For offers, on deployment, the offered values are made available to the beneficiary deployment. Future requests of the beneficiary for the offer are answered with these values, and if the beneficiary is currently connected, it is informed about the change. Upon deletion of an offer, the beneficiary deployment is informed that the offer is withdrawn, and the removal from the current state is delayed until the beneficiary confirms that none of its deployed resources depend on the offer (anymore).

### 4.3.3 Reaction Phase

To enable decoupled operations, deployments should be started and updated independently, i.e., without (synchronous) coordination across teams. In doing so, it is critical to maintain all dependency constraints across deployments to ensure correct operation. In $\mu$S, the *reactive engine* of the deployment runtime triggers the interpreter and, consecutively, the driver whenever offers from other deployments change (Figure 4.4c). Thus, a $\mu$S deployment is a long-running service continuously adapting the infrastructure rather than a one-off task setting up the infrastructure. As a result, in $\mu$S, deployments are decoupled and can be started and updated independently.

The $\mu$S reactive engine communicates about mutual offers with connected deployments, which may leave and connect at any time. Whenever the state of an offer changes that is associated with a wish in its IaC program, the re-execution of the deployment is triggered. Then, the interpreter generates the new target state, which the driver reaches. For correctness, a single deployment execution takes place at a time. If the reactive engine observes a trigger for re-execution while the deployment is still in the configuration or the deployment phase, the re-execution is delayed to the following reaction phase.

### 4.3.4 Combining All Three Phases: $\mu$S in Action

We are now ready to present how $\mu$S can be used in a DevOps organization where deployments—similar to conventional application code—change over time. As programs

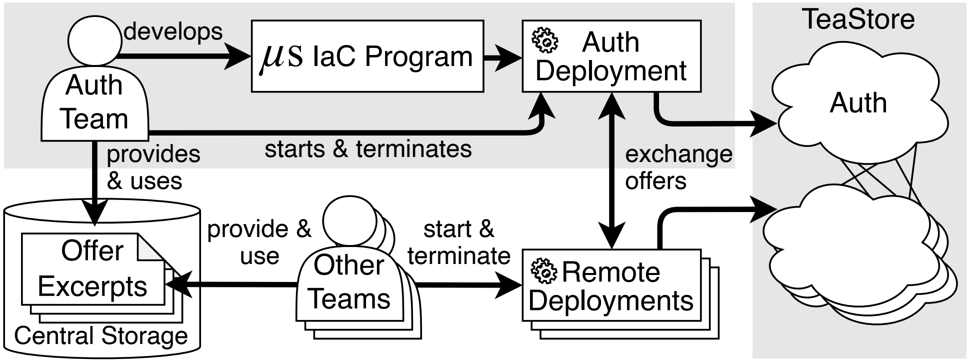


**Figure 4.5:** The Auth team's μS setup at TeaCorp.

and their infrastructure evolve, IaC programs need to be updated. To minimize the impact on the running system, the managed resources should continue operation during the update. Also, the updates of connected deployments should be independent. These requirements are met through *rolling updates* without pausing connected deployments.

A μS deployment is updated by terminating and restarting it with the new IaC program. Termination of a deployment does not undeploy its managed resources. It only stops the deployment runtime in a consistent state, i.e., it ensures that the current state correctly describes the infrastructure. For this, if the deployment phase is ongoing, μS waits for its completion. After restart, the deployment continues with the latest current state and the new IaC program.

Figure 4.5 shows the Auth team's setup at TeaCorp. μS executes the Auth team's IaC program (Listing 4.1) as a continuously running service that communicates with the deployments of the other teams to exchange mutual offers. Initially, and whenever the offer from another deployment changes, the deployment updates the infrastructure. The infrastructure hosts, together with the other teams' infrastructure, the TeaStore. Thanks to μS, the Auth team can safely change its deployment without synchronous or manual coordination with other teams. The Auth team uses a CI/CD pipeline to automate its deployment updates. When a new version of the IaC program is completed, its compatibility is checked against the offer excerpts from the other teams. In case of incompatibility (i.e., an unsatisfiable wish), the pipeline is stopped. In rare cases, the pipeline may be resumed manually, e.g., if the Registry team promised a new offer to Auth but has not updated its offer excerpts yet. Otherwise, the Auth team's offer excerpts are generated from the new IaC program and updated in the company-wide storage. Finally, the deployment is terminated and restarted with the new IaC program.

## 4.4 Implementation

With the existing PL-IaC solutions, i.e., the CDKs and Pulumi, IaC programs are executed as one-off tasks. Thus, $\mu$S cannot be directly implemented in these solutions because its IaC programs are long-running and react to external signals. Thus, we implemented $\mu$S' runtime in TypeScript based on Hareactive [258], a functional reactive programming library to ensure continuous reactivity to deployment changes (Section 4.3.3), and Pulumi [186]. $\mu$S IaC programs are executed in $\mu$S' runtime and can use any resource types available for Pulumi TypeScript programs. $\mu$S built-in resource types are implemented as Pulumi resources, too. $\mu$S is Apache 2.0 licensed and public on GitHub with long-term archived releases on Zenodo [240].

We decided to implement $\mu$S on top of Pulumi and TypeScript for multiple practical advantages: (1) full expressivity of an industrial-strength language, (2) simplified adoption as many developers are familiar with TypeScript, and (3) full compatibility with existing Pulumi TypeScript projects. Our approach extends Pulumi with a reactive runtime and resource implementations for $\mu$S built-in resource types `Offer`, `Remote`, `RemoteConnection`, and `Wish`. Pulumi provides $\mu$S' interpreter and driver, enabling that all Pulumi IaC programs implemented in TypeScript are valid in $\mu$S and, thus, can be used with $\mu$S out-of-the-box—without any changes. $\mu$S resource types are implemented as Pulumi dynamic resource providers [147]. `Remote` is a component resource to define a `RemoteConnection` and its `Wish` resources jointly (cf. Listing 4.1). In addition, all resource types available as a library from or for Pulumi can be used in $\mu$S IaC programs.

The $\mu$S runtime executes and deploys (cf. Section 4.3.1 and Section 4.3.2) a $\mu$S IaC program using Pulumi's Automation API [185]. We extended Pulumi's deployment engine with resource graph pruning to remove all resources that depend on unsatisfied wishes from the target state. This is repeated when an external offer changes (cf. Section 4.3.3). The reaction runtime ensures the sequential execution of deployment rounds.

The resource implementations communicate via gRPC with their $\mu$S runtime to update and retrieve the state of offers, wishes, and remote connections. $\mu$S deployments also use gRPC across each other as defined by `RemoteConnection` resources. Between consecutive runs, the $\mu$S runtime only requires the deployment's current state that is persisted using Pulumi's state management. No additional state is required as all information is reconstructed from the current state on the first deployment round after a restart.

**Table 4.1:** μS evaluation overview.

| Research Hypothesis | | Evaluation Method | Hypothesis Confirmed |
|---|---|---|---|
| RH 4.1.1 | μS can automate decentralized deployments. | Implementing TeaCorp's TeaStore deployment from Section 4.1. | ✓ |
| RH 4.1.2 | μS introduces only negligible coding overhead over its competitors. | Comparing SLOC of TeaStore deployments. | ✓ |
| RH 4.2.1 | μS deployments are not slower than deployments with competitors. | Comparing duration of a standard deployment. | ✓ |
| RH 4.2.2 | μS' deployment time is constant for independent dependencies. | Measuring joint duration for multiple independent deployments. | ✓ |
| RH 4.2.3 | μS' deployment time scales linearly for sequential dependencies. | Measuring joint duration for a chain of dependent deployments. | ✓ |
| RH 4.3.1 | μS applies to existing IaC programs. | Executing existing Pulumi TypeScript programs in μS. | ✓ |
| RH 4.3.2 | Existing distributed IaC programs connected with explicit interfaces can be converted to μS. | Automatically converting 64 Pulumi TypeScript programs connected through stack references to μS. | ✓ |

## 4.5 Evaluation

In this section, we evaluate the design and the implementation of μS, asking the following research questions:

**RQ 4.1** Does μS effectively support deployment automation in DevOps organizations? We are interested in whether μS effectively ensures safe deployments in decentralized environments. It is crucial to assess whether μS can automate deployments in a context where current solutions need manual coordination.

**RQ 4.2** How does μS' performance compare to industrial-strength PL-IaC solutions? We are interested in the run time performance of μS. Performance is important to ensure that μS' automation does not come at the detriment of slow deployments.

**RQ 4.3** Can μS be applied to existing IaC projects? We are interested in the applicability of μS to existing projects. Ensuring that μS can be applied to real-world IaC projects and assessing the required migration effort is important.

We break the research questions down into research hypotheses (RH) in Table 4.1 and state the evaluation method to confirm each. We ran all experiments on AWS Fargate [13] and used Pulumi and AWS CDK as a baseline.

**Table 4.2:** Size of the IaC programs at TeaCorp (SLOC).

| Team | Auth | Image | Persistence | Recommender | Registry | WebUI | Total |
|---|---|---|---|---|---|---|---|
| μS | 61 | 63 | 88 | 63 | 75 | 144 | 494 |
| Pulumi | 53 | 56 | 80 | 56 | 59 | 129 | 433 |
| CDK | 47 | 48 | 91 | 47 | 59 | 73 | 365 |

### 4.5.1 Effectiveness

To answer RQ 4.1, we implemented three versions of the TeaStore application's deployment (Section 4.1)—with μS, Pulumi, and AWS CDK—and compared automation and definition overhead.

First, we considered the support for decentralized deployments. With Pulumi, AWS CDK, and μS, each team can have a separate IaC script for its infrastructure, all together deploying the TeaStore. With all systems, the teams can manually coordinate the order of the deployments, but this limits SDO performance. With AWS CDK and Pulumi, a team can access other teams' deployment states to verify that dependencies are available. μS is the only solution that fully automates the coordination, confirming RH 4.1.1.

Second, to evaluate the coding overhead required by μS (RH 4.1.2), we compared the IaC program size for each IaC solution. Table 4.2 reports the SLOC of each team's IaC program in TeaCorp. Together, the teams need 14 % more lines with μS than with Pulumi and 35 % more lines than with AWS CDK. This is due to the additional information in offers and wishes required to enable automated coordination. AWS CDK is shorter because the default configurations of the patterns in its construct library [10] require less configuration than the pendants in Pulumi's Crosswalk for AWS library [181].

In summary, to answer RQ 4.1, μS is as effective as Pulumi or AWS CDK. There is negligible coding overhead. On top, μS neither requires centralization nor manual coordination. The evaluation suggests that adopting AWS CDK's best practice patterns as μS library could further reduce the required effort for μS deployments.

### 4.5.2 Performance

To answer RQ 4.2, we first measured the duration of a containerized HTTP web service deployment with μS and compared it to AWS CDK and Pulumi. Second, we assessed the run time of the automated coordination of multiple depending deployments in parallel and sequential setups. For the experiments, we deploy AWS Fargate container services with an echo web service instance [102] into an existing VPC and cluster.

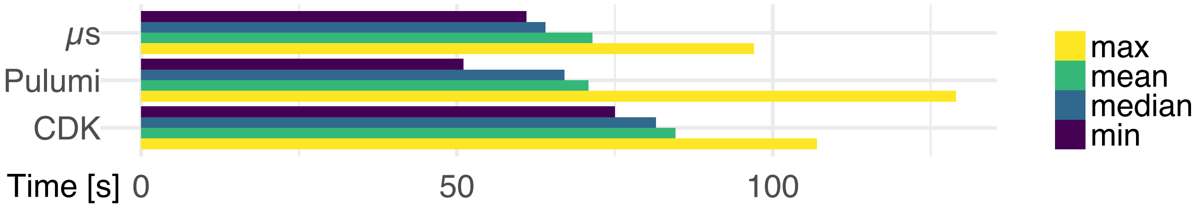


**Figure 4.6:** Required time to deploy a single service with *μ*S, Pulumi, and AWS CDK.

In the first experiment, we assessed the performance of a deployment with *μ*S compared to AWS CDK and Pulumi. We repeated the measurements of the deployment duration 15 times. Each measurement started at process start and ended for *μ*S at the first driver run termination and for Pulumi and AWS CDK at the process exit. Both appear directly after the deployment tool ensures the service is available. The results are in Figure 4.6 and show that Pulumi and *μ*S deployments took similarly long, while deployments with AWS CDK took, on average, 20 % longer. Hence, *μ*S is not slower than AWS CDK nor Pulumi, confirming RH 4.2.1.

In the second experiment, we assessed the performance of *μ*S' automated coordination. In the parallel setup, all deployments depended on the same lead deployment and could be deployed in parallel once the lead deployment was available. In the sequential setup, the deployments' dependencies built a chain towards the lead deployment. Hence, deployments took place sequentially after the lead deployment. We measured both setups with 3, 6, and 12 services and repeated each experiment 3 times. Figure 4.7 shows the number of deployed resources after starting (un)deployment of the lead deployment over time. As expected by RH 4.2.3, in the sequential setup, the time increased linearly with the number of services, i.e., three needed ~ 3.5 minutes and 12 services 4× as much (~ 14 minutes). The parallel setup (RH 4.2.2) required, independent of the number of services, roughly double the time (for the lead deployment and the deployments that depended on it), ~ 2.5 minutes, compared to the single service experiment in Figure 4.6. Deployment and undeployment showed the same behavior; undeployment was faster. The results showed no significant overhead of automated coordination, entailing the behavior expected in RH 4.2.2 and RH 4.2.3.

The experiments answer RQ 4.2, showing that deployment duration is comparable or better than state-of-the-art PL-IaC solutions, and *μ*S' automated coordination does not introduce significant delay.

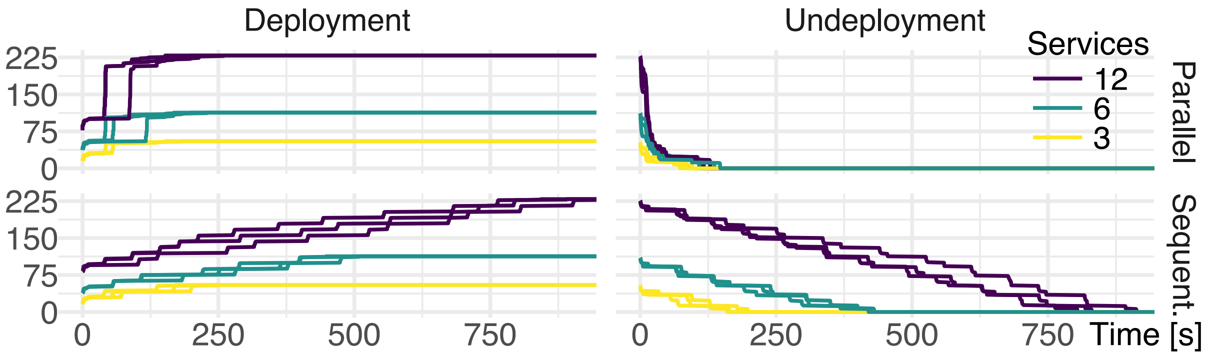


**Figure 4.7:** Number of resources over time when (un)deploying services in parallel and sequentially with $\mu$S.

### 4.5.3 Applicability

To answer RQ 4.3, we applied $\mu$S to third-party open-source projects. First, by the design of our system, every Pulumi TypeScript program is a valid $\mu$S IaC program, making $\mu$S virtually compatible with any third-party, Pulumi-based TypeScript project, satisfying RH 4.3.1. Of course, when starting from a centralized Pulumi IaC program, the correct splitting into separate deployments, one for each DevOps team, requires manual intervention because the organization's team structure is not explicit in the code. Hence, naïvely porting a Pulumi program to $\mu$S does not benefit from $\mu$S' automated coordination out-of-the-box.

To demonstrate $\mu$S' deployment automation, we focused on a subset of Pulumi programs already partitioned and used in a decentralized way. These Pulumi programs use stack references [187] to access exported state of remote deployments, making the boundary between deployments and their interfaces explicit. Such scripts can be migrated to support automated deployment by (1) replacing resource exports with offer definitions and (2) Pulumi stack references with wish definitions to access these resources. We built a dataset [238] of third-party, real-world projects, starting from all projects on GitHub containing TypeScript files creating `pulumi.StackReference` instances. Through GitHub's Search API, we obtained 64 distinct repositories (February 2021), ranging from 150 to 500 K SLOC (avg. 37 K SLOC). We automatically migrated these projects to $\mu$S by applying a simple script [239] that translated stack references and their access to remote, wish and offer resources based on AST transformation. Our migration of the dataset replaced 197 stack references with 556 wishes. It shows that RH 4.3.2

applies: μS' decentralized coordination can automatically be leveraged if distributed deployments are connected through explicit interfaces, e.g., stack references.

The experiments answer RQ 4.3, demonstrating that μS can be applied to 64 real-world projects. While μS is compatible with any existing Pulumi TypeScript program, we further showed that migrating Pulumi files that already provide the separation into different deployments provides the basis for benefiting from μS' deployment automation without requiring manual refinement of the deployment code.

## 4.6 Discussion and Limitations

Connected μS deployments construct a distributed system. In this section, we discuss aspects that do not impact μS' principles but are relevant in a real-world distributed system and limitations to our approach.

**Disclosed Information and Trust** μS deployments can (necessarily) access information from or influence each other when inter-deployment dependencies exist between them. However, information between deployments is only shared via offers and wishes. An offer discloses a concretely defined share of information to its beneficiary deployment defined as the offered object in the μS IaC program. In the opposite direction, the beneficiary discloses the case in which no resources depend on the offer (at offer withdrawal, cf. Section 4.3.2). Along these information exchanges, authorities maintaining the IaC program, e.g., teams, have to trust each other.

**Safety Protocol** As resources depending on a wish are only deployed when the corresponding offer is deployed, the wishing side has to trust that the offering side deploys its offer and that it adheres to the safety protocol: It only undeploys the offer after informing the wishing side and waiting for the undeployment of all resources depending on it. Vice-versa—given the offering deployment adheres to the protocol—the wishing side can prevent the undeployment of the offer and, thus, of the resources it depends on.

**Availability** In distributed systems like μS, faults can occur anytime. Hence, μS has to account for another deployment's (temporary) unavailability. In μS, the unavailability of a deployment does not impact the availability of its deployed infrastructure, e.g., services are not undeployed even when their deployment fails. μS' safety protocol ensures dependency availability across IaC programs, i.e., resources are only deployed when their dependencies are. On unavailability, if an offer is deployed, the corresponding wish and the resources depending on it are only deployed after the offering and the wishing deployment have reconnected. Vice-versa, the protocol delays the undeployment of an offer until the beneficiary deployment is available again. $\mu$S does not fit scenarios where either is regularly unreachable.

**Consistency** The IaC program, target state, and the current state of the deployments are separate views on the deployment and should align. Further, $\mu$S deployments share information via inter-deployment connections, which leads to replicating information across separate deployments. Developers have to be aware of the consistency properties between these entities, especially (until) when they may be inconsistent.

A deployment's target state is inferred through executing the IaC program, which is sensitive to the current state of the environment, e.g., offers from other deployments and other external state and side effects. Thus, multiple executions of a $\mu$S IaC program may result in different target states due to environment changes. Hence, the consistency of the target state with the IaC program is only guaranteed at the moment it is generated.

In $\mu$S, we ensure eventual consistency between each deployment's current state and remote deployments' view on the deployed offers: The reactive engine triggers the re-execution whenever the environment changes. By default, $\mu$S considers changes of offers. If developers use other environmental elements in an IaC program, e.g., external state, they must inform the reactive engine whenever a change occurs.

Generally, deployed infrastructure might *drift* over time, i.e., become inconsistent with the deployment's current state, e.g., through external or manual administration operations. However, ideally, infrastructure managed purely by $\mu$S does not drift. Existing drift can be eliminated by reading the infrastructure's state into the $\mu$S deployment's current state and triggering the re-execution of the deployment phase.

**Supported Infrastructure** While this chapter focuses on serverless computing, $\mu$S is not limited to serverless resources. We only require that a resource is controllable via a CRUDL API. This is naturally the case in serverless computing, but nothing prevents a similar approach form being applied to classic, non-virtualized server-based systems. Also, the tension between DevOps and IaC orchestration highlighted in this paper still holds for server-based approaches. In practice, our implementation can manage all resources for which a Pulumi resource provider can be implemented, including e.g. virtual servers (e.g., AWS EC2 [7]). Yet, we argue that serverless resources are a better fit for DevOps because many operational issues (e.g., scaling) are delegated to the cloud provider, simplifying operations.

**Deployment Coupling** Even if $\mu$S decouples the operations of teams, offers and wishes cause *architectural* coupling because they must be compatible for dependency satisfaction, requiring the exchange of interfaces during development. However, this communication is not critical for operations because it is performed at *design time*, not *deployment time*. Further, $\mu$S offers an asynchronous verification mechanism for compatibility checking (cf. Section 4.2.3). Currently, $\mu$S focuses on 1-to-1 offer–wish dependencies, where the offering and the wishing deployment directly reference each other. We could consider offers to and wishes from any deployment to increase decoupling. This mechanism would require a middleware (e.g., a publish/subscribe system) to mediate indirect dependencies.

**Performance** $\mu$S' performance results are comparable to competitors (cf. Section 4.5.2). Still, deployment time is dominated by the infrastructure platforms managing the deployed resources, causing deployment iterations to take at least seconds and minutes. This process might be too slow for adaptive systems where deployment changes are frequent and must be quick. $\mu$S' approach applies to such scenarios in combination with faster resource orchestration in a lower-latency driver than Pulumi.

## 4.7 Conclusion

Today's IaC solutions cannot automate deployment coordination for decentralized organizations because they require centralization or manual out-of-band coordination, e.g., via phone, chat, or email. To close this gap, we propose $\mu$S, an IaC solution to automate deployment coordination in a *decentralized* fashion, ensuring compatibility with the DevOps practice to embrace the independence of cross-functional teams. $\mu$S enables developers to express which resources they offer and wish from a remote deployment in IaC program code. Then, the $\mu$S runtime uses this information to safely automate the coordination, ensuring dependency availability (Definition 2.3) *across* separate deployments. Crucially, $\mu$S is the first PL-IaC solution where IaC programs are long-running processes—not one-off tasks—that react to external signals. We implemented $\mu$S and showed that it enables decentralized deployments without manual coordination, introduces negligible performance overhead, and is applicable to existing IaC programs.

## Chapter 5

# Safe Dynamic Updates for Workflow-based Systems[1]

In this chapter, we enable safe DSU for workflow-based systems within decentralized organizations. We prevent updates of independently deployable applications from breaking distributed transactions or workflows across them. For this, we present a new model and modular dissemination algorithm tailored for DSU in decentralized environments. We demonstrate the implementation of safe DSU in IaC programs of PL-IaC solutions like µS, which support decentralized deployment coordination, achieving reliable safe DSU for decentralized organizations. Chapter 4 focused on the coordination of coupled deployments, and this chapter builds on it for safe update coordination of decoupled deployments, showing how µS can enable reliable safe DSU in decentralized organizations.

Section 5.1 motivates safe DSU for workflows in decentralized organizations, and Section 5.2 exemplifies the current issues preventing it, leading to our distinction between *essential* and *non-essential* changes for safe DSU, i.e., treating updates that preserve semantics differently. Based on this insight, Section 5.3 describes our novel safe DSU approach Essential Safety in a new formal model, retaining strong safety guarantees on correct system operation at lower disruption than current safe DSU approaches. Section 5.4 presents Essential Safety's practical realization, proposing a novel, modular dissemination algorithm and illustrating how PL-IaC solutions like µS enable reliable safe DSU implementations for decentralized organizations. Section 5.5 describes how previous DSU solutions apply to our new model and dissemination algorithm and analytically compares them with Essential Safety. Section 5.6 evaluates our contributions by simulating 106 realistic collaborative BPMN workflows and analyzing eight monorepos, and Section 5.7 concludes.

[1] Based on the authors' work in [234].

## 5.1 The Need for Safe DSU

Updating long-running software systems is essential to address changing requirements and mitigate security vulnerabilities in a timely manner. Updates are frequent in modern software development following agile methods and DevOps principles [79], requiring automation of updates and having a low impact on the running system.

Traditionally, software updates are performed by shutting down software systems and restarting them after replacing some components with new versions. This applies to modern PL-IaC solutions, too, where developers change the IaC program and rerun it to deploy an update. During such an update, the PL-IaC solution ensures dependency availability (Definition 2.3) within the deployment unit, and $\mu$S ensures it even across separate IaC programs. Hence, all applications that depend on the updated component are undeployed before the update and redeployed afterward. While this is required when applications are coupled—a typical case according to our survey in Chapter 3—it is disruptive and infeasible for larger systems, where a complete restart may take long.

To solve this issue at its root, architecture principles like microservices advocate for loosely coupled applications that allow independent deployment [142, 165], eliminating the need for update coordination and making deployment in decentralized organizations easy. However, if an update can occur at any time, the system relies on fault-tolerant implementations that safely recover and retry all transactions that failed due to an update. Ignoring update safety and retrying the transactions that were broken by an update may be acceptable for applications with inexpensive, short-lived transactions, but the cost of repeating broken transactions increases with the transactions' amount, duration, and resource consumption. Thus, retries after failure on component updates potentially require large amounts of additional resources and introduce severe delays, especially for long-running or frequently executed workflows.

Researchers have investigated safe Dynamic Software Updating (DSU) [218] to alleviate this issue, allowing components to be updated while the overall system continues running *without* breaking transactions or introducing semantic inconsistencies. Safe DSU determines *when* a component can be updated without incurring semantic inconsistencies. Our contributions in this chapter enable decentralized organizations to achieve safe and efficient updates in workflow-based systems with safe DSU. Further, we show that $\mu$S' runtime behavior, i.e., allowing deployment programs to be long-running and react to external signals, enables safe DSU to be implemented directly in IaC programs.

## 5.2 The Dynamic Updating Problem

Workflows are a common solution for implementing long-running, frequent, and expensive transactions, and they are sometimes even referred to as *long-running transactions* [94, 161]. Workflows have been used for a long time and recently gained popularity to orchestrate weakly coupled components. For instance, workflow engines have been adopted at modern software companies (e.g., Conductor at Netflix [164]) and are supported by major cloud providers (e.g., AWS StepFunction [15] and Google Cloud Workflows [92]). Safe DSU is crucial for workflows: Ignoring update safety and retrying the transactions that were broken by an update may be acceptable for applications with inexpensive, short-lived transactions, but the cost of repeating broken transactions may be too high for long-running or frequently executed workflows. This issue is even more relevant in CI/CD pipelines, where changes are small and frequently deployed [50].

A challenge in using safe DSU is that existing approaches introduce significant performance overhead. They either do not reach their update condition in a timely manner (Version Consistency (VC) [24, 152]) or make strong assumptions, sacrificing safety if the assumptions are not satisfied (Tranquility (TQ) [256]). To reduce the performance overhead compared to state of the art and retain safety, we propose the safe DSU approach Essential Safety. Its update condition Essential Freeness is based on the observation that a significant amount of updates are *non-essential* changes, i.e., they never interfere with running transactions because they do not introduce semantic changes. Thanks to identifying non-essential changes, Essential Safety reduces delays and interruptions in workflow-based application updates and retains strong safety guarantees.

We now introduce a workflow, the trip booking saga, as a running example to explain the problem of safe DSU and outline when updating components is unsafe. We motivate our novel safe DSU approach Essential Safety in Section 5.2.2, achieving safe updates at lower disruption than state-of-the-art approaches.

### 5.2.1 The Trip Booking Saga

As the running example, we present the trip booking case study [206], shown as BPMN workflow [170] in Figure 5.1. A hotel, a car, and a flight are booked sequentially. Each step may fail, triggering compensating actions for the bookings performed up to the current execution point—a design pattern referred to as “saga” [82].

Each task in the workflow is implemented as a serverless function. Some of the functions are coupled through a shared database on which they operate, constituting

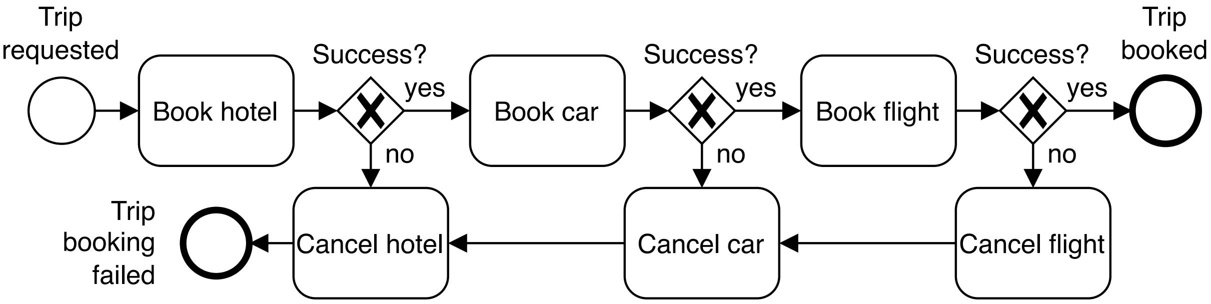


**Figure 5.1:** BPMN workflow of the trip booking saga.

components. In our example, the car booking and cancel functions constitute the *car rental* component, and the remaining four functions are the *holiday* component. These components are the smallest unit of updates; e.g., when the car rental component is updated, the serverless functions for both "book car" and "cancel car" are replaced.

Figure 5.2 shows the trip booking case study as a UML sequence diagram. We added the labels A to E and b to d to reference points in time during the execution. If there is no error, only A to E occur and not b to d, because they are on the paths that only occur on the failure of a booking task, executing the compensation tasks.

Updating a component in a workflow may break the correct execution in two cases. The first case is the update of a component while it is currently executing a task, i.e., it is active. For example, if a workflow instance runs the "book hotel" task, updating the holiday component can cause incorrect behavior. In line with the literature on dynamic updates, we consider updating an active component (i.e., one that executes a task) always unsafe—this problem is studied in a different research line [76, 249, 254] and requires *hot-swapping* code as well as migrating the state representation across versions.

The second case is when a component performs two tasks within the same transaction, and an update introduces a semantic change in between. For example, in the trip booking saga, after "book hotel" completes (after B in Figure 5.2), if "book car" is not successful and the holiday component is updated with a new version that uses a different format for hotel booking IDs, "cancel hotel" does not behave correctly: either it does not find the correct booking to cancel or—even worse—it finds the wrong one. Thus, the workflow instance fails to revoke the hotel booking. To avoid such errors, safe DSU approaches specify *update conditions*. They determine *when* an update can be performed such that it does not cause semantic mismatches.

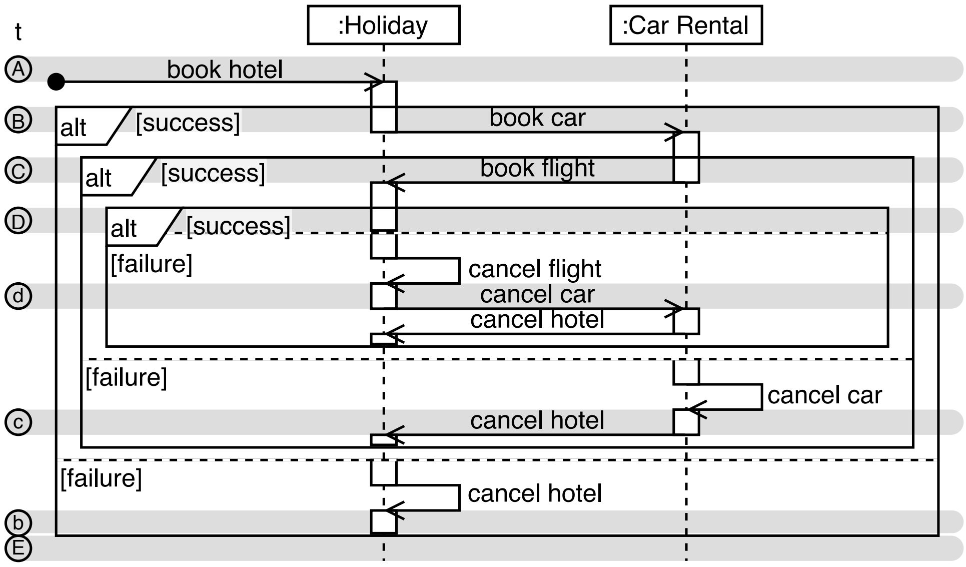


**Figure 5.2:** Sequence diagram of the trip booking saga.

### 5.2.2 The Role of Non-Essential Changes

Safe DSU approaches require monitoring and may block tasks to reach the update condition for a safe update and to uphold it until the update is completed. For instance, to update the car rental component, a DSU approach might block all calls to the car rental component until the update is completed. Once all running tasks on car rental terminate, a safe update condition is met and upheld until the update is completed. Both reaching and upholding a safe update condition lead to significant overhead and delay. The overhead grows with the amount, duration, and resource consumption of the transactions, which results in considerable overhead for highly frequent, expensive transactions that can be found in workflows. DSU approaches should block tasks as little and as short as possible.

For existing DSU approaches [128, 152, 256], the overhead is high in long-running, frequently executing workflows (cf. Section 5.6), prohibiting their use for the safe continuous deployment of such applications. However, in practice, a substantial fraction of the changes running through a continuous deployment pipeline tend to be small and do not introduce semantic changes, i.e., *non-essential changes*—a reality ignored by previous work on DSU. Thus, we can apply a less disruptive update condition to most updates. It requires less task blocking to be reached and upheld and greatly reduces unnecessary overhead.

Essential Safety (ES), our novel safe DSU approach for workflows, leverages the distinction between essential and non-essential changes. Essential Safety reduces DSU

**Table 5.1:** Overview of when safe DSU update conditions are met for the trip booking saga. ✓ met, (✓) only met for non-essential changes, ~~✓~~ met but unsafe at time intervals A to E and b to d for Quiescence (Q), Tranquility (TQ), Version Consistency (VC), and Essential Safety (ES).

| | Holiday | | | | Car Rental | | | |
|---|---|---|---|---|---|---|---|---|
| *t* | Q | TQ | VC | ES | Q | TQ | VC | ES |
| A | | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ |
| B | ▨ | ▨ | ▨ | ▨ | | ✓ | ✓ | ✓ |
| C | | ~~✓~~ | | (✓) | ▨ | ▨ | ▨ | ▨ |
| D | ▨ | ▨ | ▨ | ▨ | | | | (✓) |
| d | ▨ | ▨ | ▨ | ▨ | | | | (✓) |
| c | | ~~✓~~ | | (✓) | ▨ | ▨ | ▨ | ▨ |
| b | ▨ | ▨ | ▨ | ▨ | ✓ | ✓ | ✓ | ✓ |
| E | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

disruption to a minimum while providing the same safety as the state of the art, enabling safe DSU in real-world, long-running, frequently executed workflows.

In Table 5.1, we compare, for existing DSU approaches [128, 152, 256] and for Essential Safety, *when* updating the components in the trip booking case study (cf. Section 5.2.1) is safe. Checkmarks correspond to safe update time intervals. The highlighted cells indicate intervals where the component is active—these intervals are unsafe under all update conditions. If an update is an essential change, Essential Safety provides the same safe update intervals as Version Consistency (VC), which is generally safe—in contrast to Tranquility (TQ). For non-essential changes, there are additional safe intervals, indicated by (✓). Thus, Essential Safety provides the highest number of safe update intervals.

## 5.3 Efficient, Safe Dynamic Updates of Workflow Components

In this section, we present a formal model for workflow execution. We propose Essential Safety as a safe and efficient updating approach. We then show how Essential Safety's update condition Essential Freeness can be reached and upheld during an update.

### 5.3.1 Workflow Execution Model

Various workflow modeling languages exist, including standards like BPMN [170] and BPEL [169], as well as vendor-specific DSLs like the Amazon States Language [6] for AWS Step Functions [15]. Though these modeling languages differ in features and expressivity, they all organize consecutively executed tasks in a graph structure. We formally model their shared core concepts.

We consider the system landscape $L = (\mathscr{R}, \mathscr{W}, \mathscr{T}, \mathscr{C}, i)$ consisting of workflow engines $\mathscr{R}$, workflows $\mathscr{W}$, and tasks $\mathscr{T}$, which are implemented by components $\mathscr{C}$, related by $i : \mathscr{T} \to \mathscr{C}$. We model a workflow $W = (T, P, B, E) \in \mathscr{W}$ with a directed graph of tasks $T \subseteq \mathscr{T}$ that are connected to the tasks which can be executed next—the *succeeding* tasks—by arcs $P \subseteq T \times T$. A workflow's initial tasks are $B \subseteq T$ and the end tasks $E \subseteq T$. All non-end tasks must have at least one succeeding task. Thus, a task $t \in T$ is either in $E$ or there exists at least one edge $(t, t') \in P$. Workflows are executed as workflow instances $I = (r, W, A, V, F, S) \in \mathscr{I}$ in the workflow engine $r \in \mathscr{R}$ where $A \subseteq T$ is the active tasks, initialized as $A = B$. The workflow engine updates $A$ during the execution of $I$. The workflow instance terminates once no task is active anymore, i.e., $A = \emptyset$. $S$ is the workflow instance's state. All tasks $T$ of $I$ can read from it at the beginning of their execution and write to it after their execution. $V \subseteq T$ are the visited tasks, i.e., the set is initially empty, and all tasks that are removed from $A$ during the execution are added to $V$. $F \subseteq T$ are the potential future tasks, i.e., all tasks that are reachable in the directed graph $(T, P)$ from a task in $A$. Note that $F$ is a conservative over-approximation of the future tasks, i.e., not all tasks in $F$ have to be executed. For instance, consider BPMN exclusive gateways, as included three times in Figure 5.1, which have multiple outgoing paths, but only exactly one will be executed. All tasks on a path after a BPMN exclusive gateway are initially in $F$ and all tasks on the paths not taken are removed from $F$ without being executed once the gateway is processed.

Based on the definition of active tasks $A$, we define a component $c \in \mathscr{C}$ is active if it executes any active task:

**Definition 5.1** (Active Component)**.** A component $C \in \mathscr{C}$ is called *active* if it currently executes a task in any workflow instance: $\exists I = (r, W, A, V, F, S) \in \mathscr{I} \mid t \in A \wedge i(t) = C$.

### 5.3.2 Essential Safety

We define a safe and efficient update condition for long-running, frequently executed workflows. A workflow instance always executes correctly if every component is only updated (1) after it has executed its last task, (2) before it executes its first task, or (3) if it does not execute any task in the workflow instance. In contrast to previous work, our approach also allows a component to be updated if it has already executed a task *and* may execute a task in the future if the update does not introduce a semantic change. We call such updates *non-essential changes*, in contrast to *essential changes*, which introduce a semantic modification:

**Definition 5.2** (Essential Change)**.** An update of a component $C \in \mathscr{C}$ from version $v$ to $v'$ is an essential change for workflow instance $I = (r,W,A,V,F,S) \in \mathscr{I}$, if the possible execution of any future task $t \in F \mid i(t) = C$ on $v'$ is not guaranteed to produce the same resulting state $S$ and side effects as executing $t$ on $v$.

Every other change (given the definition above) is a non-essential change. Identifying whether a change is non-essential is not decidable in general as it boils down to the program equivalence problem. Since misclassifying essential changes as non-essential breaks safety, we conservatively under-approximate non-essential changes with a catalog of known non-essential changes that can be found through efficient analyses. Kawrykow and Robillard [124] describe non-essential changes as (1) cosmetic, (2) behavior-preserving, and (3) unlikely to provide further insight into component relationships. This includes—but is not limited to—trivial type updates, local variable extractions, rename-induced modifications, trivial keyword modifications, local variable renames, and whitespace and documentation-related updates. Definition 5.2 leaves open adding more sophisticated analyses to find non-essential changes, including application-specific ones. Identifying non-essential changes is important in practice but orthogonal to our contribution.

Updating a component with non-essential changes is always safe while the component is not active. We introduce Essential Safety (ES): only updating components when they are *essentially free*.

**Definition 5.3** (Essential Freeness)**.** A component $C \in \mathscr{C}$ is *essentially free*, if it

1. is not active and
2. a. will not be active in a workflow instance in which it already executed a task, i.e., $\nexists I = (W,A,V,F,S) \in \mathscr{I} \mid t \in V \wedge t' \in F \wedge i(t) = C \wedge i(t') = C$, or

   b. its update is a non-essential change for all workflow instances $I = (W,A,V,F,S) \in \mathscr{I} \mid t \in V \wedge i(t) = C$ in which it already processed a task.

Considering a single workflow instance of the trip booking saga, updates with non-essential changes can always be performed without violating the workflow's correctness if the respective component is not currently executing a task. For instance, using the intervals marked in Figure 5.2, the car rental component can always be updated except within C and c. If the update is an essential change, the update must not occur between a component's first and last task execution in the workflow instance. For example, an essential change of the car rental component may not occur within B, d, and c because it might be the case that "cancel car" is executed in the future after that "book car" has been already executed on the current version of the component.

### 5.3.3 Reaching Essential Freeness

Strategies to reach safe DSU update conditions trade-off between update *timeliness* and *interruption*. Timeliness is the length of the interval between requesting the update and the beginning of the component exchange, i.e., the point in time when the component stops executing tasks. Interruption is how long a workflow instance's completion is delayed due to the update. The following reaching strategies from the literature [128, 152, 256] can be used to reach Essential Freeness.

**Waiting (W)** The update waits for Essential Freeness. The interruption is limited because only workflow instances that started after the update begins are delayed and the update's duration bounds the interruption. Yet, the update is not guaranteed to start in bounded time, i.e., timeliness is unpredictable. Thus, this approach is not suitable where Essential Freeness rarely occurs by chance.

**Blocking Tasks (BT)** The starting of tasks on the component to update is delayed until after the update. This strategy ensures that Essential Freeness is reached in bounded time, but it may cause more interruption than Waiting.

**Blocking Instances (BI)** The strategy is similar to Blocking Tasks, but instead of delaying tasks, the start of new workflow instances that need the component is delayed until after the update. While this strategy also guarantees the update is reached, it might take longer. The interruption is expected to be similar to Blocking Tasks but is reduced if multiple updates occur in parallel.

**Concurrent Versions (CV)** For non-essential changes, all new task executions are served by the new version running in parallel to the old version, which completes the already running tasks. For essential changes, the old version also executes new tasks belonging to workflow instances that already executed at least one task on it. Thus, the old version remains available until no workflow instance needs it anymore. This strategy provides good timeliness and no interruption, but it requires running two parallel versions of a component, significantly increasing complexity, especially for stateful components.

Except for Concurrent Versions, all reaching strategies require Essential Freeness to hold until the update is completed. This can be achieved by applying Blocking Tasks to a component during its update, delaying the start of new tasks until the update is completed.

## 5.4 Implementation for Decentralized Organizations

Determining and reaching the update condition is trivial with a single centralized workflow engine. Such a central entity (1) knows the state of all workflow instances $\mathscr{I}$, and (2) can delay the execution of tasks and whole workflow instances. However, the system landscape $\mathscr{L}$ may comprise many workflow engines $\mathscr{R}$—especially in a decentralized organization—each hosting a subset of workflow instances $\mathscr{I}$. Hence, no centralized view exists on all workflow instances. Modern, scalable workflow engines, e.g., Zeebe [39], are by default decentralized over multiple separate workflow engines to improve scalability and fault tolerance.

To ensure their safe update, all workflow engines invoking tasks on a shared component have to coordinate. Hence, reaching Essential Freeness for a component requires considering all workflow instances $\mathscr{I}$ that use the component. We propose a dissemination algorithm that the workflow engines use to notify components of their workflow instances' status. The algorithm ensures that components are aware of their role in all workflow instances using them. Each component can then locally decide whether it has reached the update condition and can be safely updated.

### 5.4.1 Dissemination Algorithm

Algorithm 5.1 shows the dissemination algorithm that workflow engines execute for each workflow instance. The four callback procedures in Algorithm 5.1 are called reactively based on the events in the workflow execution, e.g., before a task is started, the procedure in Line 5.1.5 is called. Using the procedures, the workflow engine (1) *announces* to components that they might be used, (2) *marks* components that were used, and (3) *locks* components (not exclusively) while they are used. Every component stores its status in the workflow instances, i.e., every component maintains the information for each workflow instance, whether it received an announcement or a marking, which has not been revoked yet, and a lock counter.

BEFOREWORKFLOW, AFTERWORKFLOW, BEFOREEACHTASK, and AFTEREACHTASK are executed on the workflow engine before/after a workflow instance is executed and before/after each task is run. The ANNOUNCE, REVOKEANNOUNCEMENT, MARK, REVOKEMARKING, LOCK, and UNLOCK procedures are called remotely on the component passed as their first argument. Remote calls are asynchronous unless the execution blocks to get the return value using `await`. Components can delay their response at the `await` synchronization points to interrupt the workflow instance's execution until they can accept the announcement, marking, or lock.

The BEFOREWORKFLOW procedure (Line 5.1.2) announces to all components $c$ (Line 5.1.4) from the set of potential future tasks (Line 5.1.3) of the workflow instance $I$ that $c$ might participate in $I$. Announcements are revoked after completing tasks (Lines 5.1.11 to 5.1.14) if the component will not be used (again).

Before the workflow instance invokes a task on a component for the first time, the engine marks the component (Lines 5.1.6 to 5.1.8). Markings remain for the rest of the workflow instance's execution and are revoked after its completion (Line 5.1.17).

Every time a workflow instance invokes a task, the engine locks the component (Line 5.1.9) and unlocks it after the task is completed (Line 5.1.15). A workflow instance might lock the same component multiple times before unlocking due to parallel task execution. The components internally increase a counter with locking and decrease it with unlocking. If the counter is positive, the workflow instance runs tasks on the component.

The presentation of Algorithm 5.1 is simplified for clarity. Our implementation includes some optimizations, e.g., it sends announcements (Line 5.1.4) in parallel, and the messages to $i(Task)$ in Lines 5.1.5 to 5.1.9 are packed into a single message.

### 5.4.2 Handling Essential Freeness

Algorithm 5.1 disseminates the necessary information to the components to determine, reach, and uphold Essential Freeness. When performing an update that introduces essential changes, a component is essentially free if it holds for no workflow instance an announcement *and* a marking, i.e., no workflow instance that already used the component will use it again. For non-essential changes, a component is essentially free if it is not locked, i.e., for no workflow instance, the lock counter is greater than zero.

Using the Blocking Instances strategy, a component delays the confirmation of announcements until the update has been completed, blocking all new workflow instances that will call the component in BEFOREWORKFLOW and delaying their start. Blocking Tasks uses a similar approach with locks for non-essential changes and markings for essential changes. Locks block all workflow instances, markings only the ones that have not used the component yet. Blocking Tasks does not generally block all task invocations

**Algorithm 5.1** Modular dissemination algorithm for safe DSU in decentralized setups. Executed on workflow engine $r$ for workflow instance $(r, W, A, V, F, S)$ with identifier $I$.

Announcements Markings Locks

```
Announcements, Markings $\leftarrow \emptyset$
procedure BeforeWorkflow
    Announcements $\leftarrow \{c \in \mathscr{C} \mid t \in F \land i(t) = c\}$
    for all $c \in$ Announcements do await Announce$(c, I)$
procedure BeforeEachTask(Task)
    if $i$(Task) $\notin$ Markings then
        await Mark($i$(Task), $I$)
        Markings $\leftarrow$ Markings $\cup\, i$(Task)
    await Lock($i$(Task), $I$)
procedure AfterEachTask(Task)
    PreviousAnnouncements $\leftarrow$ Announcements
    Announcements $\leftarrow \{c \in \mathscr{C} \mid t \in F \land i(t) = c\}$
    for all $c \in$ PreviousAnnouncements $\setminus$ Announcements do
        RevokeAnnouncement$(c, I)$
    Unlock($i$(Task), $I$)
procedure AfterWorkflow
    for all $c \in$ Markings do RevokeMarking$(c, I)$
```

by delaying locks because this could lead to a deadlock in case of essential changes: The not-blocked workflow instances already used the component and may prevent Essential Freeness—which would cause a deadlock—or do not use it anymore. For this reason, Blocking Tasks can only be activated at one component at a time to prevent deadlocks.

For both strategies above, once reached, the update condition must be upheld until the update completed. For essential changes, delaying the confirmation of markings upholds the update condition. For non-essential changes, locks are delayed.

Waiting and Concurrent Versions do not require any aspect of Algorithm 5.1 for their update condition because they do not influence the execution of workflow instances. For Concurrent Versions, workflow instances that were running before the start of an update use the old version, while the new version processes all other workflow instances.

### 5.4.3 Reliable Safe DSU Implementation in IaC Programs

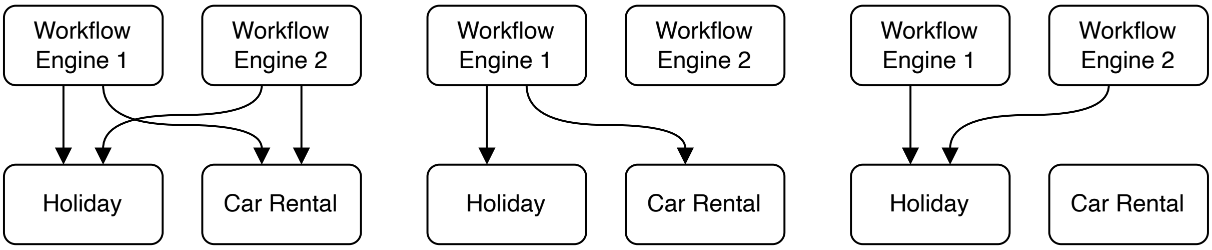
Algorithm 5.1 provides each component with the required information and control for safe DSU, and Section 5.4.2 describes how the components interpret it. At this point, components in this chapter implement two aspects. First, they host a set of tasks that workflows can execute, i.e., they are an application. Second, they determine, enforce, and uphold safe update intervals by communicating with the dissemination algorithm of all workflow instances they may participate in, and they perform their own update, i.e., they are their own orchestrator.

The application and orchestrator are typically separate concerns and software projects. For instance, the application tends to be project-specific, e.g., a custom Java HTTP web service, and the orchestrator is generic off-the-shelf software, e.g., Kubernetes [130], with an operator implementing the safe DSU logic from Section 5.4.2. Developers can write an IaC program configuring the orchestrator to deploy their application, e.g., a Kubernetes service [131] (i.e., a deployment abstraction for network applications). Upon updating, they change the IaC program and rerun it, updating the orchestrator's target state. Afterward, the orchestrator updates the component's service once its safe DSU plugin achieves a safe update interval.

This approach works with today's IaC solutions once the safe DSU mechanism has been implemented in the orchestrator, e.g., as a safe DSU operator for Kubernetes. However, it has two crucial disadvantages from a reliability point of view. (1) The IaC program's state does not correctly reflect the deployment's state, i.e., the IaC program reflects already the new version before the orchestrator deploys it, which is a visibility problem amplified by safe DSU because it is not known when the next safe update interval will be reached and the update starts compared to traditional updating. (2) The deployment logic is distributed over the IaC program, the orchestrator, and its safe DSU plugin, making testing and holistic correctness reasoning hard.

IaC solutions supporting automated decentralized deployment coordination, like $\mu$S, can solve these issues. $\mu$S IaC programs are long-running and can react to external signals, wherefore the safe DSU mechanism can be implemented in the IaC program itself, ensuring the IaC program's state reflects the deployments state and that holistic analysis of the deployment logic is possible. Still, of course, the safe DSU logic can be implemented once in a library and reused across all IaC programs, eliminating code duplication across deployments while retaining holistic reasoning capabilities.

As a concrete example, we consider the trip booking saga from Section 5.2.1 with two workflow engines. Both components and the workflow engines have their own $\mu$S IaC programs, connected through offers and wishes (Section 4.2). Algorithm 5.1 is executed for each workflow instance in both deployed workflow engines and communicates with the IaC programs of the holiday and the car rental components. The IaC programs activate and deactivate the dependencies between the workflow engines and the components based on this information, such that the dependencies only exist and prevent updates when the



**(a)** Both WEs have WIs in C, D, d, or c.

**(b)** WE1 has WIs in C, D, d, or c, WE2 only in A and E.

**(c)** Both WEs have WIs only in A, B, b, and E.

**Figure 5.3:** Global resource graphs of the trip booking saga with two workflow engines (WEs) for $\mu$S IaC programs implementing safe DSU for essential changes. All workflow instances (WIs) are within certain time intervals from Figure 5.2.

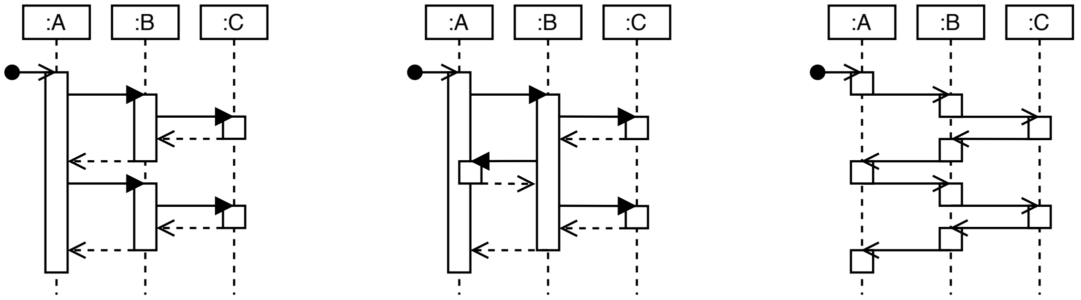
component is not essentially free (Definition 5.3), yielding the global resource graphs illustrated in Figure 5.3. In Figures 5.3a and 5.3b, both components are not essentially free, each having one or more dependent resources preventing their undeployment for updates. In Figure 5.3c, the car rental component is essentially free, i.e., no workflow instance is using it, and no instance that used it may use it again. Thus, there is no dependency arc from a workflow engine to it, allowing the IaC program to undeploy and redeploy car rental in a new version when needed.

## 5.5 Supporting Previous Safe DSU Approaches

This section shows how our work relates to previous work on safe DSU. We show how their models and update conditions fit into our proposed model and dissemination algorithm, demonstrating their compatibility and enabling detailed comparisons.

### 5.5.1 From Transactions to Workflows

Previous work on DSU [24, 128, 152, 256] focused on synchronous distributed transactions in component-based systems. They assume that external clients trigger *root transactions*, which, in turn, can run other (sub-)transactions on the same or other components. The execution blocks until a sub-transaction completes with a return value. In Figure 5.4a, component A runs a sub-transaction on B, which runs a sub-transaction on C. Afterward, C returns a value to B, B one to A, and new sub-transactions are run on B and C. Figure 5.4b shows the same pattern, but synchronous transactions are nested differently: Instead of ending the first transaction on B, B runs a sub-transaction on A.



**(a)** Synchronous transaction. **(b)** Alternative synchronous transaction. **(c)** Asynchronous workflow.

**Figure 5.4:** Comparison of synchronous transactions and asynchronous workflows.

In this work, we model systems as asynchronous workflows. Tasks are started based on their order as defined in the workflow after the previous task(s) are completed. The tasks are coordinated by the workflow engine, which acts as an event-based middleware. Thus, upon completion, tasks send their results as an update of the state $S$ to the workflow engine, which then starts the succeeding task(s) with $S$. Modern cloud-based systems—also beyond workflows—adopt the asynchronous model. Typically, such systems use asynchronous, decoupled communication patterns, e.g., event-based microservice choreographies or serverless computing [23, 142].

Our asynchronous workflow model can emulate the synchronous model. For instance, the transaction in Figure 5.4a can be modeled as in Figure 5.4c: the synchronous parent transactions are split into two tasks (before and after the subtransaction). State can be conveyed via the workflow instance's state $S$. This transformation is neither injective nor surjective: Not every asynchronous workflow can be translated to a synchronous transaction, and multiple differently nested synchronous transactions might be transformed to the same workflow, e.g., both Figures 5.4a and 5.4b result in Figure 5.4c.

### 5.5.2 Previous Safe DSU Approaches

We now present three update conditions from the literature and show how they fit into our asynchronous workflow model.

Kramer and Magee [128] proposed Quiescence (Q), which does not rely on run time information of workflow instances. Instead, the workflow's structure, which is known at design time, and whether a new workflow instance will be started suffice. If necessary, all future workflow instances are blocked to enforce Quiescence.

**Definition 5.4** (Quiescence)**.** A component is *quiescent* if it (1) is not active and (2) will not be active in a workflow instance.

Ma et al. [152] proposed Version Consistency (VC) and evaluated it with a simulation, which was later extended with a system implementation and evaluation based on Apache Tuscany [24]. The update condition of Version Consistency is called Freeness.

**Definition 5.5** (Freeness)**.** A component is *free* if it (1) is not active and (2) will not be active in a workflow instance in which it already executed a task.

Version Consistency is similar to our approach but does not distinguish between different types of updates, which are conservatively over-approximated to essential changes.

Vandewoude et al. propose Tranquility (TQ) [256].

**Definition 5.6** (Tranquility)**.** A component is *tranquil* if it (1) is not active and (2) will not be active in a workflow instance in which it might execute a succeeding task for a component for which it already executed a succeeding task.

Though proposed before Version Consistency, Tranquility effectively corresponds to Version Consistency with the additional assumption that components follow a "black-box principle" [256]. For systems satisfying this principle, Tranquility assumes that version consistency does *not* have to be enforced between the internals of different sub-transactions. For instance, if within the same root transaction, a client uses an authentication component and calls the server, which uses authentication internally, too, the client and server may safely use different versions of the authentication component.

Leveraging the black-box principle, Tranquility results in better update timeliness and less interruption than Version Consistency [152]. However, Tranquility is unsafe for systems that do not follow the black-box principle, as Ma et al. [152] already noticed. It is generally questionable whether workflows follow the black-box principle because their tasks often depend on each other, leading to a violation of the principle.

Quiescence and Version Consistency map directly to our asynchronous workflow model (Section 5.3.1), i.e., Definitions 5.4 and 5.5 can be trivially verified by inspecting the potential future tasks $F$ and the visited tasks $V$. Tranquility (Definition 5.6), however, distinguishes between sub-transactions that are called from the same transaction and ones that are called from a different transaction. For example, component C is tranquil between its two executions in Figure 5.4a, and it is not tranquil between its two task executions in Figure 5.4b. Yet, there is no such distinction in the asynchronous model in

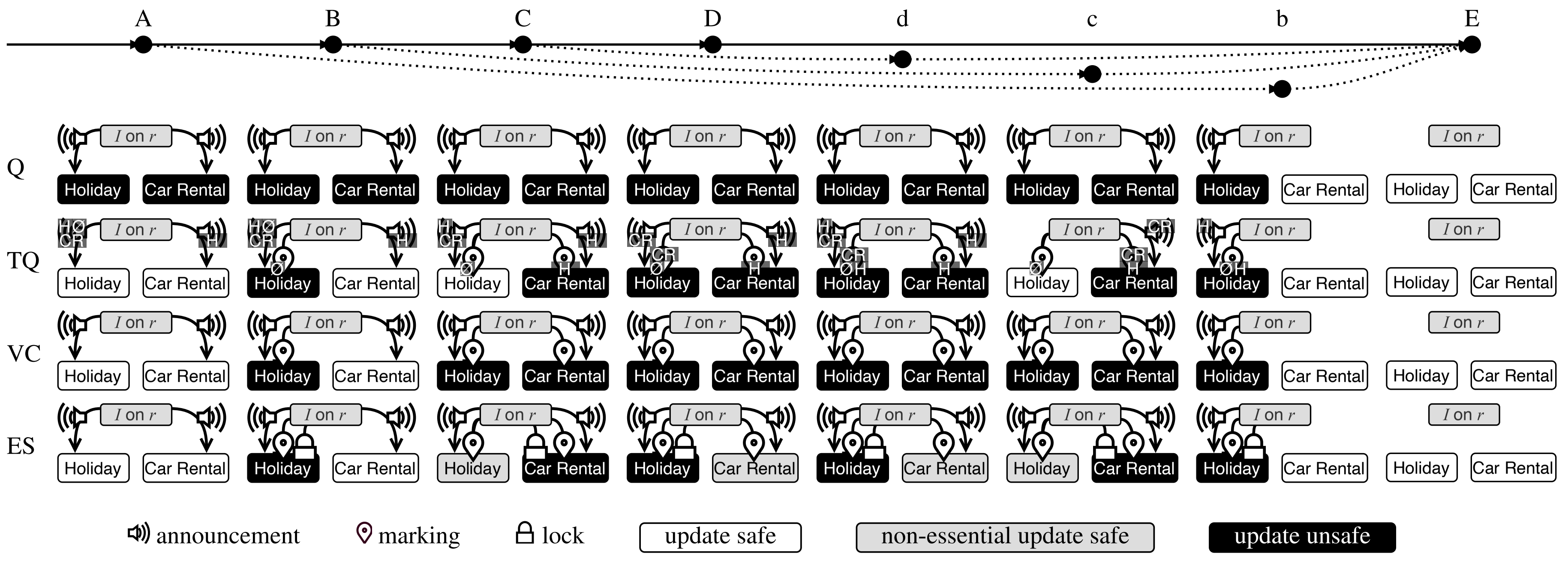


**Figure 5.5:** Example execution of Algorithm 5.1 on the trip booking saga for all safe DSU approaches. Announcements, markings, and locks at the time intervals in Figure 5.2.

**Table 5.2:** Overview of which elements of Algorithm 5.1 each safe DSU approach and reaching strategy requires.

✓ necessary    ✓$^{C}$ per preceding component    ✓* sufficient for non-essential changes
▲ uphold condition (essential changes)    ■ uphold condition (non-essential changes)

| Approach / Strategy | Announcements | Markings | Locks |
|---|---|---|---|
| Quiescence | ✓ ▲ ■ | | |
| Tranquility | ✓$^{C}$ | ✓$^{C}$▲ ■ | |
| Version Consistency | ✓ | ✓ ▲ ■ | |
| Essential Safety | ✓ | ✓ ▲ | ✓ ■ |
| Waiting | | | |
| Blocking Instances | ✓ | | |
| Blocking Tasks | | ✓ | ✓* |
| Concurrent Versions | | ✓ | |

Figure 5.4c compared to the synchronous model (Section 5.5.1). Embracing Tranquility's black-box principle, we assume that all task executions of a component with preceding tasks from the same component belong to a single, synchronous transaction, i.e., for each component, the same version of another component is used for each of its succeeding tasks. With this definition, in the asynchronous workflow in Figure 5.4c, component C is only tranquil before the first and after the second task. Component A, however, may be updated after its first task, i.e., Tranquility is unsafe if components do not respect the black-box principle.

### 5.5.3 Update Conditions, Operationally

Different subsets of our dissemination algorithm (Algorithm 5.1) provide the necessary data to a component to (1) determine that an update condition for the component is reached, (2) reach the condition (quicker), and (3) uphold the condition once reached. Our algorithm is inspired by the control algorithm proposed for Version Consistency, which is based on graph transformations [152] and verified to be correct [24]. *Announcements* are similar to their *future edges* and *markings* to *past edges*; there is no counterpart for *locks*. However, their transaction model lacks a holistic view of transactions within the same root transaction because components do not share their internal logic. In our workflow model, such a view on workflow instances is available in their workflow engine. We leverage this holistic view to reduce communication. Table 5.2 summarizes the parts of Algorithm 5.1 required by the update conditions and reaching strategies. We now provide more detailed insights.

In Quiescence, announcements are sufficient to check if components are quiescent: They are if they have no announcements. To reach Quiescence, Kramer and Magee [128] only discuss Blocking Instances as “passivation” of components, i.e., ensuring that no transactions are invoked on the component in the future. The other three reaching strategies are also applicable but require run time information, which Kramer and Magee do not consider.

Version Consistency requires announcements and markings, but no locks. A component is free if it has, for no workflow instance, an announcement *and* a marking. Ma et al. [152] discuss Waiting, Blocking Tasks, and Concurrent Versions for Version Consistency, concluding that the last one should be the preferred strategy if applicable; otherwise, Blocking Tasks. Essential Safety and Version Consistency are equivalent if all updates are essential changes.

Tranquility also requires announcements and markings. Yet, both need to be per workflow instance *and* component of the preceding task(s), not just per workflow instance, as in Algorithm 5.1. The condition is then similar to Version Consistency: The component is tranquil if it has for no pair of workflow instance and preceding tasks’ component an announcement *and* a marking. For Blocking Tasks, this may deadlock workflow instances, making Blocking Tasks for Tranquility generally unsafe—also for single updates. Vandewoude et al. [256] use Waiting for Tranquility and resort to Blocking Instances if the update point is not reached.

### 5.5.4 Comparing the Update Conditions

Figure 5.5 shows the execution of Algorithm 5.1 in the trip booking saga (Section 5.2.1) for the discussed update conditions. The columns show the time intervals from Figure 5.2 (aligning with Table 5.1). The rows show the update conditions. Each cell depicts the announcements 🔊, markings 📍, and locks 🔒 stored on both components at the given time interval when using the respective update condition. They further show whether a component can be updated safely, only for non-essential changes, or the update is unsafe.

Time interval A starts after BeforeWorkflow completes and ends before BeforeEachTask starts for the first time. All other time intervals are during the execution of the respective task.

As an example, the bottom row shows Essential Safety. The first entry illustrates that at time interval A, both the holiday and the car rental component hold an announcement for the workflow instance *I* executed on workflow engine *r*. Both components can be safely updated. In the second entry at time interval B, the holiday component is additionally marked and locked. Hence, it cannot be safely updated. At time interval C, the holiday component can be updated in case the update is a non-essential change because it holds no lock. However, an essential update would be unsafe because it holds a marking *and* an announcement of *I*.

Quiescence exhibits the fewest safe update intervals because it does not consider run-time information on workflow instances. We assume that no new workflow instance is started after the one in Figure 5.2. Otherwise, there would be no safe update interval for Quiescence at all. In contrast, the safe update intervals for the other approaches remain unchanged without such an assumption.

Tranquility features all safe update intervals of Version Consistency. Plus, it permits intervals C and c for the holiday component (✓̶ in Table 5.1), which cannot be considered generally safe if the component does not respect the black-box principle. For instance, if the holiday component is updated at these intervals in a way that it writes and reads the hotel booking id in another format to/from workflow state *S*, “cancel hotel” could fail.

Essential Safety features all safe update intervals of Version Consistency for essential changes. For non-essential changes, it provides the most safe update intervals of all discussed safe DSU approaches.

## 5.6 Evaluation

In this section, we empirically evaluate dynamic software updating for long-running and highly-frequent workflows with existing approaches, as well as our solution Essential Safety. We aim to answer the following research questions.

**RQ 5.1** Can safe DSU be adopted in real-world collaborative workflow applications? With this question, we empirically investigate whether, with our model for safe DSU in workflows (Section 5.3.1), the approaches discussed in Sections 5.3.2 and 5.5.2 can be applied to real-world collaborative workflow applications.

**RQ 5.2** Does Essential Safety significantly reduce the performance overhead of safe DSU in realistic workflows? This question empirically investigates the performance differences among the DSU approaches discussed analytically in Section 5.5.4. Specifically, we assess whether Essential Safety significantly reduces the impact of updating realistic workflow applications.

**Table 5.3:** Construction statistics of the realistic collaborative BPMN workflows dataset based on RePROSitory [54].

| Workflow Collection | Size | Recovered | Unchanged |
|---|---|---|---|
| RePROSitory | 572 | | |
| Collaborative workflows | 135 | | |
| Stuck | 37 | 22 | |
| Endless loop | 9 | 6 | |
| Incomplete | 1 | 0 | |
| Missing internals | 10 | 0 | |
| Evaluation dataset | 106 | 28 | 78 |

**RQ 5.3** How does the share of non-essential changes impact the performance of Essential Safety? This question investigates to which degree Essential Safety's performance depends on the amount of non-essential changes, estimating the share of non-essential changes that is sufficient to achieve better performance than the previous approaches. With RQ 5.4, RQ 5.3 validates the assumptions motivating Essential Safety.

**RQ 5.4** How frequent are non-essential changes in multi-component software systems? This question verifies the hypothesis behind Essential Safety—that most updates are non-essential—ensuring the generalizability of our results. Such evidence is required to determine the significance and applicability of our approach to real-world workflows.

### 5.6.1 Applicability of Safe DSU to Workflows

We now evaluate whether our model and the safe DSU approaches are applicable to workflows. For this, we constructed a dataset of real-world BPMN workflows and implemented a discrete event-based simulation for safe DSU in workflows using the dissemination algorithm (Section 5.4.1). We assessed Essential Safety (Section 5.3) and the other safe DSU approaches (Section 5.5.2). The simulation and all scripts and data are open-sourced and long-term archived on Zenodo [237].

There is no standard benchmark for realistic workflow models, possibly due to their complexity and business relevance [223, 224]. RePROSitory [54] is a database of realistic BPMN workflows. Based on a full copy from August 3, 2021, we constructed an evaluation dataset containing 106 collaborative BPMN workflows (Table 5.3). We selected all collaborative workflows with two or more BPMN lanes or pools—the BPMN elements that assign process elements to collaboration participants—which we interpret as workflow components. Everything outside any lane or pool is a separate component.

**Table 5.4:** Parameter distributions of the discrete-event simulation of safe DSU.

| Parameter | Distribution |
|---|---|
| Network latency | Weibull: $\alpha = 1.5$, $\beta = 30\,\text{ms}$ ($\mu = 27.1\,\text{ms}$, sd $= 18.4\,\text{ms}$) |
| Instances per workflow | Weibull: $\alpha = 1.5$, $\beta = 20\,160$ ($\mu = 18\,199$, sd $= 12\,357$) |
| Avg. task duration | Weibull: $\alpha = 1.5$, $\beta = 2\,\text{min}$ ($\mu = 108.3\,\text{s}$, sd $= 73.6\,\text{s}$) |
| Task duration | Gaussian: sd $= 10\,\% \cdot \mu$ |
| Avg. update interval | Gaussian: $\mu = 12\,\text{h}$, sd $= 4\,\text{h}$, min $= 1\,\text{h}$, max $= 24\,\text{h}$ |
| Avg. update duration | Weibull: $\alpha = 1.5$, $\beta = 5\,\text{min}$ ($\mu = 4.5\,\text{min}$, sd $= 3.1\,\text{min}$) |
| Update duration | Gaussian: sd $= 20\,\% \cdot \mu$ |

Accordingly, all workflows have tasks on at least two different components. 57 workflows are not executable because they get stuck, include endless loops, are incomplete, or only contain the internal workflow of one lane or pool. We manually recovered 28 of these with minimal changes.[2]

Table 5.4 provides the simulation parameter distributions. All parameters were chosen with the intent to be as realistic as possible. Interarrival parameters and durations are Weibull-distributed with $\alpha = 1.5$, commonly used for Internet-based traffic simulations [17]. Update intervals and task durations were Gaussian-distributed, simulating regular CI/CD executions and tasks with predictable, roughly constant execution time, which is common in business applications. The workflow instances were distributed over ten workflow engines. For each workflow, the number of instances was Weibull-distributed with, on average, one invocation every 66 s. 99.7 % of the components were updated between once per day and once per hour, which were fixed limits of the mean update frequency. We drew the points in time for starting workflow instances and triggering component updates from a uniform distribution over the simulation timespan of two weeks. 90 % of the updates were non-essential changes. We performed a sensitivity analysis with both double and half the value of each parameter using the trip booking saga (Section 5.2.1). The plots only confirm obvious correlations, e.g., halving the task duration increases updatability and decreases workflow duration and update time, and are reported in the evaluation artifact [237].

We successfully simulated all 106 workflows for the safe DSU approaches in Sections 5.3.2 and 5.5.2 and the reaching strategies (Section 5.3.3). This result positively answers RQ 5.1, showing that safe DSU can be applied to real-world collaborative workflow applications.

[2] All exclusions and adjustments are documented in the dataset's build script [237].

### 5.6.2 Performance of Essential Safety

We now investigate the performance differences of the DSU approaches in the simulation introduced before. Table 5.5 and Figure 5.6 compare the updating approaches (Sections 5.3.2 and 5.5.2) and reaching strategies (Section 5.3.3) with the baseline "No Updates" where no updates are performed. All simulations were executed on the same trace of workflow instance executions and updates. *Updatability* is the overall time the update condition is met at the components (i.e., when updating is safe). The *update time* is the timespan from triggering to completing a component update. It is split into the *update timeliness* (until the update condition is met) and the *update duration* (after the update condition is met). The *workflow duration* is the timespan between the start and completion of a workflow instance. The *workflow delay* is the difference between the workflow instances' start and its start in the baseline. Analogously, *workflow interruption* is the difference in the instances' completion times, measured by the sum of the instances' delays and duration differences. For Essential Safety, we also report the metrics separately for essential and non-essential changes, a distinction that does not lead to different results for the other approaches.

Among all approaches, we found the least performance impact on updates and workflow instances with Tranquility. However, Tranquility is generally unsafe—in contrast to all other approaches. Further, Tranquility with Blocking Tasks was the only simulation with deadlocked workflow instances (47 %), preventing their completion. These deadlocks positively skewed the averages of all metrics for Tranquility[3] because deadlocked workflow instances are excluded from the measurement data. We observed that Essential Safety performance similarly to Version Consistency for essential changes and slightly better performance than Tranquility for non-essential changes. Overall, Essential Safety's performance is similar to Tranquility but retains update safety. On average, Essential Safety's workflow interruption was 5.0 %, and it provided 8.0 % higher updatability, 21 % less update time, and 48 % less workflow interruption than Version Consistency—the best, safe competitor.

The reaching strategies' relative performance trends were similar among the updating approaches. All strategies added only a small delay to the workflows. Version Consistency exhibited no delay at all. Blocking Instances entailed the highest delays. For Concurrent Versions, the update timeliness was similarly low for all updating approaches, whereas the update duration exhibited some variability. Vice versa, all other reaching strategies

[3] Due to the deadlock skew, Table 5.5 also reports Blocking Tasks without Tranquility.

**Table 5.5:** Performance metric means for all safe DSU approaches and reaching strategies over all simulations in minutes.

| Approach / Strategy | Updatability | Update | | Workflow | |
|---|---|---|---|---|---|
| | | Duration | Timeliness | Duration | Delay |
| No Updates (Baseline) | | | | 15.9 | 0.0 |
| Quiescence | 2 228.7 | 9.1 | 1 384.4 | 16.6 | 0.7 |
| Version Consistency | 4 510.5 | 8.2 | 518.0 | 16.7 | 0.8 |
| Tranquility | 902.7 | 8.1 | 184.7 | 15.5 | 0.5 |
| Essential Safety (ES) | 4 870.6 | 5.1 | 409.5 | 16.3 | 0.4 |
| ES: essential changes | 4 384.7 | 8.0 | 541.8 | | |
| ES: non-ess. changes | 4 924.6 | 4.8 | 395.4 | | |
| Blocking Instances | 4 219.3 | 4.3 | 11.4 | 17.4 | 1.5 |
| Blocking Tasks (BT) | 5 309.2 | 4.4 | 785.2 | 16.1 | 0.7 |
| BT w/o Tranquility | 4 651.5 | 4.4 | 6.3 | 16.7 | 0.8 |
| Concurrent Versions | 3 744.6 | 16.3 | 0.5 | 15.9 | 0.0 |
| Waiting | 4 330.0 | 4.3 | 2 402.3 | 16.1 | 0.2 |

exhibited similar update durations but variable update timeliness. This difference was due to Concurrent Versions running two component versions in parallel, eliminating the workflow interruption because workflow instances never waited for component updates. However, Concurrent Versions requires that the components' implementations support running two different versions in parallel, which our simulation assumed is possible. If not supported by the components' implementations, Concurrent Versions must not be used. Instead, Blocking Tasks caused the least impact on component updates in such cases. Though Waiting delayed workflow instances to a lesser extent, it heavily delayed updates—on average by 40.0 hours in our simulations.

Our results answer RQ 5.2 and show that the impact of safe DSU can be significant. For Essential Safety, the impact of non-essential changes is completely negligible, and the impact of essential changes is not higher than that of previous approaches. For a realistic changeset, Essential Safety significantly decreases the overhead of safe DSU compared to the state-of-the-art.

### 5.6.3 Effect of Non-Essential Updates

To evaluate whether distinguishing essential and non-essential changes is effective—the assumption behind Essential Safety—we repeated the previous simulation with different ratios of non-essential to essential changes. Figure 5.7 shows the metrics from Section 5.6.2 for Essential Safety with only essential changes (0 %), only non-essential

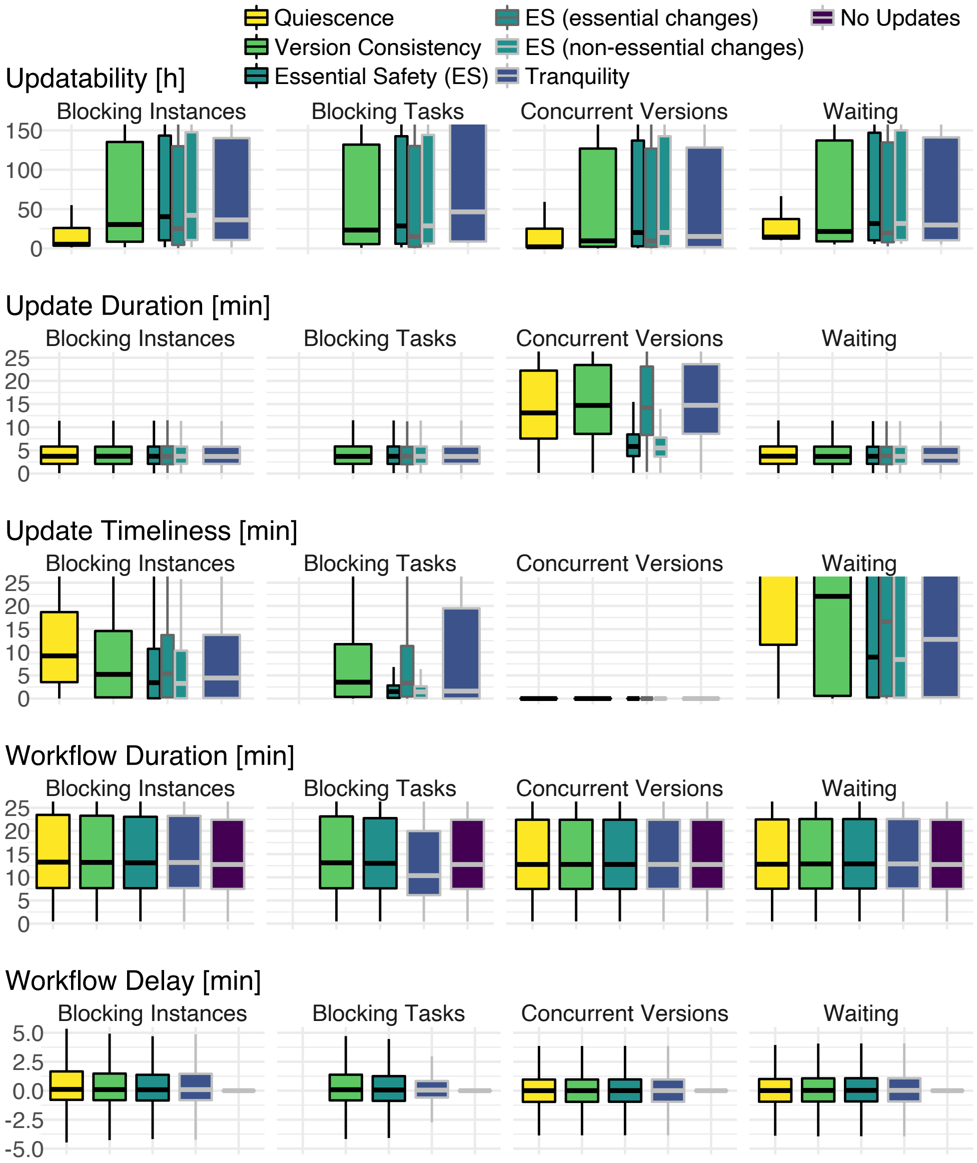
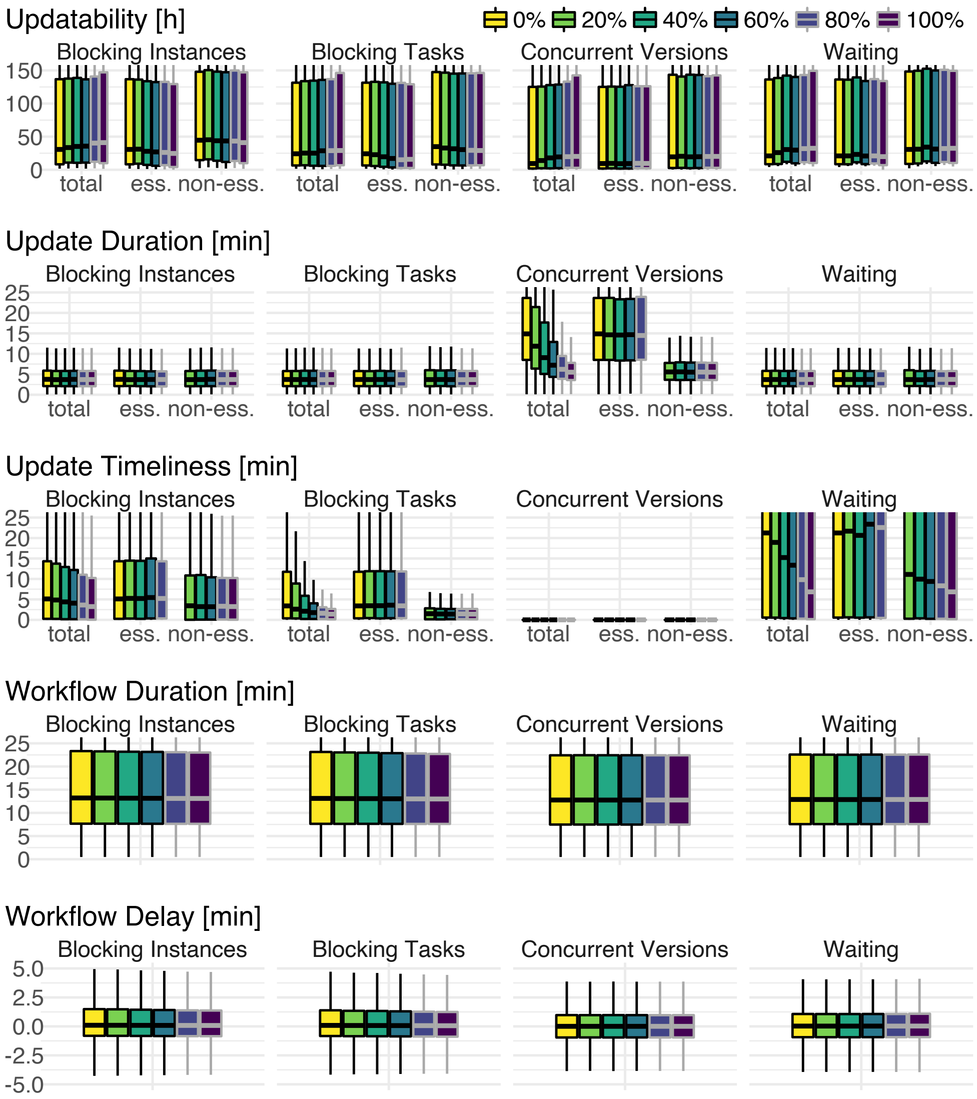


**Figure 5.6:** Performance comparison of the safe DSU approaches.

changes (100 %), and all ratios in between in steps of 20 % points. The results are presented separately for essential (ess.) and non-essential changes (non-ess.) as well as combined (total).

In total, the updatability increased with the share of non-essential changes; on average, 9.5 % from 0 % to 100 % non-essential changes. The update time was reduced, on average, by up to 52.2 % and the workflow interruption by 54.8 %.

We now consider essential and non-essential changes separately. For all metrics and approaches, the results were similar, i.e., independent from the share of non-essential changes, except for the following cases: For Blocking Instances and Blocking Tasks, the updatabilities slightly decreased with the increasing share of non-essential changes



ess.: essential changes; non-ess.: non-essential changes; total: all changes

**Figure 5.7:** Performance of Essential Safety for various shares of non-essential changes.

because the likelihood that multiple updates are reached jointly and executed as a batch was lower. This increased blocking times to reach safe update intervals. For the same reason, the timeliness of essential change updates was worse for Waiting.

The results answer RQ 5.3: The higher the share of non-essential changes, the smaller Essential Safety's overall performance impact. Thus, considering non-essential changes is crucial and effective to reduce the overhead of safe DSU for workflows.

### 5.6.4 Frequency of Non-Essential Updates

To answer RQ 5.4, we focused on open-source software repositories and assumed that they use a continuous deployment pipeline. In continuous deployment, every commit

**Table 5.6:** Number of affected components per commit in 8 open-source monorepos.

| Monorepo | Average number of components affected by a commit | Average share of components affected by a commit |
|---|---|---|
| StartupOS | 13 | 13 % - 41 % |
| Foursquare Fsq.io | 13 | 8 % - 38 % |
| M3 | 22 | 8 % - 20 % |
| Celo | 23 | 7 % - 10 % |
| Berty | 31 | 8 % - 32 % |
| Stellar Go | 41 | 3 % - 4 % |
| Habitat | 49 | 5 % - 12 % |
| Nixpkgs | 810 | <1 % - 15 % |

may trigger the deployment of an update. Identifying which components are affected by each commit requires application knowledge and cannot be easily automated. Hence, for simplicity, a common practice is to redeploy all components, even though one can easily hypothesize that a subset suffices. To assess this hypothesis, we focused on repositories that aggregate various software components. Such “monorepos” are widely used [88, 138, 150, 176, 179]. Typically, the degree to which components in them depend on each other varies, allowing us to identify the subset of components that a commit changed.

We investigated eight monorepos that were publicly available on GitHub and described in [53]. In the monorepos, each component is encapsulated in its own directory. We identified the directories that contain a component based on the repository description. We explored each repository’s most recent 10 000 commits to determine how many components were affected by each commit. We assumed that a component changed if a commit modified a file in the component’s directory. For commits that changed files not associated with any component, we considered a conservative approximation (upper bound), that the commits affected every component, and a speculative approximation (lower bound), that they did not affect any. We ignored changes to tests, documentation, and hidden directories.

Table 5.6 shows for each monorepo the absolute number and the percentage of affected components (mean over all commits). On average, even under the conservative approximation, a commit affected less than half of the components. Under the speculative approximation, commits affected less than 10 % of the components in most monorepos. Accordingly, at least 60 %–90 % of the component updates were non-essential changes.

So far, we have demonstrated how often commits were non-essential component changes because the commits did not change a component’s code. Additionally, not all code changes introduced semantic changes, i.e., they were non-essential for *all* components. Such commits further reduce the observed numbers in Table 5.6. Previous studies provided evidence that the amount of such non-essential changes is significant: (1) Kawrykow and Robillard [124] analyzed seven open-source Java systems, finding that up to 15.5 % of the method updates were cosmetic, behavior-preserving, or unlikely to provide further insight into component relationships. (2) Based on the TravisTorrent dataset [28], Abdalkareem et al. [1] found that 10 % of the commits developers manually skipped in CI/CD pipelines were skipped because they were non-essential changes, i.e., they only touched documentation, source code comments, formatting of source code, meta files, or were code release preparations.

These results answer RQ 5.4: On average, 60 % of the component changes are non-essential as a lower bound, while we realistically assume a considerably higher percentage of over 90 %.

## 5.7 Conclusion

Traditionally, software updates require shutting down the system before replacing any component. To avoid service disruption, safe DSU techniques ensure that components can be replaced safely while the overall system runs. Unfortunately, existing safe DSU approaches introduce a significant performance overhead, and it is unclear how to apply them to workflow-based systems in decentralized organizations.

To close this gap, we propose a unified formal model for safe DSU in workflows suitable for decentralized setups. We show how it captures state-of-the-art DSU approaches and compare them analytically with Essential Safety, our novel safe DSU solution. Essential Safety leverages the identification of updates that have no semantic changes—non-essential changes—effectively reducing safe DSU's performance overhead. Based on our safe DSU model, we propose a modular dissemination algorithm as a plugin to workflow engines, enabling the application of the discussed safe DSU approaches in workflow-based systems of decentralized organizations. Further, we show how $\mu$S' novel runtime behavior and our dissemination algorithm allow developers to implement safe DSU directly in IaC programs, enabling holistic reasoning about the deployment and its behavior in one place, which is beneficial for its reliability. The empirical evaluation with 106 realistic collaborative BPMN workflows and eight monorepos confirms that we enable efficient, safe dynamic software updating for decentralized organizations with long-running and frequently executed workflows.

## Chapter 6

# A Dataset of IaC Programs[1]

In this chapter, we present PIPr, the first PL-IaC dataset. PIPr enables urgently needed studies on IaC programs to understand how existing software engineering techniques can be effectively applied to IaC programs and where novel reliability techniques need to be developed. Our initial analysis shows that IaC programs rarely implement tests, indicating the lack of suitable testing techniques that we address in Chapter 7.

We motivate the dataset creation and analysis in Section 6.1 and summarize related IaC datasets in Section 6.2 before describing the construction of PIPr in Section 6.3. As initial analyses, we inspect the IaC programs for their (1) programming languages (Section 6.4.1), (2) testing techniques (Section 6.4.2), and (3) licenses (Section 6.4.3). Section 6.5 discusses limitations and threats to validity, and Section 6.6 concludes.

## 6.1 Motivation and Research Questions

In the previous chapters, we enhanced the state of PL-IaC by introducing new code and runtime features, enabling IaC programs to express and automate coordination requirements across deployments. Still, being able to express and automate all requirements is not enough to achieve reliability. The code developers write in IaC programs must be correct, too, requiring quality assurance techniques. As IaC programs use the same surface languages as traditional software, there is great potential to apply existing software engineering methods, e.g., testing, verification, and static analysis. Yet, in general, we know little about IaC programs and their differences from other software. Such insights are imperative to apply existing techniques effectively and to develop new approaches that leverage the peculiarities of this domain. Further, there is no systematic PL-IaC dataset on which researchers could study such questions.

[1] Based on the authors' work in [232].

To build ground for closing this gap, we present PIPr, the first systematic dataset of public IaC programs. PIPr comprises metadata of 37 712 IaC programs from 21 445 public GitHub repositories and shallow copies (i.e., without history) of the ones permitting redistribution. The dataset enables studies on PL-IaC in practice and the differences and similarities between such programs and traditional software, which is crucial to transfer existing software engineering techniques and develop new ones optimized for PL-IaC. As the initial analysis, we use it to answer the following research questions, characterizing IaC programs today and shedding light on their application of existing quality assurance techniques, specifically testing.

**RQ 6.1** Which programming languages are used in public IaC programs?

**RQ 6.2** Which testing techniques are employed in public IaC programs?

**RQ 6.3** Which licenses are applied to public IaC programs?

## 6.2 Related Datasets

All PL-IaC solutions provide example programs.[2][3][4] They are public on GitHub and have explicit open-source licenses. PIPr contains their metadata and code from August 2022. Beyond these, the only datasets of IaC programs we know were created by us to evaluate *µ*S [233]. They contain 64 Pulumi TypeScript programs using stack references [239] and simple benchmarking programs [241] in Pulumi TypeScript, AWS CDK TypeScript, and µs. We are not aware of other research analyzing IaC programs for PL-IaC solutions.

Various studies examined IaC scripts for Ansible, Chef, and Puppet. Most published analysis scripts and results, but not all analyzed IaC scripts [30, 58, 106, 113, 171, 172, 173, 175, 190, 196, 197, 201, 204, 221]. In contrast, Sotiropoulos et al. [243] provided the 33 studied Puppet scripts, and Saavedra and Ferreira [207] published a comprehensive dataset of 108 509 Ansible, 70 939 Chef, and 17 037 Puppet scripts, containing unpublished IaC scripts of earlier studies [196, 201]. Further, Opdebeeck et al. [171] built a dataset of Ansible Galaxy ecosystem metadata, including abstract structural representations of over 125 000 Ansible roles and 800 000 changes.

PIPr is the first systematic dataset for PL-IaC. It provides the code of 15 504 redistributable IaC Programs and metadata of all 37 712 IaC programs we found on GitHub in

[2] https://github.com/aws-samples/aws-cdk-examples

[3] https://github.com/pulumi/examples

[4] https://developer.hashicorp.com/terraform/cdktf/examples-and-guides/examples

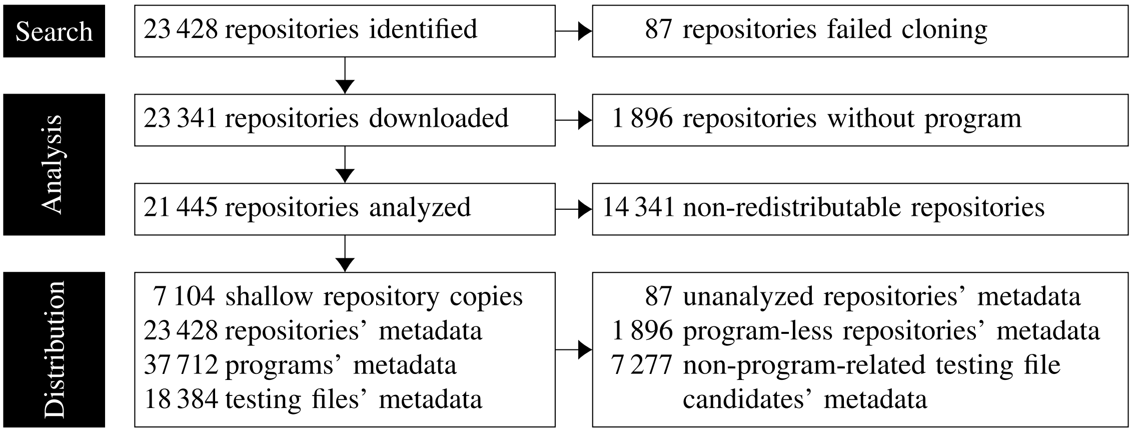

Search
23 428 repositories identified
87 repositories failed cloning
Analysis
23 341 repositories downloaded
1 896 repositories without program
21 445 repositories analyzed
14 341 non-redistributable repositories
Distribution
7 104 shallow repository copies
23 428 repositories' metadata
37 712 programs' metadata
18 384 testing files' metadata
87 unanalyzed repositories' metadata
1 896 program-less repositories' metadata
7 277 non-program-related testing file candidates' metadata

**Figure 6.1:** Flow diagram quantifying the creation of the PIPr dataset.

August 2022. Further, it includes our analysis results for the programs' programming languages, testing techniques, and licenses.

## 6.3 Dataset Construction

Figure 6.1 provides an overview of PIPr's construction, analysis, and distribution, which we now describe in detail.

### 6.3.1 Repository Identification

We searched for platforms hosting AWS CDK, Pulumi, and CDKTF programs—the only three established PL-IaC solutions. We chose GitHub as the data source of PIPr because we did not find another platform publicly hosting many IaC programs—even AWS CDK, Pulumi, and CDKTF themselves are publicly maintained on GitHub.

We identified GitHub repositories with IaC programs by searching for IaC program configuration files. By design, in PL-IaC solutions, each IaC program has one configuration file named `cdk.json`, `cdktf.json`, `Pulumi.yml`, or `Pulumi.yaml`. Munaiah et al. [163] summarized techniques to query GitHub, including Boa [69] and the discontinued GHTorrent [93]. We used the GitHub REST Code Search API [85]—despite having to cope with its severe limitations that we will discuss next—because it is the only solution that allows up-to-date searches for file names across GitHub.

Our study addresses the API's limitations [85]. First, the API only returns files in repositories that (i) are not a fork or a fork with more stars than their parent, (ii) have fewer than 500 000 files, and (iii) saw activity or were returned in search results in the last year. Further, the searched files must be (iv) on the default branch (v) and smaller than 384 KB. Inheriting these criteria ensures we do not analyze irrelevant (i, iii, iv) and technically not tractable (ii, v) repositories. Second, the API's results are incomplete and unstable: it (a) only returns up to 1 000 results per query, (b) returns varying query result counts across responses, and (c) returns varying results for the same result page that are often fewer than expected and repeating results from previous pages. To address (a), we recursively divided the queries by file size until each sub-query has at most 990 results; for (b), we only accepted a changed result count if we received it five consecutive times; for (c), we requested the pages with default size (max. 30 results) up to 200 times until the combined responses contained the expected number of *new* results.

The search started on August 16, 2022, and was distributed over two GitHub accounts. It required two weeks due to retrying upon incomplete results and API rate limiting. We recorded the metadata for all 23 428 identified repositories. We downloaded a shallow recursive copy of each repository's default branch using `git clone` on August 31 and September 1, 2022. 87 repositories could not be downloaded (one retry), most of them due to missing permissions.

### 6.3.2 IaC Program Identification

In the analysis, we identified all IaC programs in the downloaded repositories. We searched all `cdk.json`, `cdktf.json`, `Pulumi.yml`, and `Pulumi.yaml` files that are not in a `node_modules` folder, yielding 39 255 files. 105 are not parsable, and 892 do not contain a certain set of fields, i.e., they do not define the IaC program's runtime, which is the only information PL-IaC solutions require to run an IaC program. For AWS CDK, this means there is no `app` field; for Pulumi, there is no `runtime` field; and for CDKTF, there is neither `app` nor `language`. Further, we removed 469 files whose path contains a string from a denylist of 28 entries. The list was manually created to identify dependency and PL-IaC solution implementation paths and 77 files due to implausible runtime values. For the remaining 37 712 IaC programs, we extracted the runtime, the PL-IaC solution, and, if present, the program name and description.

### 6.3.3 Distribution

PIPr, including all scripts, is long-time archived under the Open Data Commons Attribution License (ODC-By) v1.0 on Zenodo [230]. Figure 6.2 describes the schema of the CSV files `repositories.csv`, `programs.csv`, and `testing-files.csv`, which contain the metadata of:

- 23 428 repositories (Section 6.3.1), of which 87 failed to download, and in 21 445 (1 896), we found (not) an IaC program (Section 6.3.2).
- 37 712 IaC programs (Section 6.3.2) with their programming language (Section 6.4.1), applied testing techniques (Section 6.4.2), and licenses (Section 6.4.3).
- 18 384 testing file candidates (Section 6.4.2), of which we identified a testing technique in 13 631 files and related 11 107 files to an IaC program.

Further, PIPr contains copies of 7 104 repositories with IaC programs whose licenses permit redistribution, resulting in 58 GB of code. Lastly, we release all creation and analysis scripts, execution logs, and additional documentation.

## 6.4 Initial Analysis

We now analyse PIPr to answer the research questions posed in the beginning of this chapter, characterizing the dataset's IaC programs and showcasing the use of PIPr for studies on PL-IaC.

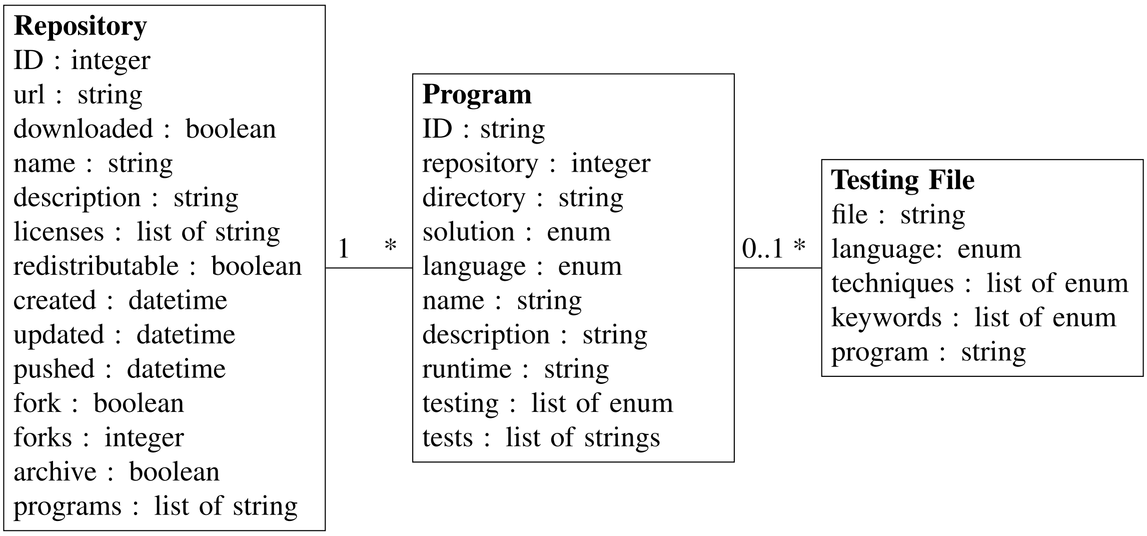
### 6.4.1 Languages of IaC Programs

To answer RQ 6.1, "Which programming languages are used in public IaC programs?", we map each IaC program's runtime configuration value to a programming language using regular expressions. For less popular languages where the IaC program reuses the runtime of another language, we assign the program to the main language of the runtime because identifying them based on the runtime configuration is not reliable and would lead to additional insignificant minorities with low confidence in the results (Table 6.1). We map all non-TypeScript NodeJS, e.g., CoffeeScript and Scala.js, programs to JavaScript, JVM programs to Java, and .NET programs to C#.

Table 6.1 summarizes the results. TypeScript is, with 56 % (21 245) of the IaC programs, by far the most popular language across all PL-IaC solutions. Python is popular, too, with 23 % (8 610). Further, there is a significant amount of Pulumi C# and Go programs (each 14 % of the Pulumi programs) and AWS CDK JavaScript and Java programs (8 % and 4 % of the AWS CDK programs).

### 6.4.2 Testing Techniques of IaC Programs

To answer RQ 6.2, "Which testing techniques are employed in public IaC programs?", we consider the available PL-IaC testing techniques discussed in Section 2.2.4. All PL-IaC



**Solutions:** AWS CDK | CDKTF | Pulumi

**Languages:** csharp | go | haskell | java | javascript | python | typescript | yaml

**Testing & Techniques:** awscdk | awscdk_assert | awscdk_snapshot | cdktf | cdktf_snapshot | cdktf_tf | pulumi_crossguard | pulumi_integration | pulumi_unit | pulumi_unit_mocking

**Keywords:** /go/auto | /testing/integration | @AfterAll | @BeforeAll | @Test | @aws-cdk | @aws-cdk/assert | pulumi.runtime.test | @pulumi/ | @pulumi/policy | @pulumi/pulumi/automation | Amazon.CDK | Amazon.CDK.Assertions | Assertions_ | HashiCorp.Cdktf | IMocks | Moq | NUnit | PolicyPack( | ProgramTest | Pulumi | Pulumi.Automation | PulumiTest | ResourceValidationArgs | ResourceValidationPolicy | SnapshotTest() | StackValidationPolicy | Testing | Testing_ToBeValidTerraform( | ToBeValidTerraform( | Verifier.Verify( | WithMocks( | [Fact] | [TestClass] | [TestFixture] | [TestMethod] | [Test] | afterAll( | assertions | automation | aws-cdk-lib | aws-cdk-lib/assert | aws_cdk | aws_cdk.assertions | awscdk | beforeAll( | cdktf | com.pulumi | def test_ | describe( | github.com/aws/aws-cdk-go/awscdk | github.com/hashicorp/terraform-cdk-go/cdktf | github.com/pulumi/pulumi | integration | junit | pulumi | pulumi.runtime.setMocks( | pulumi.runtime.set_mocks( | pulumi_policy | pytest | setMocks( | set_mocks( | snapshot | software.amazon.awscdk.assertions | stretchr | test( | testing | toBeValidTerraform( | toMatchInlineSnapshot( | toMatchSnapshot( | to_be_valid_terraform( | unittest | withMocks(

**Figure 6.2:** Schema of the PIPr metadata and results.

solutions support unit testing [14, 105, 188]. Additionally, Pulumi features policy testing with CrossGuard [184] and an integration testing library [182]. All other PL-IaC testing techniques we know are ad-hoc and do not lead to additional configuration or code, e.g., dry-running Pulumi programs and manual end-to-end testing.

Our method to identify testing files in repositories is similar to other studies [56, 153, 163]. We created a list of 34 keywords that are specific to PL-IaC testing code (e.g., `@aws-cdk/assert` and `setMocks()`, 14 common keywords in PL-IaC code (e.g., `@aws-cdk` and `@pulumi/`), and 26 common keywords in testing code (e.g., `describe(` and `test()`. Potential PL-IaC testing files are those that contain at least one PL-IaC-

**Table 6.1:** Public IaC programs on GitHub by solution and language.

| Language | Pulumi | AWS CDK | CDKTF | Total |
|---|---:|---:|---:|---:|
| TypeScript | 6 081 | 14 639 | 525 | 21 245 |
| Python | 2 927 | 5 521 | 162 | 8 610 |
| C# | 1 835 | 563 | 28 | 2 426 |
| Go | 1 834 | 338 | 73 | 2 245 |
| JavaScript | 35 | 1 844 | 5 | 1 884 |
| Java | 75 | 1 035 | 34 | 1 144 |
| YAML | 157 | 0 | 0 | 157 |
| Haskell | 1 | 0 | 0 | 1 |
| Total | 12 945 | 23 940 | 827 | 37 712 |

testing-specific keyword or one out of 72 combinations of the remaining keywords lines that do not start with # or // as the first non-whitespace characters (i.e., that are comment lines) as well as all files that are named `PulumiPolicy.yaml`. We found 65 156 files and extracted their path and keywords. We manually inspected this data and the full content of some files to (1) identify which file extensions to ignore, (2) validate and apply the file path filtering we applied in Section 6.3.2, and (3) create a function that maps to a testing technique the remaining 18 384 files based on their keywords and file extensions. We identified a testing technique in 13 631 files and related 11 107 files to an IaC program based on their file path by matching the nearest IaC program in a parent folder.

Table 6.2 summarizes the results. For each technique, it shows the number of files in PIPr and IaC programs containing one. We report adoption in absolute numbers and relative to all programs of the respective PL-IaC solution and language, e.g., 1 % (118) of the Pulumi programs use unit testing, 51 of them are in TypeScript (1 % of the Pulumi TypeScript programs), of which 38 use runtime mocking. Only 25 % of the IaC programs implement tests. Further, testing is far more common for CDK programs (38 % for AWS CDK and 15 % for CDKTF). Only 1 % of the Pulumi programs implement tests.

### 6.4.3 Licenses of IaC Programs

To answer RQ 6.3, “Which licenses are applied to public IaC programs?”, we applied Licensee [144] to all repositories. We chose Licensee because GitHub recommends and uses it [86]. Licensee analyses all files commonly containing license information, e.g., `LICENSE` and `README`, for license content or references to licenses, using the license database of https://choosealicense.com/.

**Table 6.2:** Number of IaC programs in PIPr applying testing techniques in total and by language. Number of programs (% of programs in group).

| | Testing Technique | Files | Total | | TypeScript | | Python | | C# | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | Go | | JavaScript | | Java | |
| Pulumi | Unit Testing | 259 | 118 | (1 %) | 51 | (1 %) | 27 | (1 %) | 22 | (1 %) |
| | | | | | 15 | (1 %) | 0 | (0 %) | 3 | (4 %) |
| | with Runtime Mocking | 149 | 100 | (1 %) | 38 | (1 %) | 26 | (1 %) | 20 | (1 %) |
| | | | | | 15 | (1 %) | 0 | (0 %) | 1 | (1 %) |
| | CrossGuard | 399 | 33 | (0 %) | 29 | (0 %) | 4 | (0 %) | 0 | (0 %) |
| | | | | | 0 | (0 %) | 0 | (0 %) | 0 | (0 %) |
| | Integration Testing | 677 | 22 | (0 %) | 12 | (0 %) | 8 | (0 %) | 0 | (0 %) |
| | | | | | 2 | (0 %) | 0 | (0 %) | 0 | (0 %) |
| AWS CDK | Unit Testing | 12 102 | 9 152 | (38 %) | 7 116 | (49 %) | 1 141 | (21 %) | 12 | (2 %) |
| | | | | | 151 | (45 %) | 328 | (18 %) | 404 | (39 %) |
| | with AWS CDK Assertions | 10 436 | 8 161 | (34 %) | 6 967 | (48 %) | 772 | (14 %) | 11 | (2 %) |
| | | | | | 40 | (12 %) | 320 | (17 %) | 51 | (5 %) |
| | with Snapshot Testing | 1 338 | 819 | (3 %) | 788 | (5 %) | 10 | (0 %) | 0 | (0 %) |
| | | | | | 0 | (0 %) | 13 | (1 %) | 8 | (1 %) |
| CDKTF | Unit Testing | 194 | 121 | (15 %) | 81 | (15 %) | 21 | (13 %) | 4 | (14 %) |
| | | | | | 10 | (14 %) | 0 | (0 %) | 5 | (15 %) |
| | with Snapshot Testing | 80 | 36 | (4 %) | 29 | (6 %) | 1 | (1 %) | 2 | (7 %) |
| | | | | | 2 | (3 %) | 0 | (0 %) | 2 | (6 %) |
| | with Terraform Compatibility | 23 | 23 | (3 %) | 14 | (3 %) | 1 | (1 %) | 1 | (4 %) |
| | | | | | 6 | (8 %) | 0 | (0 %) | 1 | (3 %) |
| Total | | 13 631 | 9 435 | (25 %) | 7 284 | (34 %) | 1 196 | (14 %) | 38 | (2 %) |
| | | | | | 177 | (8 %) | 328 | (17 %) | 412 | (36 %) |

We did not find a license in 67 % (14 330) of the repositories. The most popular are MIT and Apache 2.0, which are used by 3 545 (17 %) and 1 988 (9 %) repositories. Only 11 repositories prohibit redistribution explicitly. In PIPr, we redistribute all repositories with an IaC program that have at least one and only licenses permitting redistribution. Table 6.3 shows the redistributed IaC programs by PL-IaC solution, language, and how many use testing, e.g., we provide 51 % (3 084) of the Pulumi TypeScript programs; 53 of them test.

## 6.5 Limitations and Threats to Validity

PIPr is limited to the CDKs and Pulumi and the testing techniques provided by the PL-IaC solutions. We carefully researched other PL-IaC solutions and testing techniques but found none. Further, PIPr is based on a snapshot in August 2022 and does not contain historical data, limiting its direct use for longitudinal studies.

**Table 6.3:** Redistributable IaC programs by PL-IaC solution and language in PIPr. Number of programs with license permitting redistribution (% of programs with any license) of which #T use testing.

| Language | Pulumi | | AWS CDK | | CDKTF | | Total | |
|---|---|---|---|---|---|---|---|---|
| TypeScript | 3 084 (51 %) | 53 T | 5 131 (35 %) | 2 230 T | 401 (76 %) | 60 T | 8 616 (41 %) | 2 343 T |
| Python | 1 201 (41 %) | 18 T | 2 085 (38 %) | 264 T | 64 (40 %) | 11 T | 3 350 (39 %) | 293 T |
| C# | 633 (34 %) | 19 T | 299 (53 %) | 10 T | 13 (46 %) | 3 T | 945 (39 %) | 32 T |
| Go | 624 (34 %) | 7 T | 193 (57 %) | 88 T | 25 (34 %) | 7 T | 842 (38 %) | 102 T |
| JavaScript | 29 (83 %) | 0 T | 1 092 (59 %) | 115 T | 4 (80 %) | 0 T | 1 125 (60 %) | 115 T |
| Java | 49 (65 %) | 3 T | 442 (43 %) | 166 T | 16 (47 %) | 4 T | 507 (44 %) | 173 T |
| YAML | 119 (76 %) | 0 T | 0 (–) | 0 T | 0 (–) | 0 T | 119 (76 %) | 0 T |
| Haskell | 0 (0 %) | 0 T | 0 (–) | 0 T | 0 (–) | 0 T | 0 (0 %) | 0 T |
| Total | 5 739 (44 %) | 100 T | 9 242 (39 %) | 2 873 T | 523 (63 %) | 85 T | 15 504 (41 %) | 3 058 T |

The internal validity of studies on PIPr may be impacted by using the GitHub REST Code Search API, inheriting its inclusion criteria that eliminate old projects and forks. The API also prevents reproduction, as new data is continuously added and old, unpopular results are removed, on top of its reliability issues (cf. Section 6.3.1). Another threat is caused by identifying IaC programs based on project file names. To mitigate this issue, we ensured that the files are valid by parsing them and checking their content for plausibility. The identification of IaC programs may be impacted by our list of exclusion file path fragments to filter. This aspect also applies to the testing file identification for RQ 6.2, which is further threatened by keyword searches and manually selecting the keywords. Also, mapping testing files to projects based on their file paths may cause mistakes because we systematically miss if testing files are managed separately; however, we could not find that this is common. Relatedly, the internal validity for RQ 6.2 may be threatened because the project may still use the testing technique even if we did not find a testing file in an IaC program. The identification of the programming language in RQ 6.1 relies on regular expressions. The results are limited by mapping some languages to the primary language of their runtime, e.g., Scala to Java. Lastly, for RQ 6.3, we inherit the limitations and validity constraints of Licensee [144].

A threat to the external validity is that we only analyze public projects on GitHub, which may not be generally representative, e.g., for proprietary software. Further, PL-IaC solutions and their use evolve quickly, and we capture data up to August 2022 that was retrievable through the GitHub REST Code Search API, wherefore PIPr may not generalize to older and recent PL-IaC.

## 6.6 Conclusion

PIPr [230] is the first systematic dataset for PL-IaC. It is open-source and contains metadata of 37 712 public IaC programs on GitHub and the source code of the 15 504 IaC programs whose licenses permit redistribution. The metadata includes information about the used programming languages, applied testing techniques, and licenses. PIPr is a suitable basis for future studies on (1) PL-IaC itself, (2) its relation to IaC approaches, and (3) its relation to software in general. All three areas are vital to enhance the development of IaC programs, while (2) and (3) are imperative to applying existing and developing new techniques for IaC programs effectively. We present concrete ideas for future research starting from PIPr in Section 8.2.

Our initial analysis of PIPr shows that AWS CDK is the most widespread PL-IaC solution, with 63 % of all programs we found, followed by Pulumi with 34 %; CDKTF is negligible. TypeScript is by far the most popular programming language and is used in 56 % of the IaC programs, followed by Python (23 %). Further, there is a significant amount of C# and Go Pulumi IaC programs and JavaScript and Java AWS CDK programs. Notably, for most IaC programs, developers do not implement tests. For two-phase PL-IaC, there is still a fair amount of programs with testing code, 38 % for AWS CDK and 15 % for CDKTF. However, for Pulumi, the only general PL-IaC solution, we found tests only for 162 IaC programs, worthing only 1 % of the Pulumi IaC programs in PIPr.

## Chapter 7

# Automating IaC Program Testing[1]

In this chapter, we present ACT and ProTI, a novel automated unit testing approach for IaC programs, and ProTI, its implementation for Pulumi TypeScript. ACT embraces a high level of automation and reusable plugins for test generation and oracles, enabling efficient unit testing of IaC programs. With ProTI, developers can test IaC programs in hundreds of configurations in a short time, often without writing any testing code. This chapter addresses the research gap in efficient testing techniques for IaC programs, which is motivated by and evaluated on our contributions in Chapter 6.

We motivate this chapter and provide a running example in Section 7.1, which is used in Section 7.2 to establish the testing dilemma of PL-IaC. To enable efficient testing of IaC programs, we propose Automated Configuration Testing (ACT) in Section 7.3, an automated framework allowing developers to efficiently unit-test IaC programs, addressing the testing dilemma. Section 7.4 presents ProTI, our implementation of ACT for Pulumi TypeScript with a default generator and oracle, leveraging type information from Pulumi package schemas. Finally, Section 7.5 evaluates ACT and ProTI on all Pulumi TypeScript programs in PIPr and artificial benchmarks, and Section 7.6 concludes.

## 7.1 Motivation and Running Example

Testing IaC is an open research problem and critical in practice. For example, Rahman et al. [194] urge in their mapping study of IaC research for more work on testing, and Guerriero et al. [97] found that declarativity and "impossible testing" are the most mentioned differences between IaC and traditional software in 44 semi-structured interviews with senior developers. The lack of suitable testing techniques is especially apparent for PL-IaC: while studies found that more than 50 % of public software projects on GitHub use testing [154, 222], we found in Section 6.4.2 that only 25 % of the IaC programs use

[1] Based on the authors' work in [229] © 2024 IEEE.

Listing 7.1: Random Word Website (RWW): A Pulumi TypeScript program that deploys a static website on AWS S3 showing a random word [229] © 2024 IEEE.[2]

```
 7.1.1 import * as pulumi from '@pulumi/pulumi';
 7.1.2 import * as aws    from '@pulumi/aws';
 7.1.3 import * as random from '@pulumi/random';
 7.1.4
 7.1.5 const words = ['software', 'is', 'great'];
 7.1.6 const bucket = new aws.s3.Bucket('website', {
 7.1.7     website: { indexDocument: 'index.html' }
 7.1.8 });
 7.1.9 const range = { min: 0, max: words.length };
7.1.10 const rng = new random.RandomInteger('word-id', range);
7.1.11 rng.result.apply((wordId) => {
7.1.12     new aws.s3.BucketObject('index', {
7.1.13         bucket: bucket, key: 'index.html',
7.1.14         contentType: 'text/html; charset=utf-8',
7.1.15         content: '<!DOCTYPE html>' +
7.1.16                  words[wordId].toUpperCase()
7.1.17     });
7.1.18 });
7.1.19
7.1.20 export const url = bucket.websiteEndpoint;
```

testing, dropping to 1 % for Pulumi. We conjecture that the reason for Pulumi's even lower share is that it is the only solution that implements general PL-IaC, making testing even harder (Section 2.2.4). This chapter focuses on Pulumi because it provides general PL-IaC and is more open than the CDKs.

Before analyzing the available PL-IaC testing techniques to understand their shortcomings, we introduce the Random Word Website (RWW) in Listing 7.1 as a running example for this chapter. It defines the target state in Figure 7.1 and is an advancement of the SW example (Listing 1.1), deploying a static website on AWS S3 [9] that displays a word randomly selected from the array in Line 7.1.5 instead of a fixed text. Lines 7.1.6 to 7.1.8 define the S3 bucket, and Line 7.1.10 the `word-id` resource. It receives `range` (Line 7.1.9) as input configuration and is assigned to `rng`. After `word-id` is deployed, the deployment engine provides a randomly drawn number as the `result` field of the resource's output configuration. Such output configuration values are available as properties of the resource objects, in this case as `rng.result`. To access the value, `apply` (Line 7.1.11) registers a callback (Lines 7.1.11 to 7.1.18), which executes as soon as the random number is available. The number is used to select a word from the `words`

[2] For brevity, we omit the bucket's ownership controls, public access block, and policy resources that are required to allow public access from the Internet.

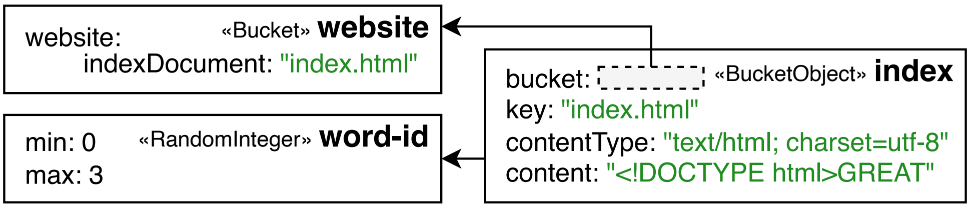


**Figure 7.1:** Example of a target state of RWW (Listing 7.1) [229] © 2024 IEEE.[2]

array in Line 7.1.16, which is capitalized and set as `content` in the input configuration of the `index` resource (Lines 7.1.12 to 7.1.17). The dependence of `index` on `word-id` is defined implicitly by defining `index` in the apply callback, a program part depending on `word-id`'s output configuration. The dependence on the S3 bucket is made explicit by referencing its object in the input configuration (Line 7.1.13). Finally, Line 7.1.20 exports the website's URL.

## 7.2 The Dilemma of Testing IaC Programs

Even though neglected by PL-IaC developers (cf. Section 6.4.2), systematic testing is crucial for the high-velocity development of IaC—no less than for traditional software [111, 112]. Without testing, e.g., it is easy to miss the bug in Listing 7.1: The random number ranges from zero to three (Line 7.1.9), but the `words` array index only from zero to two. If three is drawn, Line 7.1.16 calls `toUpperCase()` on `undefined`, causing an error. We now consider today's PL-IaC testing techniques from Section 2.2.4 for Listing 7.1.

For integration testing IaC programs, including end-to-end and property testing, a single run of Listing 7.1 takes at least seconds. Programs with more complex resources may require hours and cause high infrastructure costs. Testing only a few configurations can miss corner-case bugs, like in Listing 7.1. The latency and cost limit the velocity of the IaC program development.

Dry running is fast and does not require coding. Yet, it cannot find many errors, including the one in Listing 7.1, because it does not execute code depending on output configuration that is only available post-deployment, e.g., the `apply` callback in Lines 7.1.11 to 7.1.18 is not executed when dry running the program before deployment.

Unit testing PL-IaC is labor-intensive and error-prone compared to developing the program. First, one has to mock *all* resource definitions—three in Listing 7.1. This step is not problematic per se, e.g., by adopting the runtime mocking Pulumi provides. Yet, to create effective mocks, developers must implement validation logic for the input configuration and generate output configurations as test inputs for the rest of the program. Such code simulates the logic of cloud configuration, which is complex and requires a correct model. Lastly, developers must ensure the tests cover all relevant cases and may need to update mocks with every change. Thus, unit testing IaC programs is very labor-intensive and error-prone.

In summary, current testing techniques for PL-IaC, e.g., for Pulumi, pose a dilemma to developers: They either invest excessive programming effort for efficient unit testing or resort to integration testing, which is notoriously slow and causes high infrastructure costs. Both hamper the development velocity of deployments.

## 7.3 Automated Configuration Testing

To solve this issue, we now introduce Automated Configuration Testing (ACT), a novel testing methodology for IaC programs. To effectively address the testing dilemma, ACT is a *unit testing* technique because the core issue of integration testing, being slow and resource-intensive, is caused by the cloud providers, e.g., AWS and Azure, and cannot be significantly improved at the side of IaC developers. Thus, we aim to understand and minimize the developer's unit testing effort.

### 7.3.1 Why Unit Testing IaC Programs is Effortful: Mocks

Efficient unit testing requires eliminating of integration with external, slow, and resource-intensive components. For IaC programs, this means mocking the interaction with the cloud, which is encapsulated in resource definitions. To this end, all resource object instantiations, a substantial part of the IaC program's code, must be mocked—most of the code in the RWW example (Listing 7.1).

Mocking all resource definitions with a naïve mock is trivial, requiring, e.g., in Pulumi TypeScript, only a couple of lines of code—independent of the IaC program's size. Yet, for effective unit testing, the mocks have to implement the cloud logic in two crucial aspects. (1) The mocks have to return an output configuration for each resource input configuration they receive. This is because, in a real deployment, the cloud provides the resource's output configuration to the IaC program after the resource deployment. As the output configurations are accessible in the remaining IaC program, they indeed constitute *test input*. Thus, the returned output configurations have to be realistic to test the remaining IaC program precisely. Further, to cover all paths, it may be necessary to return different output configurations across test executions. (2) To test the declarative target state the IaC program defines, i.e., the cloud configuration to set up and not only the imperative IaC program execution, the mocks have to validate the received resource input configurations (i.e., they have to implement *test oracles*). This is because the cloud provides feedback to the IaC program on the resource input configurations by reporting an error when an invalid configuration is deployed.

Such oracles should be *intentless*, i.e., they reject configurations that are generally invalid, independent of the IaC program's context. Ideally, they are further *intentful*, i.e., they also reject configurations that violate the IaC program's application-specific goals.

Finally, the significant challenge is that mocks have to implement both suitable test generators *and* oracles. Suitable test generators ensure coverage and minimize false positives because they do not generate unrealistic test inputs that trigger issues that would never occur in practice. Suitable test oracles verify the cloud configuration the IaC program defines. Both are non-trivial and require a significant amount of code—likely a multiple of the IaC program's code. Further, such mocks mirror the logic of the IaC program under test and the cloud it uses, leading to code tightly coupled with the IaC program, ultimately slowing down any future changes.

### 7.3.2 Automating Unit Testing with ACT

To solve these issues, we propose ACT (Figure 7.2). ACT automatically mocks all resource definitions by intercepting the constructors of resource classes, e.g., the constructor of `aws.s3.Bucket` in Lines 7.1.6 to 7.1.8 of Listing 7.1. The ACT resource mocks receive the input configuration of each resource and return suitable output configurations. The resource mocks implement both a test generator and a set of test oracles.

A test generator provides a separate value producer for each test case. During the execution of a test case, its value producer receives the resources' input configurations and returns for each an output configuration. As the output configurations are test input, the test case is ultimately defined as the sequence of output configurations its value producer returns. An oracle is a predicate function that decides whether the resource's input configurations are valid. We distinguish between two kinds of oracles. *Resource oracles* receive individual resource input configurations during the IaC program execution. *Deployment oracles* receive the input configuration of all resources after the IaC program completed, enabling holistic validations.

In ACT, both the generator and the oracles are plugins, allowing for exchange, adoption, and experimentation with test generators and oracles. Ideally, these plugins

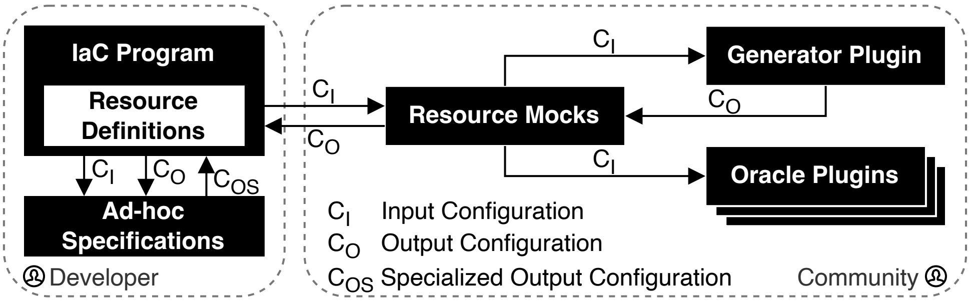


**Figure 7.2:** Overview of Automated Configuration Testing (ACT) [229] © 2024 IEEE.

implement generalized, reusable generation and validation strategies decoupled from a specific IaC program. ACT solves the issue of unit testing IaC programs by moving the development effort of testing code from the developers of an individual IaC program to the community. Once the community instantiates ACT for a specific platform (e.g., .Net or Python; our reference implementation covers Pulumi Typescript) and provides suitable plugins, developers can test the basic correctness of the imperative IaC program and its target state without implementing any code.

ACT's approach fosters the reuse of plugins across different applications. To ensure that testing is also based on application-specific knowledge (e.g., *intentful* oracles, Section 7.3.1), a mechanism to augment the community-provided generators and oracles with application-specific generation and validation specifications is needed. For this, ACT implementations can leverage various approaches, e.g., specification DSLs separated from or embedded into the IaC program code. ProTI features *ad-hoc specifications*, an embedded DSL integrated into the IaC program code (Section 7.4.3).

### 7.3.3 Running Test Sequences with ACT

With automated test execution, generation, and validation, ACT can execute the IaC program in many different configurations. For a sequence of tests, the generator plugin provides a different value producer for each test case. The test case selection it performs is crucial, i.e., which value producer instances it chooses, as it determines which and how parts of the IaC program are tested. ACT terminates once an oracle finds a bug, the program under test crashes, or, if no bug is found, after a defined amount of runs or a timeout. Thus, a generator's prioritization and selection of test cases is crucial to ensure relevant bugs are triggered (early).

Conceptually, ACT combines property-based testing (PBT) [49, 78] and fuzzing [266] techniques for IaC programs. Both systematically test a program *s* in many configurations

$c \in C$, which are put into relation by a property $p$, leading to $\forall c \in C.\, p(c, s(c))$ if $s$ is correct. However, the pessimistic assumption is that $s$ contains a bug, yielding the goal to find and test a configuration $e$ leading to $\neg p(e, s(e))$ as early as possible in a sequence of tests. As the generator plugin is exchangeable, ACT is amenable to new state-of-the-art fuzzing and PBT test case selection strategies, e.g., based on testing feedback [177], search-based techniques [149], code coverage [143], and combinatorial coverage [87].

### 7.3.4 Discussion

We now discuss bugs in IaC programs, ACT's design, its relation to cloud models, and the resulting limitations.

**IaC Program Bugs** We propose a bug taxonomy for IaC programs. In contrast to previous, more fine-grained bug taxonomies, e.g., for IaC defects by Rahman et al. [192], we focus purely on the required oracle to find a bug. Recent fuzzing literature, e.g., Su et al. [248] and Li et al. [143], commonly distinguishes *crash bugs* that cause the program to crash and non-crashing *logic bugs*, which require a more precise oracle than crash detection to identify erroneous computations. We add two categories for bugs where the program logic may be correct, but the resulting resource configuration is faulty. *Configuration bugs* are the wrong configuration of an isolated resource, e.g., setting an IPv4 address to the invalid value `400.0.0.1`. With *configuration interaction bugs*, the configuration of the individual resources is valid but invalid in combination. For example, there is a subnet `192.168.0.1/24` and a server in it has the IP address `192.168.1.2`, which is invalid in this subnet. In contrast to crash and logic bugs, configuration bugs require oracles that can identify invalid cloud configurations and, for configuration interaction bugs, even across multiple resources.

Crash bugs and logic bugs are related to "traditional" code. In contrast, configuration (interaction) bugs are related to the embedded DSL code in IaC programs that defines the target state of the deployment through instantiating objects of the resource types' classes. However, IaC programs mix traditional code (Lines 7.1.1 to 7.1.5, Line 7.1.9, and Line 7.1.20 of Listing 7.1) with the embedded DSL code (Lines 7.1.6 to 7.1.8 and Lines 7.1.10 to 7.1.18). This mixing prevents testing the kinds of code in isolation and causes existing testing methods to be only applicable with a huge mocking effort (Section 7.3.1).

**ACT's Approach** ACT focuses on finding configuration (interaction) bugs. To this end, static analysis is a suitable alternative; for example, it can easily find the bug in Listing 7.1. Yet, we base ACT on automated testing because it does not incur the limitations of static analysis when covering complex dynamic behavior of the IaC program code and supporting all features of the host language. In such systematic testing, the generator has to exercise the IaC program in different configurations to find crash and logic bugs that yield wrong configurations effectively. We argue that covering such configuration-related crash and logic bugs is sufficient because IaC programs focus on the configuration, and all relevant logic drives this purpose. If an IaC program implements complicated configuration-unrelated logic, it should be separated from the embedded DSL code and specifically checked with existing, well-established testing techniques.

**Cloud Configuration Models** Generators and oracles implicitly define models of cloud resources. Such models could be derived from specifications, be hand-crafted, or, more realistically, be derived from existing approximate models, including *types*. For instance, Pulumi providers, i.e., vendor-specific plugins used by Pulumi to interact with the cloud, are distributed as packages that contain a schema JSON file defining the types of the resources' target and output configuration. Such type definitions are a configuration model that is available for all resources—even for dynamically-typed languages—and they can be leveraged for *type-based* generators and oracles [49]. ACT's open architecture ensures that developers can adopt and combine available models and plug in domain-specific optimizations. ACT is not limited to functional properties. For instance, models of cloud performance and security, predicting bad performance and insecure setups based on resource configurations, can be embedded in ACT oracle plugins to cover such non-functional aspects.

Ideally, models for ACT generators and oracles are (1) *complete*, i.e., they can produce all valid configurations, and (2) *correct*, i.e., they include only valid configurations. Incomplete models in a generator systematically prevent generating test cases that may be needed to find bugs, and incorrect models can yield test cases that never occur in practice. Incomplete models in oracles can trigger false positives (i.e., alerts in the absence of a bug), and incorrect models false negatives (i.e., missing bugs). In practice, cloud models are not perfect. For instance, Pulumi package schema types are complete, but not fully correct. In RWW (Listing 7.1), a correct generator should generate integers in the range (Line 7.1.9) for `RandomInteger`'s `result` field (Line 7.1.11). Yet, a type-based generator provides any number, including outside the range and fractions, because the type of `RandomInteger.result` is number. Similarly, a correct oracle only accepts valid HTML for the `content` field (Line 7.1.16), but a type-based one accepts any string.

**Table 7.1:** Size of all ProTI packages. Only non-blank and non-comment SLOC. [229] © 2024 IEEE.

| Package | Description | Source SLOC | Test SLOC |
|---|---|---|---|
| @proti-iac/core | Core abstractions | 758 | 863 |
| @proti-iac/runner | Jest runner | 26 | 51 |
| @proti-iac/test-runner | Jest test runner | 429 | 90 |
| @proti-iac/reporter | Jest reporter for check results | 149 | 19 |
| @proti-iac/spec | Ad-hoc specifications | 12 | 74 |
| @proti-iac/pulumi-packages-schema | Pulumi packages schema infrastructure, oracle, and generator | 1 334 | 1 960 |
| Total | | 2 708 | 3 057 |

In practice, useful test generators and oracles may still generate irrelevant tests or miss bugs. Even if application-specific knowledge can further limit the configuration space, correcting the model in generator and oracle plugins may overfit the plugins to the specific program, reducing reusability or slowing down development. ACT addresses these issues by enabling fine-tuning of test generation and oracles for a specific application, e.g., ProTI provides an ad-hoc specifications syntax (Section 7.4.3).

## 7.4 ProTI: ACT for Pulumi TypeScript

We present ProTI, an instantiation of ACT for Pulumi TypeScript. ProTI is open source and publicly maintained on GitHub[3] with long-term archived releases [231]. We built ProTI upon the popular JavaScript testing tool Jest [157], fast-check [66] for the test execution strategy and arbitraries, and Pulumi's runtime mocking. During the development of ProTI, we added support for asynchronous mocks to Pulumi's runtime mocking, which we contributed back to Pulumi's open-source codebase and is meanwhile by default available in Pulumi's NodeJS SDK.[4] ProTI comprises six TypeScript packages (Table 7.1). The first four packages implement the core abstractions and Jest plugins for a Jest runner, test runner, and reporter. `@proti/pulumi-packages-schema` is a Pulumi-packages-schema-based oracle and a generator plugin. `@proti/spec` implements the ad-hoc specification syntax. ProTI is used through Jest's CLI, which's configuration it facilitates with a preset. ProTI preserves Jest's pre-test features and optimizations, e.g., creating, caching, and watching an in-memory file system for the code.

[3] https://github.com/proti-iac/proti

[4] GitHub issue: https://github.com/pulumi/pulumi/issues/13049.

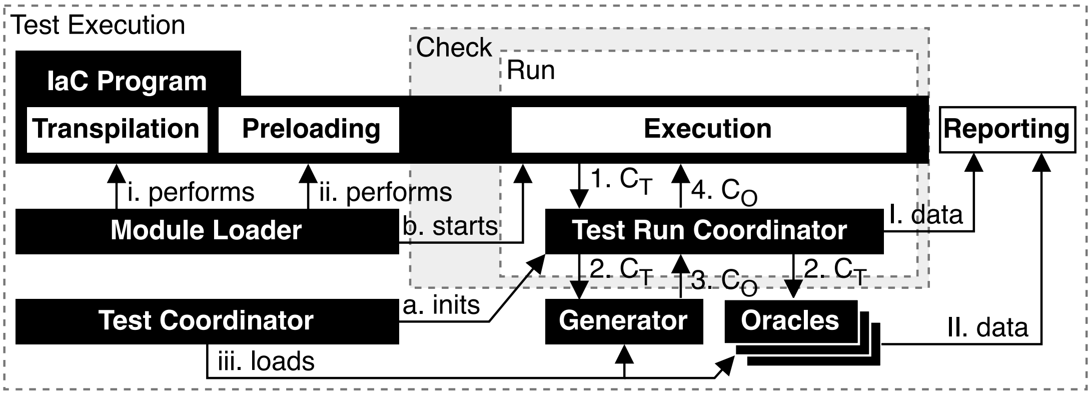
### 7.4.1 Test Execution with ProTI

Jest *runners* distribute tests over multiple workers. They invoke a *test runner* for each test suite. ProTI's runner extends Jest's default by (1) verifying the test configuration and (2) forwarding file system and module resolution information to ProTI's test runners, which they had to re-generate otherwise.

ProTI's test runner is invoked once on the `Pulumi.yaml` of each IaC program and implements ACT (Section 7.3). Figure 7.3 details the test execution. First, the IaC program and its dependencies are transpiled to JavaScript (i) and a configured set of dependencies is preloaded (ii). Preloaded modules are shared among all IaC program runs, breaking isolation but reducing overhead. For technical reasons, Pulumi's SDK must be preloaded. Further, the test coordinator loads the generator and oracle plugins (iii). ProTI *checks* the IaC program in several *runs*, each configured with its own test run coordinator (a), managing isolated run states for the generator and oracles. Each run executes the IaC program once (b). ProTI mocks all resource definitions by intercepting the constructors of all resource classes with Pulumi's runtime mocking feature. This way, each resource input configuration $\mathrm{C_I}$ is run through validations and transformations that the provider's SDK may implement in the resources' constructors. We call the checked and potentially transformed $\mathrm{C_I}$ target configuration $\mathrm{C_T}$. For instance, a resource's $\mathrm{C_I}$ may contain additional fields, which is valid in TypeScript's structural type system, but the resource constructor does not add them to $\mathrm{C_T}$. ProTI uses $\mathrm{C_T}$ instead of $\mathrm{C_I}$ in the remaining ACT workflow. The mock provides all resources' target configuration $\mathrm{C_T}$ to the generator and oracle plugins (1 – 2), and receives the output configurations $\mathrm{C_O}$ to use in the remaining program execution (3 – 4). Finally, ProTI reports the test results (I – II).

### 7.4.2 Test Generator and Oracle Plugins

In an execution, ProTI loads exactly one generator plugin and a variable number of oracle plugins, which are invoked in parallel. We do not provide an explicit mechanism to compose different plugins; however, when developers write a plugin's code, they can also combine other plugins programmatically. ProTI plugins are implemented as NodeJS modules, exporting the respective plugin as default and, optionally, an `init` function of ProTI's `TestModuleInitFn` type that can implement initialization code called by ProTI when loading the plugin and also implements a plugin configuration interface. `@proti-iac/core` implements all plugin-related types.



i. – iii. initialization  a. – b. run initialization  1. – 4. resource mocking  I. – II. reporting

**Figure 7.3:** ProTI test execution overview [229] © 2024 IEEE.

Generator plugins are implemented as fast-check value generators of ProTI's `Generator` type, i.e., type `Arbitrary<Generator>`. The arbitrary is called once for each test run to provide a `Generator` and may implement shrinking, a technique from property-based testing where, once an error is found, simplified versions are tested and presented to the developer as an easier-to-understand alternative if they still trigger the bug [49]. The test run's generator is invoked for each resource with its target configuration and returns its output configuration for the run. Further, the generator is invoked with the arbitrary of each ad-hoc generator specification, guiding its execution to enable deterministic test generation strategies, including shrinking.

Oracle plugins are implemented as a class inheriting from ProTI's `Oracle<S>` type and can leverage state of type `S` that is initialized for every test run through a function they implement and passed to all invocations of the oracle in the run. For these invocations, oracles implement at least one out of four resource input configuration validation interfaces, which are separately called for each resource or once with all resources, both available synchronously and asynchronously.

For now, ProTI provides default generator and oracle plugins based on Pulumi packages schema types in `@proti-iac/pulumi-packages-schema`. The package implements the infrastructure to automatically retrieve the schemas of all resources in the IaC program under test. The oracle translates the schemas' resource types to validation functions to dynamically check each resource input configuration. The generator composes fast-check arbitraries to generate output configurations, inheriting fast-check's random value generation strategy, which is biased towards generating extremes, e.g., instead of using an even distribution, it prioritizes generating small and big values. How-

Listing 7.2: RWW (Listing 7.1) with ProTI ad-hoc specifications (orange) [229] © 2024 IEEE.

```
import * as ps from '@proti/spec';
... // Same as Lines 7.1.1 to 7.1.8
const range = { min: 0, max: words.length - 1 };
const rng = new random.RandomInteger('word-id', range);
ps.generate(rng.result).with(ps.integer(range))
    .apply((wordId) => {
        new aws.s3.BucketObject('index', {
            bucket: bucket, key: 'index.html',
            contentType: 'text/html; charset=utf-8',
            content: '<!DOCTYPE html>' +
                ps.expect(words[wordId].toUpperCase())
                    .to((s) => s.length > 0)
        });
    });

export const url = bucket.websiteEndpoint;
```

ever, ProTI can be easily extended with oracles and generator arbitraries based on other model sources, e.g., codified policies and cloud specifications.

During this package's development, we identified, reported, and supported fixing issues in Pulumi's CLI schema feature[5] and 169 types in the AWS classic[6] and random[7] providers, which the maintainers acknowledged. The first two issues, in the CLI and in the random provider, are fixed meanwhile. The root cause investigation for the third one, in the AWS classic provider, is ongoing.

### 7.4.3 Ad-hoc Specifications in ProTI

To fine-tune generators and oracles, ProTI provides ad-hoc specification syntax. Developers can use `generate(e).with(a)` to define an ad-hoc specification to replace values returned by `e` with values from a fast-check arbitrary `a`. For ad-hoc oracles, `expect(e).to(p)` applies an oracle predicate function to an expression `e`. In a regular execution, the ad-hoc syntax only returns the evaluation of the wrapped expression `e`, with no change in the semantics of the IaC program. When running with ProTI, however, `generate(e).with(a)` calls the generator plugin with `a` and returns a value from `a`. The oracle syntax still returns the evaluation of `e`, but it introduces a check, reporting an error if `p(e)` is `false` or fails.

[5] GitHub issue: https://github.com/pulumi/pulumi/issues/13279.

[6] GitHub issue: https://github.com/pulumi/pulumi-aws/issues/2565.

[7] GitHub issue: https://github.com/pulumi/pulumi-random/issues/279.

Listing 7.2 fixes the indexing bug in Listing 7.1 and is fine-tuned with ad-hoc specifications. In ProTI executions, the ad-hoc generator specification in Line 7.2.12 addresses the imprecision of the type-based oracle in Line 7.1.11, which generates any number, not only realistic output configuration values. Instead, Line 7.2.12 specifies thath integer values shall only be generated in the correct `range` interval. Further, Lines 7.2.18 to 7.2.19 specify an oracle that checks that the webpage's content is not empty, encoding the developers' application-specific intent to show a non-empty webpage.

While ProTI implements this embedded specification DSL for application-specific generator and oracle directives, ACT implementations could use external DSLs or encourage separating the specification code into other files, as common with most testing frameworks today. Such separation is also possible with ProTI's ad-hoc specifications but would require restructuring the IaC program code to improve testability. For instance, the code augmented with specifications in Listing 7.2 could be wrapped in functions that separate testing files mock during testing. We support inlining in ProTI for simplicity, assuming that few ad-hoc specifications are required with good plugins. Yet, if a lot of ad-hoc specifications are required, separation is preferable to avoid the added complexity of mixing concerns, obfuscating the IaC program code, and potentially introducing new error sources.

## 7.5 Evaluation

We evaluate ACT's effectiveness, applicability, performance, and extensibility by answering the following research questions about its ProTI implementation.

**RQ 7.1** Can ProTI find bugs reliably? We determined whether ACT and ProTI are a feasible testing methodology and tool for IaC programs and compared them with the existing PL-IaC testing techniques (cf. Section 2.2.4).

**RQ 7.2** Is ProTI applicable to real-world open-source code? We explored whether our ACT implementation ProTI is mature enough to be applied to real-world IaC programs.

**RQ 7.3** How long does ProTI run, and how does the run time scale? We measured ProTI's execution duration and scalability to ensure it is fast enough.

**RQ 7.4** Can existing test generation and oracle tools be integrated into ProTI? We investigated whether ACT's architecture allows to leverage third-party oracles and generators.

The following four subsections present our experiments, and Section 7.5.5 discusses their results and threats to validity. We ran all experiments on serverless AWS Fargate [13] containers with 1 vCPU and 4 GB of memory on AWS Elastic Container Service (ECS) [5] in the region eu-west-1.

### 7.5.1 Finding Errors in IaC Programs

We compared ProTI with the available testing techniques for Pulumi TypeScript programs (cf. Section 2.2.4) on nine variants of the RWW example. The variants are the following. VC is correct, VS is Listing 7.2, i.e., VC with ProTI ad-hoc specifications, and VSDB adds the deployment of a serverless database to VS. Most remaining variants have a crash bug according to our bug taxonomy (Section 7.3.4): VNT has syntax errors, VE always throws an error, which VAE throws asynchronously, VO is Listing 7.1, i.e., it has a one-off bug in asynchronous code that leads to a crash, which we combine with the ad-hoc specifications of Listing 7.2 in VSO. VSB is VS with a configuration bug, setting a string instead of an object for the bucket's website property (cf. Line 7.1.7).

ProTI was configured with the type-based oracle and generator (Section 7.4) and up to 100 runs. Unit testing used Jest [157] and ran the program once with a naïve mock that returned empty configurations. Dry running executed `pulumi preview`. (Dry) property testing executed Pulumi CrossGuard [184] via (`pulumi preview`) `pulumi up` with the AWSGuard policy pack [183]. All Pulumi commands were non-interactive with skipped previews. End-to-end testing used Pulumi's Go integration testing framework [182], checking the content of the deployed website. We executed each experiment 10 times after warmup. Table 7.2 reports whether an error was (always) found and the minimum and average run time.

As expected, dry running did not find asynchronous errors (VAE, VO, and VSO) as it does not run code depending on unknown output configurations. Property testing and end-to-end testing found the one-off bugs (VO and VSO) only occasionally. ProTI was the only technique that spotted all errors reliably. However, the imprecision of the type-based generator, i.e., generating any number for `rng.result` and not only integer values in the defined range, increased the likelihood of finding the error in VO, but also caused that ProTI identified VC as faulty; a false positive. This imprecision is resolved in VS, VSO, VSB, and VSDB with ProTI ad-hoc specifications (cf. Section 7.4.2). ProTI always identified bugs in the first test run, except in VSO, where it required 2 to 6 tests, causing a slightly longer run time compared to VO.

The experiment answers RQ 7.1: ProTI can find bugs reliably and is able to uncover errors in edge cases without explicitly testing for them.

**Table 7.2:** PL-IaC testing techniques on variants of the RWW example (Listing 7.1). ∗ faulty variant. Error found ∗ (always ⊛), minimum (average) run time over 10 repetitions. [229] © 2024 IEEE.

| | | ProTI | Dry Run | Property Test |
|---|---|---|---|---|
| | | Unit Test | Dry Property Test | End-to-end Test |
| ∗ | VNT: | ⊛ 16.7 s (16.8 s) | ⊛ 10.0 s (10.3 s) | ⊛ 12.4 s (12.4 s) |
| | Non-transpilable | ⊛ 1.9 s (2.0 s) | ⊛ 11.6 s (11.7 s) | ⊛ 47.9 s (65.7 s) |
| ∗ | VE: Error | ⊛ 7.0 s (7.2 s) | ⊛ 2.3 s (2.4 s) | ⊛ 4.4 s (4.5 s) |
| | | ⊛ 2.2 s (2.2 s) | ⊛ 3.7 s (3.8 s) | ⊛ 52.5 s (59.6 s) |
| ∗ | VAE: | ⊛ 7.4 s (7.6 s) | 3.4 s (3.5 s) | ⊛ 9.4 s (9.6 s) |
| | Async Error | ⊛ 2.4 s (2.5 s) | 4.8 s (4.9 s) | ⊛ 50.8 s (60.7 s) |
| | VC: Correct | ⊛ 7.5 s (7.6 s) | 3.4 s (3.4 s) | 9.5 s (9.7 s) |
| | | 2.7 s (2.7 s) | 4.8 s (4.9 s) | 53.5 s (59.0 s) |
| | VS: Listing 7.2 | 21.0 s (21.1 s) | 3.5 s (3.5 s) | 9.5 s (9.7 s) |
| | (ad-hoc specs.) | 2.8 s (2.9 s) | 5.0 s (5.0 s) | 52.6 s (62.3 s) |
| ∗ | VO: Listing 7.1 | ⊛ 7.4 s (7.6 s) | 3.4 s (3.4 s) | ∗ 9.4 s (9.6 s) |
| | (one-off bug) | 2.7 s (2.7 s) | 4.8 s (4.9 s) | ∗ 51.9 s (58.4 s) |
| ∗ | VSO: Listing 7.2 | ⊛ 8.1 s (8.3 s) | 3.5 s (3.6 s) | ∗ 9.5 s (9.7 s) |
| | with one-off bug | 2.8 s (2.9 s) | 4.9 s (5.0 s) | ∗ 59.5 s (66.6 s) |
| ∗ | VSB: Listing 7.2 | ⊛ 7.6 s (7.8 s) | ⊛ 3.5 s (3.5 s) | ⊛ 5.6 s (5.7 s) |
| | with config. bug | 2.8 s (2.9 s) | ⊛ 4.8 s (4.9 s) | ⊛ 48.4 s (57.4 s) |
| | VSDB: Listing 7.2 | 39.2 s (39.6 s) | 8.1 s (8.4 s) | 163.4 s (189.9 s) |
| | with AWS RDS | 3.1 s (3.1 s) | 8.0 s (8.1 s) | 212.5 s (265.7 s) |

### 7.5.2 Applicability to Real-world Programs

We executed ProTI on all 6 081 Pulumi TypeScript programs in the PIPr dataset [232] of all public IaC programs on GitHub in August 2022. PIPr contains examples, toy projects, and production projects in unknown shares and is only filtered by the relevance criteria inherent to the GitHub Code Search API [85] we used for the evaluation and that we discuss in detail in the dataset's paper [232]. PNPM was used to install dependencies and TypeScript version 5.1.6 for the execution.

The first two columns of Table 7.3 show the results. We categorized the executions by the phase in the ProTI run where a problem was detected: invalid *project* files that prevent execution, failures during *transpilation*, failures during module *preloading*, failures during *checking*, successfully *passed*, and *crashed* executions. Within each category, we grouped errors by common causes and report their frequency. Both the categorization and error labeling are based on string matching on the execution logs, and the error grouping by open coding. This process was incrementally performed and implemented by the first author and reviewed by the second author. The authors know Pulumi and ProTI well through their research.

On a technical level, ProTI was able to test 40 % of the IaC programs out of the box. This share is extremely remarkable and exceeds our initial expectations because (1) we did not filter for buggy or non-functional programs, (2) ran all programs with current NodeJS and TypeScript versions, and (3) did neither look into nor provide any program-specific environments. We suspect that ProTI can be used for most of the remaining IaC programs, too, after little effort is invested to understand their expected execution environment or bug.

The most common reasons why ProTI could not test a program are module resolution and type checking, failing 1 745 (29 %) and 984 (16 %) executions. The causes include incompatibility with PNPM, the TypeScript version, unmet environment assumptions, and incomplete, broken setups. Among the programs ProTI was able to test, it found issues in 68 %. The tests found 659 (11 %) executions where the setup was incomplete, e.g., missing configuration or programs. Mocking failed in 468 (8 %) executions, which can be caused by incompatible, outdated Pulumi versions. Our type-based oracle and generator failed to find type definitions in 416 (7 %) executions because they are dynamic resources, stack references, or missing in the provider's schema. Our oracle identified invalid resource configurations in 58 (1 %) executions. ProTI ran only an unknown number of tests in crashed executions, 100 tests in the passing ones, and only a single test in 98 % of the executions under checking. In the other 26 checking executions, ProTI ran between 2 and 38 tests until an error was found. Due to a lack of ground truth, we cannot determine the precision and recall of the experiment.

The experiment answers RQ 7.2: ProTI can be applied to existing IaC programs.

### 7.5.3 Execution Duration and Scaling Behavior

We performed time measurements on Pulumi programs that define 0, 1, 10, 50, and 100 AWS S3 bucket resources. The experiment considered two program variants, one defining the resources *independently* for parallel deployment, and one in a dependency *chain* for sequential deployment. We ran ProTI three times on each program and repeated the experiment five times. As the programs are correct, ProTI runs them 100 times in each execution without identifying a bug. Table 7.4 and Figure 7.4 report the average execution time in total and separated by phase. Table 7.4 shows the absolute values separately for

**Table 7.3:** Execution time and result classification of ProTI executions on 6 081 Pulumi TypeScript programs [229] © 2024 IEEE.

| Category<br># programs. | Error Reason [# programs. (% in category)] | Execution Time<br>average<br>(std) |
|---|---|---|
| Project<br>2 (0 %) | invalid Pulumi.yaml 2 (100 %) | 1.6 s<br>(0.1 s) |
| Transpilation<br>2 649 (44 %) | module resolution 1 335 (50 %), type checking 984 (37 %), program resolution 324 (12 %), legacy NodeJS 5 (0 %), JSX 1 (0 %) | 8.9 s<br>(5.6 s) |
| Preloading<br>482 (8 %) | module resolution 410 (85 %), legacy NodeJS/Pulumi 20 (4 %), unknown 18 (4 %), syntax error 18 (4 %), config 16 (3 %) | 7.8 s<br>(5.9 s) |
| Checking<br>1 633 (27 %) | setup 659 (40 %), mocking 468 (29 %), missing type definition 416 (25 %), application 86 (5 %), other 64 (4 %), oracle 58 (4 %) | 17.2 s<br>(17.2 s) |
| Passed<br>772 (13 %) | | 23.4 s<br>(11.4 s) |
| Crashed<br>543 (9 %) | out of memory 473 (87 %), unknown 70 (13 %) | 25.9 s<br>(38.9 s) |
| Total<br>6 081 (100 %) | | 14.4 s<br>(17.0 s) |

the first and consecutive runs. Figure 7.4 separates the first, second, and third runs and also shows results as relative values.

Execution times are higher for first runs because the transpilation overhead is significant and, on average, 76 % lower in subsequent runs (Figure 7.4). Test runs, transpilation, and module preloading are the only actions of the test runner taking significant time. The remaining execution time was consumed outside the test runner, including Jest's setup and reporting. A single test run in the experiments took 10 ms to 5.9 s, and the duration scales linearly with the resource number.

We found similar execution times in the IaC programs from GitHub (Table 7.3). Conservatively approximating a single test duration by dividing the total run time of all *passed* ProTI executions by 100 (the number of runs), we measured test run durations from 34 ms to 1.0 s; 234 ms on average. It is an approximation because the total run time also includes overhead like setup and reporting, and it is conservative because we assume these contributors are instant, i.e., the test run duration is likely a bit lower. The RWW experiments (Table 7.2) confirm these durations, too. Lastly, our experiments show that ProTI is quicker when it finds a bug because of early termination.

**Table 7.4:** Total average execution time of ProTI over 5 repetitions of the duration experiments by first/consecutive execution and phase for IaC programs with 0, 10, 50, and 100 resources with both independent and chained dependencies [229] © 2024 IEEE.

| | Resources: | 0 | 1 | 10 | | 50 | | 100 | |
|---|---|---|---|---|---|---|---|---|---|
| | Phase | | | indep. | chain | indep. | chain | indep. | chain |
| Run 1 | Remaining | 1.7 s | 1.6 s | 2.2 s | 2.2 s | 6.3 s | 4.6 s | 14.8 s | 15.9 s |
| | 100 Runs | 1.0 s | 9.3 s | 57.4 s | 69.5 s | 274.5 s | 262.8 s | 563.3 s | 535.8 s |
| | Preloading | 0.7 s | 0.7 s | 0.7 s | 0.7 s | 0.7 s | 0.7 s | 0.7 s | 0.7 s |
| | Transpilation | 15.1 s | 15.2 s | 15.3 s | 15.3 s | 15.2 s | 15.3 s | 15.3 s | 15.3 s |
| | Total | 18.5 s | 26.8 s | 75.6 s | 87.7 s | 296.6 s | 283.4 s | 594.2 s | 567.8 s |
| Run 2 & 3 | Remaining | 1.6 s | 1.6 s | 2.2 s | 2.3 s | 6.3 s | 6.3 s | 11.1 s | 10.0 s |
| | 100 Runs | 1.4 s | 7.4 s | 51.7 s | 50.3 s | 260.8 s | 243.1 s | 520.2 s | 493.6 s |
| | Preloading | 0.8 s | 0.8 s | 0.8 s | 0.8 s | 0.7 s | 0.8 s | 0.8 s | 0.8 s |
| | Transpilation | 3.7 s | 3.7 s | 3.7 s | 3.7 s | 3.7 s | 3.7 s | 3.7 s | 3.7 s |
| | Total | 7.5 s | 13.5 s | 58.3 s | 57.1 s | 271.6 s | 253.9 s | 535.7 s | 508.1 s |

The experiments passing RWW experiments with 6 resources (VS) and 25 resources (VSDB) in Table 7.2 confirm that test time grows with the number of resources (on average, 21 s and 40 s including overhead). They further show that the performance of integration testing heavily depends on the deployment time of the resources—which ProTI is independent of. Deploying AWS RDS databases takes longer than AWS S3 resources, yielding testing VSDB takes 20× and 4× longer than VS with property testing and end-to-end testing, respectively, while ProTI was only 2× slower.

The results answer RQ 7.3: A single test run of ProTI typically takes hundreds of milliseconds and test duration scales with the number of resources—not with their deployment time—permitting to quickly check hundreds of configurations.

### 7.5.4 Integrating Existing Tools into ProTI

ACT's effectiveness is crucially dependent on the quality of its plugins. Many techniques have been developed for test generation and oracles (cf. Section 7.3.3). To leverage advanced techniques from related work, ProTI must be open to extension with them. To demonstrate ProTI's extendability, we implemented ProTI plugins using the Radamsa fuzzer [107] and the Daikon invariant detector [73] with a generator and an oracle plugin based on existing tools. This experiment assesses the feasibility of integrating existing approaches; optimizing them and evaluating their effectiveness and efficiency is the subject of future work focusing on test generation and oracle techniques, while this paper focuses on the overall approach.

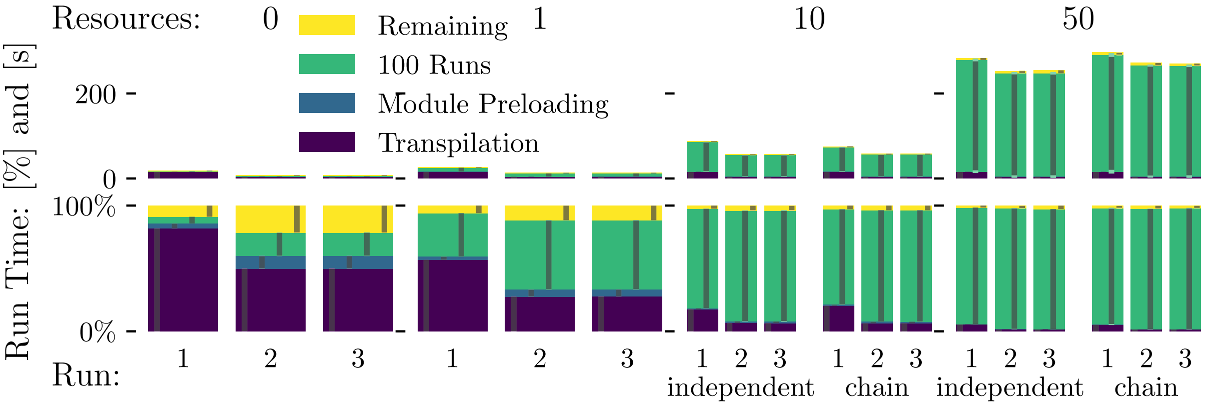


**Figure 7.4:** Average execution time of ProTI over 5 repetitions of the duration experiments (Table 7.4) by phase, resource count, and dependency. Results for three consecutive executions (1, 2, 3). In total (top row) and relative (bottom row). [229] © 2024 IEEE.

Radamsa [107] is a fuzzing tool that derives fresh test inputs from an example. We adopted it for a ProTI generator plugin that, separately for each resource type, uses the type-based generator to generate an output configuration example, which is passed to Radamsa as JSON to generate a list of derived test inputs. We filter non-parsable configurations from Radamsa's results and use the remaining ones as test input in ProTI. Whenever ProTI runs out of Radamsa-generated inputs, we repeat the procedure. The generator implementation required 83 SLOC, of which only 48 differ from a naïve generator returning empty configurations.

Daikon [73] is a dynamic invariant detector that identifies application invariants in a set of program traces. We used it for an invariant regression oracle that detects behavior changes across different versions of an IaC program. In the first ProTI execution, the oracle records all resources' target and output states and invokes Daikon on them to find resource configuration invariants over all runs, e.g., a particular bucket's id equals a field of a policy, independent of the concrete value. In consecutive ProTI executions, we repeat the procedure and additionally compare the obtained invariants with the previously generated ones, issuing a warning if an invariant cannot be found anymore, i.e., it may be violated in the new program version. The oracle plugin comprises only 120 SLOC, mainly for converting resource configurations between ProTI and Daikon and managing state across executions.

Our observations positively answer RQ 7.4: Existing tools can be integrated into ProTI by implementing a plugin, demonstrating ProTI's openness to third-party techniques.

### 7.5.5 Limitations, Threats to Validity, and Implications

Our experiments on ProTI show that ACT can find bugs quickly and reliably in IaC programs, even in edge cases (RQ 7.1), can be applied to IaC programs without adjustments (RQ 7.2), can be fast enough to run hundreds of tests in a short time (RQ 7.3), and can be extended with existing tools through generator and oracle plugins (RQ 7.4). Yet, our experiments do not provide quantitative insight into ACT's effectiveness, i.e., the likelihood that all bugs and no false positives are found and after which time. Such insights require an IaC program dataset *with correctness annotations*, i.e., precise knowledge about bugs in them. Such evaluation is planned in future work to assess advanced generator and oracle plugins. This paper focuses on the feasibility of the ACT approach to test IaC, not on the precision and recall of a specific testing technique.

Relevant threats to validity in this work include that we evaluate ACT through ProTI, a single instantiation for one specific PL-IaC solution and language. Yet, we expect that implementations for other languages and PL-IaC solutions yield similar results because IaC programs for other tools and other languages, i.e., the embedded PL-IaC DSL, are, technically, analogous. The IaC program selection in our experiments is also a threat. For RQ 7.1, the set of variants n RWW suffices to demonstrate the behavioral differences of ACT compared to other techniques; yet, more experiments are needed to show with statistical significance that these differences are relevant in practice such that ACT is beneficial on other IaC programs. For RQ 7.2, we inherit the limitations and validity threats of the PIPr dataset [232]—including generalizability—but, based on our experience, we expect the qualitative insight to apply to other IaC programs. For RQ 7.3, we focused on the number of resources and their dependencies in IaC programs, showing how they influence performance. We rely on our experience that resource number and dependency are the factors that most significantly impact performance, but other factors can be studied with a more comprehensive sensitivity analysis. The categorization, as well as the error labeling and grouping in RQ 7.2, may be subjective, an issue we limited through the review of a second author. Another potential issue is that ProTI is a random-based testing tool, which, in case of a bug, may cause the bug to be inconsistently (not) caught by different test cases across executions. Hence, we apply 10 repetitions for RQ 7.1. For RQ 7.2, we saw negligible variance in tests. As the programs in RQ 7.3 are correct, they are not impacted by this threat. RQ 7.4 is also not affected because it only demonstrates that existing tools can be leveraged in ACT. RQ 7.4 does not measure ProTI executions to quantify the effectiveness of specific tools in the context of IaC. This aspect must be evaluated for each plugin and crucially depends on the implemented method.

For practitioners, ACT and ProTI are new techniques whose effectiveness depends, in the long run, on a community effort to maintain the framework and the test generation and oracle plugins. Practitioners can now try out ACT with low effort on existing Pulumi TypeScript IaC programs. This solution can already reduce the development time through earlier bug detection and increase the reliability of IaC programs, supporting faster evolving, functional, secure systems. A user study assessing user acceptance of ACT and ProTI is left to future work. For researchers, ACT and ProTI are novel testbeds that facilitate exploring advanced test generation and oracle techniques for IaC programs and correct and secure cloud configuration.

## 7.6 Conclusion

Testing is rarely used for IaC programs because available techniques either hinder development velocity or require much programming effort. We present Automated Configuration Testing (ACT) for quick IaC program testing at low effort by automatically mocking all resource definitions and using oracle and generator plugins for validation and test input generation. We implemented ACT for Pulumi TypeScript in ProTI with type-based oracles and generators and support for ad-hoc specifications. ProTI is effective on existing IaC programs, and its modular architecture enables the use of existing third-party and novel test generators and oracles.

## Chapter 8

# Conclusion[1]

This dissertation on reliable IaC for decentralized organizations contributes to the safe coordination and automated testing of IaC programs. Both directions are imperative for modern IT organizations, which must adopt their applications quickly to changing requirements while ensuring their correctness and robustness, i.e., reliability. Teams must be agile, i.e., able to act as independently as possible to minimize friction and delays. Still, their deployments may need coordination, but centralizing it limits their independence, especially if there is no perfectly fitting, tailored coordination solution. Hence, enabling teams to automate coordination decentrally, which we introduced with $\mu$S, is necessary to ensure agility and reliability. At the same time, the teams' deployments tend to become more complex and, still, must be correct. Errors in deployment code cause the entire system to malfunction or be insecure. Enabling frequent quality assurance with short feedback cycles, like we introduced with ProTI, is crucial to ensure reliability in fast-paced development.

Section 8.1 summarizes the contribution chapters of this dissertation. Section 8.2 outlines future perspectives for research on related empirical studies, safely automating further requirements through IaC, and quality assurance techniques for IaC programs.

## 8.1 Summary

We performed the Dependencies in DevOps Survey 2021 on 134 IT professionals about dependencies between applications in practice and whether they imply coordination requirements. We found that most applications depend on other applications, and dependencies between applications often constrain the order of their (un)deployment, which for 76 % of the participants requires manual coordination, e.g., via phone, chat, or email. Nevertheless, IT professionals are convinced that automated coordination leads to better

[1] Based on the authors' work in [226, 228, 232]. [228] © 2023 IEEE.

SDO performance. Starting from this mismatch, we noticed that available deployment coordination solutions are centralized, failing to support decentralized organizations.

To fill this gap, we introduced µS, a PL-IaC solution supporting decentralized coordination of deployments, enabling automation in decentralized organizations. In contrast to previous PL-IaC solutions, µS introduces new resource types allowing developers to express in IaC programs (1) with which remote deployment they interact (`RemoteConnection`), (2) which information and resources they produce for a remote, i.e., they offer (`Offer`), and (3) which information and resources they expect from a remote, i.e., they wish (`Wish`). To automate coordination, µS leverages these explicit interfaces, ensuring wishes and resources depending on them are only deployed when their corresponding offers are. For this, µS has a novel PL-IaC runtime that treats IaC programs as long-running programs that react to external signals, contrasting the independent one-off task behavior of previous PL-IaC solutions. µS' implementation is built upon Pulumi TypeScript and is fully compatible with Pulumi TypeScript IaC programs. Our evaluation showed that µS' performance is comparable with state of the art, µS scales as expected, and the coding overhead to express interfaces for coordination is low. Further, µS easily applies to existing decentralized IaC programs, adding coordination to the previously available, limited state-sharing capabilities.

In the next step, we noticed that deployment coordination is needed in systems with distributed transactions across applications, even if they are independently deployable. Especially if these transactions are long-running and frequent, which is typical for workflows, breaking and repeating them due to an update can cause tremendous additional resource consumption and delays. Safe DSU finds when a system's component can be updated while the remaining system continues running without breaking transactions. However, safe DSU was not yet applicable to workflows in decentralized organizations.

To solve this issue, we introduced a new model for safe DSU in workflows and an information dissemination and control algorithm enabling implementation in decentralized organizations, where many independent workflow engines may exist, and each component may have its own orchestrator. Further, we showed that with coordination-supporting IaC solutions like µS, the safe DSU orchestration mechanisms can be implemented directly in IaC programs. This is beneficial from a reliability point of view because it enables holistic analysis of the deployment and its behavior on the IaC program. Otherwise, the orchestrator's and its safe DSU extension's code must also be considered, making testing and reasoning harder. Finally, we proposed the optimized safe DSU approach Essential

Safety, drastically reducing the safe DSU overhead for non-essential changes, i.e., that do not introduce semantic changes. The empirical evaluation through simulating 106 realistic collaborative BPMN workflows and analyzing eight monorepos confirms that safe DSU applies to workflows in decentralized organizations, non-essential changes are common, and Essential Safety reduces safe DSU overhead compared to previous approaches while retaining the strong update safety guarantees.

Beyond enabling safe coordination by automating it in IaC programs, we addressed the quality of the IaC program code developers write. We recognized that applying the vast body of software engineering techniques and tools, e.g., for testing and verification, requires understanding the similarities and differences between traditional software and IaC programs. It is not enough for both to leverage the same general-purpose programming languages. To enable such studies, we built PIPr, the first systematic, open-source dataset of public IaC programs based on all 37 712 public IaC programs we found on GitHub. In initial analyses, we found that AWS CDK is the most used PL-IaC solution, followed by Pulumi. TypeScript and Python are the most popular programming languages, and developers only implemented tests for 25 % of the programs. For Pulumi, the only general PL-IaC solution, it is only 1 %.

We analyzed present testing techniques for IaC programs to find why developers do not test them systematically. We found that they pose a dilemma: either developers have to resort to slow and resource-intensive integration testing or invest huge development efforts, which are much higher than for traditional software and the IaC program itself. To solve this issue, we presented Automated Configuration Testing (ACT), an extensible approach for efficient automated unit testing of IaC programs, enabling quick testing in hundreds of different configurations, often without writing any project-specific testing code. We implemented ACT in ProTI for Pulumi TypeScript with type-based generator and oracle plugins based on Pulumi package schemas and ad-hoc specification syntax. We evaluated ProTI on all 21 245 Pulumi TypeScript programs in PIPr and benchmarks, showing that ProTI can find bugs quickly, even in edge cases, is applicable to existing IaC programs, and can leverage existing test generator and oracle tools through plugins.

## 8.2 Perspectives

We now outline various perspectives for future research related to this dissertation.

**Empirical Studies on Deployment Coordination** While our Dependencies in DevOps Survey 2021 was the first cross-sectional empirical study on deployment coordination, replications after 2021 or with changed demographics are insightful. Beyond simple replication, though, assessing coordination through a longitudinal study can lead to additional insights into how awareness and various coordination measures change organizations' SDO performance and other metrics. Further, a closer look into the applications, their technology, and legacy can provide better insight into when it is more effective to bear with and automate deployment-time coupling or to eliminate coordination, e.g., through changing application boundaries and fault-tolerant implementations.

**Design-time-decoupled Deployment Coordination** The decentralized coordination we introduced with µS supports environments where deployment dependencies are known and enumerable at design time. Specifically, developers have to define a remote connection resource to one specific remote deployment for each offer and wish they define. However, there are environments where resources are shared with or consumed from a dynamically decided number of remote deployments, requiring the expression and automation of one-to-many and not only one-to-one relationships. Further, the specific remote deployment may not be known statically but matched dynamically. An example at scale is the volunteer grid computing platform BOINC, powering projects like Rosetta@home and SETI@home, and allowing anyone to donate their computing resources for research projects [16]. However, private and smaller applications with these requirements certainly exist, too. We believe supporting one-to-many remote connections, wishes, and offers and introducing matching systems to connect deployments dynamically are straightforward extensions of µS, achieving decentralized automation for design-time decoupled deployments with coordination requirements. However, the implications on the organizations and operations, especially regarding security and ensuring deployment, i.e., guaranteed wish satisfaction, are important questions researchers must also address.

**Dynamic IaC** IaC solutions today target static deployments, i.e., deploying, updating, and undeploying are one-off tasks that execute and terminate once completed. Dynamic behavior requires external implementation, e.g., to control updates, a CI/CD pipeline has to execute the updated IaC program, or, e.g., to scale a deployment based on load, the IaC program statically configures an auto-scaler resource that implements the scaling behavior. µS introduces dynamic behavior to IaC because its deployments are long-running processes that adopt the target state in reaction to external signals, i.e., state changes in remote deployments. Researchers can explore IaC solutions abstracting over µS and generalizing these runtime capabilities, yielding dynamic IaC.

Dynamic IaC can be declarative, too, but instead of defining one target state per execution and setting it up, a stream of target states is defined, always setting up the most recent target state. In a generalized PL-IaC solution, this could be achieved by turning each resource output from a once resolving value into a stream, triggering re-evaluation of program parts depending on the output whenever it changes, and replacing its part in the following target state. Conceptually, this idea integrates functional reactive programming abstractions [22, 52, 71], which have proven to yield good program comprehension [212] and will not change how IaC programs look. Dynamic IaC will enable expressing dynamic orchestration mechanisms like CI/CD, load-balancing, and auto-scaling in IaC programs, enabling fine-grained control for developers without crossing boundaries to other systems and DSLs and holistic reasoning about the infrastructure, including its dynamic behavior. The complexity of dynamic mechanisms can be encapsulated in modules and libraries of the used programming language's ecosystem, enabling reuse across projects and eliminating code duplication while retaining holistic reasoning capabilities.

**Reconfiguration in Provisioning-focused IaC** Provisioning-focused IaC solutions, e.g., PL-IaC solutions and Terraform, are designed for immutable infrastructure management. This approach entails setting up resources once and replacing them with new versions for updates rather than modifying them. However, this method can be inefficient and sometimes undesirable. For example, replacing a virtual machine to increase its memory can be more time-consuming than simply expanding its existing memory, and replacing it can result in losing the machine's state, which may not be recoverable.

In practice, IaC solutions do allow mutable changes with limited control and safety guarantees. For example, Pulumi's provider developers use ad-hoc logic to determine whether a resource should be replaced or modified for specific updates. Additionally, IaC program developers can override provider logic to force resource replacement. However, this method has several drawbacks: (1) it lacks transparency, (2) it is insensitive to the context of other resource updates, and (3) it is simplistic and potentially unsafe. Mutable change behavior is undefined and can lead to undesirable, vulnerable states [141].

To address these issues, researchers can explore formal modeling of resource lifecycles in resource providers. Integrating solutions like Madeus [43] and Concerto [44] can yield precise configuration state models with clear transition conditions, considering interactions with other resource lifecycles. This enables controlled mutable changes in provisioning-focused IaC and formally verifying reconfiguration invariants [55]. Moreover, this approach could enhance deployment and update efficiency by utilizing fine-grained intermediate states and enabling safe, parallel execution of actions.

**Identifying Essential Changes** Our safe DSU approach Essential Safety differentiates between essential changes, i.e., introducing semantically different behavior, and non-essential changes. However, whether a change to a system is essential is generally hard to (safely) predict because it depends on the changed behavior and its context, which decides whether the change is even visible and, if it is, whether the visible change is a semantic difference. In this work, deciding whether a change is essential is excluded and practically pushed to the user. However, research on the nature of essential changes and reliable methods to decide whether a change is essential in a deployment context is crucial. New insights on when a change is essential can also be used to optimize change-related automation beyond safe DSU, e.g., caching and CI/CD pipeline executions.

**Empirical Studies on PL-IaC** Empirical studies on (1) PL-IaC itself, (2) its relation to IaC approaches, and (3) its relation to other software are important to enhance the development of IaC programs, while (2) and (3) are urgently needed to effectively apply and specialize existing software engineering techniques and tools to PL-IaC. PIPr is a suitable starting point for such investigations. Concretely, researchers can assess the IaC programs in PIPr to understand the nature and issues of IaC programs by applying static and dynamic analyses, checking the programs against oracles like CVEs and best practices, and investigating how errors in target states are reflected on the code level.

Considering additional data, e.g., from issue trackers, pull requests, and commit histories, researchers can study the software engineering processes of PL-IaC, e.g., testing and reviewing. Researchers can compare insights from the longitudinal studies above and, e.g., code metrics, used language features (distributions), and evolution patterns with existing insights for traditional software.

Apart from new studies, researchers can replicate previous IaC studies for PL-IaC, enabling synergies with research on other IaC technologies. Researchers can explore, e.g., whether known IaC code smells exist in IaC programs, whether current linters are effective [172, 196, 197, 204, 207, 221], and the applicability of IaC code quality metrics [58, 59].

**Advanced IaC Testing** ACT is an abstract framework that heavily relies on the test generator and oracle plugins. We showed that the approach is already useful with current type-based plugins for ProTI; however, researchers can now use the platform to ideate and evaluate other, more advanced techniques inspired by the plethora of techniques that have been invented for testing and fuzzing.

So far, our test generation uses naïve uniformly distributed and biased random value generation without analyzing the program, previous test runs, and applying heuristics. The literature discussed test generation strategies that are feedback-directed [177], search-based [149], coverage-guided [143, 146], combinatorial-coverage-guided [87], grammar-based [110, 246], and more.

Our current test oracles are also limited because the types used are imprecise and context-insensitive, i.e., they cannot express constraints across properties or resources or leverage domain-specific context like best practices. Beyond additional and more precise sources of ground truth, in the IaC context, researchers can explore proposed oracle techniques, like iterative oracle improvement [119], differential testing [75], metamorphic testing [45], intramorphic testing [205], and learning-based approaches [64, 114, 255].

**Verification of IaC Programs** Verification techniques can prove the absence of errors, i.e., provide ultimate confidence that an IaC program is reliable. While general program verification is practically too effort-intensive, automatically verifying certain domain-specific properties can be (1) technically feasible and (2) critical enough to justify the automation effort. Recent research, e.g., at AWS, demonstrated this idea but is, unfortunately, limited to specific properties and platforms (cf. Section 2.4.3).

Research on IaC verification techniques can leverage a shared-responsibility model where the verification model and automation technique are part of the resource plugins developed by the community and (re)used by all IaC program developers. Developers only add specifications to their IaC programs, which can be automatically verified. This way, the tedious, resource-intensive development of the verification model and automation is centralized, and all resource plugin consumers can use it with low effort.

IaC verification tools can apply to declarative IaC in general and specifically to target states from IaC programs. Developers can verify the set-up infrastructure by running such tools on the target state after deployment and check potential target states before deployment. For pre-deployment checks, the verification tools can be ProTI oracle plugins, verifying the target state in each test run. This combines the feasibility of automated testing with verified certainty for each systematically explored test case.

# Bibliography

[1] Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Juergen Rilling. “Which Commits Can Be CI Skipped?” In: *IEEE Trans. Software Eng.* 47.3 (2021), pp. 448–463. DOI: 10.1109/TSE.2019.2897300.

[2] Amani Abu Jabal, Maryam Davari, Elisa Bertino, Christian Makaya, Seraphin B. Calo, Dinesh C. Verma, Alessandra Russo, and Christopher Williams. “Methods and Tools for Policy Analysis”. In: *ACM Comput. Surv.* 51.6 (2019), 121:1–121:35. DOI: 10.1145/3295749.

[3] Emma Ahrens, Marius Bozga, Radu Iosif, and Joost-Pieter Katoen. “Reasoning about Distributed Reconfigurable Systems”. In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (2022), pp. 145–174. DOI: 10.1145/3563293.

[4] Jonathan Aldrich, Craig Chambers, and David Notkin. “ArchJava: Connecting Software Architecture to Implementation”. In: *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*. 2002, pp. 187–197. DOI: 10.1145/581339.581365.

[5] Amazon Web Services. *Amazon Elastic Container Service*. https://aws.amazon.com/ecs/. Accessed: 2023-11-29.

[6] Amazon Web Services. *Amazon States Language*. https://docs.aws.amazon.com/step-functions/latest/dg/concepts-amazon-states-language.html. Accessed: 2023-12-02.

[7] Amazon Web Services. *Cloud Compute Capazity: Amazon EC2*. https://aws.amazon.com/ec2/. Accessed: 2023-11-29.

[8] Amazon Web Services. *Cloud Development Framework: AWS Cloud Development Kit*. https://aws.amazon.com/cdk/. Accessed: 2023-11-29.

[9] Amazon Web Services. *Cloud Object Storage: Amazon S3*. https://aws.amazon.com/s3/. Accessed: 2023-11-29.

[10] Amazon Web Services. *Constructs: AWS Cloud Development Kit (AWS CDK) v2*. https://docs.aws.amazon.com/cdk/v2/guide/constructs.html. Accessed: 2023-11-28.

[11] Amazon Web Services. *Infrastructure as Code Provisioning Tool: AWS CloudFormation*. https://aws.amazon.com/cloudformation/. Accessed: 2023-11-29.

[12] Amazon Web Services. *Managed SQL Database: Amazon Relational Database Service (RDS)*. https://aws.amazon.com/rds/. Accessed: 2023-11-29.

[13] Amazon Web Services. *Serverless Compute: AWS Fargate*. https://aws.amazon.com/fargate/. Accessed: 2023-11-29.

[14] Amazon Web Services. *Testing Constructs: AWS Cloud Development Kit (AWS CDK) v2*. https://docs.aws.amazon.com/cdk/v2/guide/testing.html. Accessed: 2023-11-29.

[15] Amazon Web Services. *Workflow Orchestration: AWS Step Functions*. https://aws.amazon.com/step-functions. Accessed: 2023-12-02.

[16] David P. Anderson. “BOINC: A System for Public-resource Computing and Storage”. In: *5th International Workshop on Grid Computing (GRID 2004), 8 November 2004, Pittsburgh, PA, USA, Proceedings*. 2004, pp. 4–10. DOI: 10.1109/GRID.2004.14.

[17] Muhammad Asad Arfeen, Krzysztof Pawlikowski, Donald C. McNickle, and Andreas Willig. “The Role of the Weibull Distribution in Internet Traffic Modeling”. In: *25th International Teletraffic Congress, ITC 2013, Shanghai, China, September 10-12, 2013*. 2013, pp. 1–8. DOI: 10.1109/ITC.2013.6662948.

[18] Joe Armstrong. “Erlang”. In: *Commun. ACM* 53.9 (2010), pp. 68–75. DOI: 10.1145/1810891.1810910.

[19] John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan J. Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, Sean McLaughlin, Jason Reed, Neha Rungta, John Sizemore, Mark Stalzer, Preethi Srinivasan, Pavle Subotić, Carsten Varming, and Blake Whaley. “Reachability Analysis for AWS-based Networks”. In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*. Vol. 11562. 2019, pp. 231–241. DOI: 10.1007/978-3-030-25543-5_14.

[20] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Søe Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. “Semantic-based Automated Reasoning for AWS Access Policies Using SMT”. In: *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*. 2018, pp. 1–9. DOI: 10.23919/FMCAD.2018.8602994.

[21] John Backes, Pauline Bolignano, Byron Cook, Andrew Gacek, Kasper Søe Luckow, Neha Rungta, Martin Schäf, Cole Schlesinger, Rima Tanash, Carsten Varming, and Michael W. Whalen. “One-click Formal Methods”. In: *IEEE Softw.* 36.6 (2019), pp. 61–65. DOI: 10.1109/MS.2019.2930609.

[22] Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. “A Survey on Reactive Programming”. In: *ACM Comput. Surv.* 45.4 (2013), 52:1–52:34. DOI: 10.1145/2501654.2501666.

[23] Ioana Baldini, Paul C. Castro, Kerry Shih-Ping Chang, Perry Chang, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. “Serverless Computing: Current Trends and Open Problems”. In: *Research Advances in Cloud Computing*. 2017, pp. 1–20. DOI: 10.1007/978-981-10-5026-8_1.

[24] Luciano Baresi, Carlo Ghezzi, Xiaoxing Ma, and Valerio Panzica La Manna. “Efficient Dynamic Updates of Distributed Components Through Version Consistency”. In: *IEEE Trans. Software Eng.* 43.4 (2017), pp. 340–358. DOI: 10.1109/TSE.2016.2592913.

[25] Luciano Baresi, Giovanni Quattrocchi, Damian Andrew Tamburri, and Luca Terracciano. “A Declarative Modelling Framework for the Deployment and Management of Blockchain Applications”. In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, MODELS 2022, Montreal, Quebec, Canada, October 23-28, 2022*. 2022, pp. 311–321. DOI: 10.1145/3550355.3552417.

[26] Mahi Begoug, Narjes Bessghaier, Ali Ouni, Eman Abdullah AlOmar, and Mohamed Wiem Mkaouer. "What Do Infrastructure-as-Code Practitioners Discuss: An Empirical Study on Stack Overflow". In: *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2023, New Orleans, LA, USA, October 26-27, 2023*. 2023, pp. 1–12. DOI: 10.1109/ESEM56168.2023.10304847.

[27] Julian Bellendorf and Zoltán Ádám Mann. "Specification of Cloud Topologies and Orchestration Using TOSCA: A Survey". In: *Computing* 102.8 (2020), pp. 1793–1815. DOI: 10.1007/S00607-019-00750-3.

[28] Moritz Beller, Georgios Gousios, and Andy Zaidman. "TravisTorrent: Synthesizing Travis CI and GitHub for Full-stack Research on Continuous Integration". In: *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*. 2017, pp. 447–450. DOI: 10.1109/MSR.2017.24.

[29] Boutheina Bennour, Ludovic Henrio, and Marcela Rivera. "A Reconfiguration Framework for Distributed Components". In: *Proceedings of the 2009 ESEC/FSE Workshop on Software Integration and Evolution @ Runtime*. 2009, pp. 49–56. ISBN: 978-1-60558-681-6. DOI: 10.1145/1596495.1596509.

[30] Eduard van der Bent, Jurriaan Hage, Joost Visser, and Georgios Gousios. "How Good Is Your Puppet? An Empirically Defined and Validated Quality Model for Puppet". In: *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*. 2018, pp. 164–174. DOI: 10.1109/SANER.2018.8330206.

[31] Christophe Bidan, Valérie Issarny, Titos Saridakis, and Apostolos V. Zarras. "A Dynamic Reconfiguration Service for CORBA". In: *Fourth International Conference on Configurable Distributed Systems, 1998, Proceedings, Annapolis, MA, USA, 6 May, 1998*. 1998, pp. 35–42. DOI: 10.1109/CDS.1998.675756.

[32] Nemania Borovits, Indika Kumara, Dario Di Nucci, Parvathy Krishnan, Stefano Dalla Palma, Fabio Palomba, Damian Andrew Tamburri, and Willem-Jan van den Heuvel. "FindICI: Using Machine Learning to Detect Linguistic Inconsistencies Between Code and Natural Language Descriptions in Infrastructure-as-Code". In: *Empir. Softw. Eng.* 27.7 (2022), p. 178. DOI: 10.1007/s10664-022-10215-5.

[33] Malik Bouchet, Byron Cook, Bryant Cutler, Anna Druzkina, Andrew Gacek, Liana Hadarean, Ranjit Jhala, Brad Marshall, Daniel Peebles, Neha Rungta, Cole Schlesinger, Chriss Stephens, Carsten Varming, and Andy Warfield. "Block Public Access: Trust Safety Verification of Access Control Policies". In: *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. 2020, pp. 281–291. DOI: 10.1145/3368089.3409728.

[34] Linda Bourque and Eve Fielder. *How to Conduct Self-administered and Mail Surveys*. Vol. 3. 2003. ISBN: 978-0-7619-2562-0.

[35] Uwe Breitenbücher, Tobias Binz, Kálmán Képes, Oliver Kopp, Frank Leymann, and Johannes Wettinger. "Combining Declarative and Imperative Cloud Application Provisioning Based on TOSCA". In: *2014 IEEE International Conference on Cloud Engineering, Boston, MA, USA, March 11-14, 2014*. 2014, pp. 87–96. DOI: 10.1109/IC2E.2014.56.

[36] Alanna Brown, Nigel Kersten, and Michael Stahnke. *2020 State of DevOps Report*. https://www.puppet.com/system/files/2020-State-of-DevOps-Report.pdf. Accessed: 2023-11-27. 2020.

[37] Brendan Burns, Brian Grant, David Oppenheimer, Eric A. Brewer, and John Wilkes. "Borg, Omega, and Kubernetes". In: *Commun. ACM* 59.5 (2016), pp. 50–57. DOI: 10.1145/2890784.

[38] Domenico Calcaterra and Orazio Tomarchio. "Policy-based Holistic Application Management with BPMN and TOSCA". In: *SN Comput. Sci.* 4.3 (2023), p. 232. DOI: 10.1007/s42979-022-01616-w.

[39] Camunda. *Zeebe: Cloud-native Workflow Engine*. https://camunda.com/platform/zeebe/. Accessed: 2023-12-02.

[40] Fabio Casati, Stefano Ceri, Barbara Pernici, and Giuseppe Pozzi. "Workflow Evolution". In: *Data Knowl. Eng.* 24.3 (1998), pp. 211–238. DOI: 10.1016/S0169-023X(97)00033-5.

[41] Claudia Cauli, Meng Li, Nir Piterman, and Oksana Tkachuk. "Pre-deployment Security Assessment for Cloud Services Through Semantic Reasoning". In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*. Vol. 12759. 2021, pp. 767–780. DOI: 10.1007/978-3-030-81685-8_36.

[42] Claudia Cauli, Magdalena Ortiz, and Nir Piterman. "Actions over Core-closed Knowledge Bases". In: *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*. Vol. 13385. 2022, pp. 281–299. DOI: 10.1007/978-3-031-10769-6_17.

[43] Maverick Chardet, Hélène Coullon, Dimitri Pertin, and Christian Pérez. "Madeus: A Formal Deployment Model". In: *2018 International Conference on High Performance Computing & Simulation, HPCS 2018, Orleans, France, July 16-20, 2018*. 2018, pp. 724–731. DOI: 10.1109/HPCS.2018.00118.

[44] Maverick Chardet, Hélène Coullon, and Simon Robillard. "Toward Safe and Efficient Reconfiguration with Concerto". In: *Sci. Comput. Program.* 203 (2021), p. 102582. DOI: 10.1016/J.SCICO.2020.102582.

[45] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. "Metamorphic Testing: A Review of Challenges and Opportunities". In: *ACM Comput. Surv.* 51.1 (2018), 4:1–4:27. DOI: 10.1145/3143561.

[46] Wei Chen, Guoquan Wu, and Jun Wei. "An Approach to Identifying Error Patterns for Infrastructure as Code". In: *2018 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Memphis, TN, USA, October 15-18, 2018*. 2018, pp. 124–129. DOI: 10.1109/ISSREW.2018.00-19.

[47] Michele Chiari, Michele De Pascalis, and Matteo Pradella. "Static Analysis of Infrastructure as Code: A Survey". In: *IEEE 19th International Conference on Software Architecture Companion, ICSA Companion 2022, Honolulu, HI, USA, March 12-15, 2022*. 2022, pp. 218–225. DOI: 10.1109/ICSA-C54293.2022.00049.

[48] Jürgen Cito, Ruzica Piskac, Mark Santolucito, Andy Zaidman, and Daniel Sokolowski. "Resilient Software Configuration and Infrastructure Code Analysis (Dagstuhl Seminar 23082)". In: *Dagstuhl Reports* 13.2 (2023), pp. 163–182. DOI: 10.4230/DAGREP.13.2.163.

[49] Koen Claessen and John Hughes. “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs”. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP ’00), Montreal, Canada, September 18-21, 2000*. 2000, pp. 268–279. DOI: 10.1145/351240.351266.

[50] Gerry Gerard Claps, Richard Berntsson-Svensson, and Aybüke Aurum. “On the Journey to Continuous Deployment: Technical and Social Challenges Along the Way”. In: *Inf. Softw. Technol.* 57 (2015), pp. 21–31. DOI: 10.1016/J.INFSOF.2014.07.009.

[51] Byron Cook. “Formal Reasoning about the Security of Amazon Web Services”. In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. Vol. 10981. 2018, pp. 38–47. DOI: 10.1007/978-3-319-96145-3_3.

[52] Gregory Harold Cooper. “Integrating Dataflow Evaluation into a Practical Higher-order Call-by-value Language”. PhD thesis. Brown University, USA, 2008. ISBN: 978-0-549-89696-8.

[53] Uriel Corfa. *Awesome Monorepo: Notable Public Monorepos*. https://github.com/korfuri/awesome-monorepo#notable-public-monorepos. Accessed: 2023-12-02. 2017.

[54] Flavio Corradini, Fabrizio Fornari, Andrea Polini, Barbara Re, and Francesco Tiezzi. *RePROSitory: A Repository Platform for Sharing Business PROcess modelS*. Ed. by Benoît Depaire, Johannes De Smedt, Marlon Dumas, Dirk Fahland, Akhil Kumar, Henrik Leopold, Manfred Reichert, Stefanie Rinderle-Ma, Stefan Schulte, Stefan Seidel, and Wil M. P. van der Aalst. https://ceur-ws.org/Vol-2420/paperDT7.pdf. Accessed: 2023-11-30. 2019.

[55] Hélène Coullon, Ludovic Henrio, Frédéric Loulergue, and Simon Robillard. “Component-based Distributed Software Reconfiguration: A Verification-oriented Survey”. In: *ACM Comput. Surv.* 56.1 (2024), 2:1–2:37. DOI: 10.1145/3595376.

[56] Luis Cruz, Rui Abreu, and David Lo. “To the Attention of Mobile Software Developers: Guess What, Test Your App!” In: *Empir. Softw. Eng.* 24.4 (2019), pp. 2438–2468. DOI: 10.1007/S10664-019-09701-0.

[57] Daniel Cukier. “DevOps Patterns to Scale Web Applications Using Cloud Services”. In: *SPLASH’13 - The Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity, Indianapolis, IN, USA, October 26-31, 2013*. 2013, pp. 143–152. DOI: 10.1145/2508075.2508432.

[58] Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, and Damian A. Tamburri. “Within-project Defect Prediction of Infrastructure-as-Code Using Product and Process Metrics”. In: *IEEE Trans. Software Eng.* 48.6 (2022), pp. 2086–2104. DOI: 10.1109/TSE.2021.3051492.

[59] Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, and Damian Andrew Tamburri. “Toward a Catalog of Software Quality Metrics for Infrastructure Code”. In: *J. Syst. Softw.* 170 (2020), p. 110726. DOI: 10.1016/J.JSS.2020.110726.

[60] Stefano Dalla Palma, Dario Di Nucci, and Damian Andrew Tamburri. “AnsibleMetrics: A Python Library for Measuring Infrastructure-as-Code Blueprints in Ansible”. In: *SoftwareX* 12 (2020), p. 100633. DOI: 10.1016/J.SOFTX.2020.100633.

[61] Emanuele De Angelis, Fabio Fioravanti, Adrián Palacios, Alberto Pettorossi, and Maurizio Proietti. "Property-based Test Case Generators for Free". In: *Tests and Proofs - 13th International Conference, TAP@FM 2019, Porto, Portugal, October 9-11, 2019, Proceedings*. Vol. 11823. 2019, pp. 186–206. DOI: 10.1007/978-3-030-31157-5_12.

[62] Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli, and Gianluigi Zavattaro. "Aeolus: A Component Model for the Cloud". In: *Inf. Comput.* 239 (2014), pp. 100–121. DOI: 10.1016/J.IC.2014.11.002.

[63] Digital.ai. *14th Annual State of Agile Report*. https://explore.digital.ai/state-of-agile/14th-annual-state-of-agile-report. Accessed: 2020-11-30. 2020.

[64] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K. Lahiri. "TOGA: A Neural Method for Test Oracle Generation". In: *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. 2022, pp. 2130–2141. DOI: 10.1145/3510003.3510141.

[65] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. "Microservices: Yesterday, Today, and Tomorrow". In: *Present and Ulterior Software Engineering*. 2017, pp. 195–216. DOI: 10.1007/978-3-319-67425-4_12.

[66] Nicolas Dubien. *fast-check: Official Documentation*. https://fast-check.dev/. Accessed: 2023-11-29.

[67] Joe Duffy. *Building the Best Infrastructure as Code with $41M Series C Funding*. https://www.pulumi.com/blog/series-c/. Accessed: 2023-11-30. Pulumi Blog, Oct. 2023.

[68] Joe Duffy. *Pulumi Raises Series B to Build the Future of Cloud Engineering*. https://www.pulumi.com/blog/series-b/. Accessed: 2023-11-30. Pulumi Blog, Oct. 2020.

[69] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. "Boa: A Language and Infrastructure for Analyzing Ultra-large-scale Software Repositories". In: *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 2013, pp. 422–431. DOI: 10.1109/ICSE.2013.6606588.

[70] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolás Serrano. "DevOps". In: *IEEE Softw.* 33.3 (2016), pp. 94–100. DOI: 10.1109/MS.2016.68.

[71] Conal Elliott and Paul Hudak. "Functional Reactive Animation". In: *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997*. 1997, pp. 263–273. DOI: 10.1145/258948.258973.

[72] Christian Endres, Uwe Breitenbücher, Michael Falkenthal, Oliver Kopp, Frank Leymann, and Johannes Wettinger. *Declarative Vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications*. https://www.iaas.uni-stuttgart.de/publications/INPROC-2017-12-Declarative-vs-Imperative-Modeling-Patterns.pdf. Accessed: 2023-11-30. 2017.

[73] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. "The Daikon System for Dynamic Detection of Likely Invariants". In: *Sci. Comput. Program.* 69.1-3 (2007), pp. 35–45. DOI: 10.1016/j.scico.2007.01.015.

[74] Alexandros Evangelidis, David Parker, and Rami Bahsoon. “Performance Modelling and Verification of Cloud-based Auto-scaling Policies”. In: *Future Gener. Comput. Syst.* 87 (2018), pp. 629–638. DOI: 10.1016/j.future.2017.12.047.

[75] Robert B. Evans and Alberto Savoia. “Differential Testing: A New Approach to Change Detection”. In: *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*. 2007, pp. 549–552. DOI: 10.1145/1287624.1287707.

[76] Robert S. Fabry. *How to Design a System in Which Modules Can Be Changed on the Fly*. Ed. by Raymond T. Yeh and C. V. Ramamoorthy. http://dl.acm.org/citation.cfm?id=807720. Accessed: 2023-12-02. 1976.

[77] Mattia Fazzini, Alessandra Gorla, and Alessandro Orso. “A Framework for Automated Test Mocking of Mobile Apps”. In: *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. 2020, pp. 1204–1208. DOI: 10.1145/3324884.3418927.

[78] George Fink and Matt Bishop. “Property-based Testing: A New Approach to Testing for Assurance”. In: *ACM SIGSOFT Softw. Eng. Notes* 22.4 (1997), pp. 74–80. DOI: 10.1145/263244.263267.

[79] Nicole Forsgren, Dustin Smith, Jez Humble, and Jessie Frazelle. *2019 Accelerate State of DevOps Report*. https://services.google.com/fh/files/misc/state-of-devops-2019.pdf. Accessed: 2023-11-28. 2019.

[80] Martin Fowler. *Blue Green Deployment*. https://martinfowler.com/bliki/BlueGreenDeployment.html. Accessed: 2023-12-02. Mar. 2010.

[81] Keheliya Gallaba, John Ewart, Yves Junqueira, and Shane McIntosh. “Accelerating Continuous Integration by Caching Environments and Inferring Dependencies”. In: *IEEE Trans. Software Eng.* 48.6 (2022), pp. 2040–2052. DOI: 10.1109/TSE.2020.3048335.

[82] Hector Garcia-Molina and Kenneth Salem. “Sagas”. In: *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, CA, USA, May 27-29, 1987*. 1987, pp. 249–259. DOI: 10.1145/38713.38742.

[83] Matthias Geiger, Simon Harrer, Jörg Lenhard, and Guido Wirtz. “BPMN 2.0: The State of Support and Implementation”. In: *Future Gener. Comput. Syst.* 80 (2018), pp. 250–262. DOI: 10.1016/J.FUTURE.2017.01.006.

[84] Matthias Geiger, Simon Harrer, Jörg Lenhard, and Guido Wirtz. “On the Evolution of BPMN 2.0 Support and Implementation”. In: *2016 IEEE Symposium on Service-Oriented System Engineering, SOSE 2016, Oxford, United Kingdom, March 29 - April 2, 2016*. 2016, pp. 101–110. DOI: 10.1109/SOSE.2016.39.

[85] GitHub. *Github Docs: Searching Code (Legacy)*. https://docs.github.com/en/search-github/searching-on-github/searching-code. Accessed: 2023-11-30.

[86] GitHub. *Licensing a Repository*. https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/licensing-a-repository. Accessed: 2023-11-30.

[87] Harrison Goldstein, John Hughes, Leonidas Lampropoulos, and Benjamin C. Pierce. “Do Judge a Test by Its Cover - Combining Combinatorial and Property-based Testing”. In: *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*. Vol. 12648. 2021, pp. 264–291. DOI: 10.1007/978-3-030-72019-3_10.

[88] Durham Goode and Rain. *Scaling Mercurial at Facebook*. https://engineering.fb.com/2014/01/07/core-data/scaling-mercurial-at-facebook/. Accessed: 2023-12-02. Jan. 2014.

[89] Leo A. Goodman. “Snowball Sampling”. In: *The Annals of Mathematical Statistics* 32.1 (1961), pp. 148–170. ISSN: 00034851.

[90] Google. *Google Forms: Online Form Creator*. https://www.google.com/forms/about/. Accessed: 2023-12-22.

[91] Google. *Google Sheets: Online Spreadsheet Editor*. https://www.google.com/sheets/about/. Accessed: 2023-12-22.

[92] Google Cloud. *Workflows*. https://cloud.google.com/workflows. Accessed: 2023-12-02.

[93] Georgios Gousios. “The GHTorrent Dataset and Tool Suite”. In: *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR ’13, San Francisco, CA, USA, May 18-19, 2013*. 2013, pp. 233–236. DOI: 10.1109/MSR.2013.6624034.

[94] Paul W. P. J. Grefen, Jochem Vonk, and Peter M. G. Apers. “Global Transaction Support for Workflow Management Systems: From Formal Specification to Practical Implementation”. In: *VLDB J.* 10.4 (2001), pp. 316–333. DOI: 10.1007/S007780100056.

[95] Boris Grubic, Yang Wang, Tyler Petrochko, Ran Yaniv, Brad Jones, David Callies, Matt Clarke-Lauer, Dan Kelley, Soteris Demetriou, Kenny Yu, and Chunqiang Tang. “Conveyor: One-tool-fits-all Continuous Software Deployment at Meta”. In: *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*. 2023, pp. 325–342. ISBN: 978-1-939133-34-2.

[96] Tianxiao Gu, Xiaoxing Ma, Chang Xu, Yanyan Jiang, Chun Cao, and Jian Lu. “Automating Object Transformations for Dynamic Software Updating via Online Execution Synthesis”. In: *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*. Vol. 109. 2018, 19:1–19:28. DOI: 10.4230/LIPICS.ECOOP.2018.19.

[97] Michele Guerriero, Martin Garriga, Damian Andrew Tamburri, and Fabio Palomba. “Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry”. In: *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*. 2019, pp. 580–589. DOI: 10.1109/ICSME.2019.00092.

[98] Deepak Gupta, Pankaj Jalote, and Gautam Barua. “A Formal Framework for On-line Software Version Change”. In: *IEEE Trans. Software Eng.* 22.2 (1996), pp. 120–131. DOI: 10.1109/32.485222.

[99] Oliver Hanappi, Waldemar Hummer, and Schahram Dustdar. “Asserting Reliable Convergence for Configuration Management Scripts”. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. 2016, pp. 328–343. DOI: 10.1145/2983990.2984000.

[100] Lukas Harzenetter, Uwe Breitenbücher, Tobias Binz, and Frank Leymann. “An Integrated Management System for Composed Applications Deployed by Different Deployment Automation Technologies”. In: *SN Comput. Sci.* 4.4 (2023), p. 370. DOI: 10.1007/s42979-023-01810-4.

[101] HashiCorp. *CDK for Terraform*. https://developer.hashicorp.com/terraform/cdktf. Accessed: 2023-11-29.

[102] HashiCorp. *hashicorp/http-echo*. https://hub.docker.com/r/hashicorp/http-echo/. Accessed: 2023-11-29.

[103] HashiCorp. *HCL*. https://github.com/hashicorp/hcl. Accessed: 2023-11-30.

[104] HashiCorp. *Terraform*. https://www.terraform.io/. Accessed: 2023-11-29.

[105] HashiCorp. *Unit Tests: CDK for Terraform*. https://developer.hashicorp.com/terraform/cdktf/test/unit-tests. Accessed: 2023-11-29.

[106] Mohammad Mehedi Hassan and Akond Rahman. “As Code Testing: Characterizing Test Quality in Open Source Ansible Development”. In: *15th IEEE Conference on Software Testing, Verification and Validation, ICST 2022, Valencia, Spain, April 4-14, 2022*. 2022, pp. 208–219. DOI: 10.1109/ICST53961.2022.00031.

[107] Aki Helin. *Radamsa: A General-purpose Fuzzer*. https://gitlab.com/akihe/radamsa. Accessed: 2023-11-30.

[108] Michael W. Hicks and Scott Nettles. “Dynamic Software Updating”. In: *ACM Trans. Program. Lang. Syst.* 27.6 (2005), pp. 1049–1096. DOI: 10.1145/1108970.1108971.

[109] Luke Hoban. *Introducing AWS CDK on Pulumi*. https://www.pulumi.com/blog/aws-cdk-on-pulumi/. Accessed: 2023-11-30. Pulumi Blog, May 2022.

[110] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. “Grammarinator: A Grammar-based Open Source Fuzzer”. In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST@SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 05, 2018*. 2018, pp. 45–48. DOI: 10.1145/3278186.3278193.

[111] Helena Holmström Olsson, Hiva Allahyari, and Jan Bosch. “Climbing the "Stairway to Heaven" - A Mulitiple-case Study Exploring Barriers in the Transition from Agile Development Towards Continuous Deployment of Software”. In: *38th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2012, Cesme, Izmir, Turkey, September 5-8, 2012*. 2012, pp. 392–399. DOI: 10.1109/SEAA.2012.54.

[112] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. 2010. ISBN: 978-0-321-60191-9.

[113] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. “Testing Idempotence for Infrastructure as Code”. In: *Middleware 2013 - ACM/IFIP/USENIX 14th International Middleware Conference, Beijing, China, December 9-13, 2013, Proceedings*. Vol. 8275. 2013, pp. 368–388. DOI: 10.1007/978-3-642-45065-5_19.

[114] Ali Reza Ibrahimzada, Yigit Varli, Dilara Tekinoglu, and Reyhaneh Jabbarvand. “Perfect Is the Enemy of Test Oracle”. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. 2022, pp. 70–81. DOI: 10.1145/3540250.3549086.

[115] IDC and Statista. *Worldwide Technology Employment Impact Guide*. https://www.statista.com/statistics/1126677/it-employment-worldwide/. Accessed: 2021-02-03. Dec. 2019.

[116] Katsuhiko Ikeshita, Fuyuki Ishikawa, and Shinich Honiden. “Test Suite Reduction in Idempotence Testing of Infrastructure as Code”. In: *Tests and Proofs - 11th International Conference, TAP@STAF 2017, Marburg, Germany, July 19-20, 2017, Proceedings*. Vol. 10375. 2017, pp. 98–115. DOI: 10.1007/978-3-319-61467-0_6.

[117] Chadni Islam, Victor Prokhorenko, and Ali Babar. “Runtime Software Patching: Taxonomy, Survey and Future Directions”. In: *J. Syst. Softw.* 200 (2023), p. 111652. DOI: 10.1016/J.JSS.2023.111652.

[118] Daniel Jackson. “Alloy: A Lightweight Object Modelling Notation”. In: *ACM Trans. Softw. Eng. Methodol.* 11.2 (2002), pp. 256–290. DOI: 10.1145/505145.505149.

[119] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. “Test Oracle Assessment and Improvement”. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. 2016, pp. 247–258. DOI: 10.1145/2931037.2931062.

[120] Yujuan Jiang and Bram Adams. “Co-evolution of Infrastructure and Source Code - An Empirical Study”. In: *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*. 2015, pp. 45–55. DOI: 10.1109/MSR.2015.12.

[121] Lara Lorna Jiménez and Olov Schelén. “DOCMA: A Decentralized Orchestrator for Containerized Microservice Applications”. In: *2019 IEEE Cloud Summit*. 2019, pp. 45–51. DOI: 10.1109/CloudSummit47114.2019.00014.

[122] Shrinivas Joshi and Alessandro Orso. “SCARPE: A Technique and Tool for Selective Capture and Replay of Program Executions”. In: *23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France*. 2007, pp. 234–243. DOI: 10.1109/ICSM.2007.4362636.

[123] Mark Kasunic. “Designing an Effective Survey”. In: (2005). DOI: 10.1184/R1/6573062.v1.

[124] David Kawrykow and Martin P. Robillard. “Non-essential Changes in Version Histories”. In: *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. 2011, pp. 351–360. DOI: 10.1145/1985793.1985842.

[125] Gene Kim, Jez Humble, Patrick Debois, John Willis, and Nicole Forsgren. *The DevOps Handbook: How to Create World-class Agility, Reliability, & Security in Technology Organizations*. Second. 2021. ISBN: 978-1-950508-40-2.

[126] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. “TeaStore: A Micro-service Reference Application for Benchmarking, Modeling and Resource Management Research”. In: *26th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2018, Milwaukee, WI, USA, September 25-28, 2018*. 2018, pp. 223–236. DOI: 10.1109/MASCOTS.2018.00030.

[127] Barbara A. Kitchenham and Shari Lawrence Pfleeger. “Personal Opinion Surveys”. In: *Guide to Advanced Empirical Software Engineering*. 2008, pp. 63–92. DOI: 10.1007/978-1-84800-044-5_3.

[128] Jeff Kramer and Jeff Magee. “The Evolving Philosophers Problem: Dynamic Change Management”. In: *IEEE Trans. Software Eng.* 16.11 (1990), pp. 1293–1306. DOI: 10.1109/32.60317.

[129] Ingolf Krüger, Barry Demchak, and Massimiliano Menarini. “Dynamic Service Composition and Deployment with OpenRichServices”. In: *Software Service and Application Engineering - Essays Dedicated to Bernd Krämer on the Occasion of His 65th Birthday*. Vol. 7365. 2012, pp. 120–146. DOI: 10.1007/978-3-642-30835-2_9.

[130] Kubernetes. *Kubernetes*. https://kubernetes.io/. Accessed: 2023-12-02.

[131] Kubernetes. *Service*. https://kubernetes.io/docs/concepts/services-networking/service/. Accessed: 2023-12-26.

[132] Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo. “Software Fault Interactions and Implications for Software Testing”. In: *IEEE Trans. Software Eng.* 30.6 (2004), pp. 418–421. DOI: 10.1109/TSE.2004.24.

[133] Rick Kuhn, Yu Lei, and Raghu Kacker. “Practical Combinatorial Testing: Beyond Pairwise”. In: *IT Prof.* 10.3 (2008), pp. 19–23. DOI: 10.1109/MITP.2008.54.

[134] Indika Kumara, Martin Garriga, Angel Urbano Romeu, Dario Di Nucci, Fabio Palomba, Damian Andrew Tamburri, and Willem-Jan van den Heuvel. “The Do’s and Don’ts of Infrastructure Code: A Systematic Gray Literature Review”. In: *Inf. Softw. Technol.* 137 (2021), p. 106593. DOI: 10.1016/J.INFSOF.2021.106593.

[135] Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. “Beginner’s Luck: A Language for Property-based Generators”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 2017, pp. 114–129. DOI: 10.1145/3009837.3009868.

[136] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. “Coverage Guided, Property-based Testing”. In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019), 181:1–181:29. DOI: 10.1145/3360607.

[137] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. “Generating Good Generators for Inductive Relations”. In: *Proc. ACM Program. Lang.* 2.POPL (2018), 45:1–45:30. DOI: 10.1145/3158133.

[138] Frederic Lardinois. *Microsoft Now Uses Git and GVFS to Develop Windows*. https://techcrunch.com/2017/05/24/microsoft-now-uses-git-and-gvfs-to-develop-windows/. Accessed: 2023-12-02. May 2017.

[139] Eero I. Laukkanen, Juha Itkonen, and Casper Lassenius. “Problems, Causes and Solutions When Adopting Continuous Delivery - A Systematic Literature Review”. In: *Inf. Softw. Technol.* 82 (2017), pp. 55–79. DOI: 10.1016/J.INFSOF.2016.10.001.

[140] Leonardo A. F. Leite, Carla Rocha, Fabio Kon, Dejan S. Milojicic, and Paulo Meirelles. “A Survey of DevOps Concepts and Challenges”. In: *ACM Comput. Surv.* 52.6 (2020), 127:1–127:35. DOI: 10.1145/3359981.

[141] Julien Lepiller, Ruzica Piskac, Martin Schäf, and Mark Santolucito. “Analyzing Infrastructure as Code to Prevent Intra-update Sniping Vulnerabilities”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*. Vol. 12652. 2021, pp. 105–123. DOI: 10.1007/978-3-030-72013-1_6.

[142] James Lewis and Martin Fowler. *Microservices: A Definition of This New Architectural Term*. http://martinfowler.com/articles/microservices.html. Accessed: 2023-12-02. Mar. 2014.

[143] Jun Li, Bodong Zhao, and Chao Zhang. “Fuzzing: A Survey”. In: *Cybersecur.* 1.1 (2018), p. 6. DOI: 10.1186/S42400-018-0002-Y.

[144] *Licensee: A Ruby Gem to Detect Under What License a Project Is Distributed*. https://licensee.github.io/licensee/. Accessed: 2023-11-30.

[145] Changbin Liu, Boon Thau Loo, and Yun Mao. “Declarative Automated Cloud Resource Orchestration”. In: *ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC ’11, Cascais, Portugal, October 26-28, 2011*. 2011, p. 26. DOI: 10.1145/2038916.2038942.

[146] Dongge Liu, Gidon Ernst, Toby Murray, and Benjamin I. P. Rubinstein. “LEGION: Best-first Concolic Testing”. In: *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. 2020, pp. 54–65. DOI: 10.1145/3324884.3416629.

[147] Praneet Loke. *Dynamic Providers*. https://www.pulumi.com/blog/dynamic-providers/. Accessed: 2023-11-30. Jan. 2020.

[148] Andreas Löscher and Konstantinos Sagonas. “Automating Targeted Property-based Testing”. In: *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*. 2018, pp. 70–80. DOI: 10.1109/ICST.2018.00017.

[149] Andreas Löscher and Konstantinos Sagonas. “Targeted Property-based Testing”. In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*. 2017, pp. 46–56. DOI: 10.1145/3092703.3092711.

[150] Aimee Lucido. *Monorepo to Multirepo and Back Again*. https://www.youtube.com/watch?v=lV8-1S28ycM. Accessed: 2023-12-02. 2017.

[151] Lucy Ellen Lwakatare, Terhi Kilamo, Teemu Karvonen, Tanja Sauvola, Ville Heikkilä, Juha Itkonen, Pasi Kuvaja, Tommi Mikkonen, Markku Oivo, and Casper Lassenius. “DevOps in Practice: A Multiple Case Study of Five Companies”. In: *Inf. Softw. Technol.* 114 (2019), pp. 217–230. DOI: 10.1016/J.INFSOF.2019.06.010.

[152] Xiaoxing Ma, Luciano Baresi, Carlo Ghezzi, Valerio Panzica La Manna, and Jian Lu. “Version-consistent Dynamic Reconfiguration of Component-based Distributed Systems”. In: *SIGSOFT/FSE’11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC’11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. 2011, pp. 245–255. DOI: 10.1145/2025113.2025148.

[153] Matej Madeja, Jaroslav Porubän, Michaela Bacíková, Matús Sulír, Ján Juhár, Sergej Chodarev, and Filip Gurbál. “Automating Test Case Identification in Java Open Source Projects on GitHub”. In: *Comput. Informatics* 40.3 (2021). DOI: 10.31577/CAI_2021_3_575.

[154] Matej Madeja, Jaroslav Porubän, Sergej Chodarev, Matúš Sulír, and Filip Gurbál’. “Empirical Study of Test Case and Test Framework Presence in Public Projects on GitHub”. In: *Applied Sciences* 11.16 (2021). ISSN: 2076-3417. DOI: 10.3390/app11167250.

[155] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. “Composing Adaptive Software”. In: *Computer* 37.7 (2004), pp. 56–64. DOI: 10.1109/MC.2004.48.

[156] Nenad Medvidovic and Richard N. Taylor. “A Classification and Comparison Framework for Software Architecture Description Languages”. In: *IEEE Trans. Software Eng.* 26.1 (2000), pp. 70–93. DOI: 10.1109/32.825767.

[157] Meta Platforms. *Jest: Delightful JavaScript Testing*. https://jestjs.io/. Accessed: 2023-11-29.

[158] Bertrand Meyer. *Object-oriented Software Construction, 2nd Edition*. 1997. ISBN: 0-13-629155-4.

[159] Microsoft Azure. *Azure Resource Manager*. https://azure.microsoft.com/en-us/features/resource-manager/. Accessed: 2023-11-29.

[160] Microsoft Azure. *Bicep*. https://github.com/Azure/bicep. Accessed: 2023-11-30.

[161] Frederic Montagut, Refik Molva, and Silvan Tecumseh Golega. “The Pervasive Workflow: A Decentralized Workflow System Supporting Long-running Transactions”. In: *IEEE Trans. Syst. Man Cybern. Part C* 38.3 (2008), pp. 319–333. DOI: 10.1109/TSMCC.2008.919184.

[162] Kief Morris. *Infrastructure as Code: Dynamic Systems for the Cloud Age*. Second. 2021. ISBN: 978-1-0981-1467-1.

[163] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. “Curating GitHub for Engineered Software Projects”. In: *Empir. Softw. Eng.* 22.6 (2017), pp. 3219–3253. DOI: 10.1007/S10664-017-9512-6.

[164] Netflix. *Conductor*. https://conductor.netflix.com/. Accessed: 2023-12-02.

[165] Sam Newman. *Building Microservices*. Second. 2021. ISBN: 978-1-4920-3402-5.

[166] Northern.tech. *CFEngine*. https://cfengine.com/. Accessed: 2024-01-03.

[167] Kristian Nybom, Jens Smeds, and Ivan Porres. "On the Impact of Mixing Responsibilities Between Devs and Ops". In: *Agile Processes, in Software Engineering, and Extreme Programming - 17th International Conference, XP 2016, Edinburgh, UK, May 24-27, 2016, Proceedings*. Vol. 251. 2016, pp. 131–143. DOI: 10.1007/978-3-319-33515-5_11.

[168] OASIS. *Topology and Orchestration Specification for Cloud Applications Version 1.0*. http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html. Accessed: 2023-11-29. 2013.

[169] OASIS. *Web Services Business Process Execution Language Version 2.0*. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. Accessed: 2023-12-02. 2007.

[170] Object Management Group. *Business Process Model and Notation Version 2.0.2*. https://www.omg.org/spec/BPMN. Accessed: 2023-12-02. 2014.

[171] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. "Andromeda: A Dataset of Ansible Galaxy Roles and Their Evolution". In: *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*. 2021, pp. 580–584. DOI: 10.1109/MSR52588.2021.00078.

[172] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. "Control and Data Flow in Security Smell Detection for Infrastructure as Code: Is It Worth the Effort?" In: *20th IEEE/ACM International Conference on Mining Software Repositories, MSR 2023, Melbourne, Australia, May 15-16, 2023*. 2023, pp. 534–545. DOI: 10.1109/MSR59073.2023.00079.

[173] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. "Smelly Variables in Ansible Infrastructure Code: Detection, Prevalence, and Lifetime". In: *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. 2022, pp. 61–72. DOI: 10.1145/3524842.3527964.

[174] Ruben Opdebeeck, Ahmed Zerouali, Camilo Velázquez-Rodríguez, and Coen De Roover. "Does Infrastructure as Code Adhere to Semantic Versioning? An Analysis of Ansible Role Evolution". In: *20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide, Australia, September 28 - October 2, 2020*. 2020, pp. 238–248. DOI: 10.1109/SCAM51674.2020.00032.

[175] Ruben Opdebeeck, Ahmed Zerouali, Camilo Velázquez-Rodríguez, and Coen De Roover. "On the Practice of Semantic Versioning for Ansible Galaxy Roles: An Empirical Study and a Change Classification Model". In: *J. Syst. Softw.* 182 (2021), p. 111059. DOI: 10.1016/j.jss.2021.111059.

[176] Dorothy Ordogh. *Pants and Monorepos*. https://www.youtube.com/watch?v=IL6LBWNi3fE. Accessed: 2023-12-02. 2018.

[177] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. "Feedback-directed Random Test Generation". In: *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. 2007, pp. 75–84. DOI: 10.1109/ICSE.2007.37.

[178] Zoe Paraskevopoulou, Catalin Hritcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. “Foundational Property-based Testing”. In: *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*. Vol. 9236. 2015, pp. 325–343. DOI: 10.1007/978-3-319-22102-1_22.

[179] Rachel Potvin and Josh Levenberg. “Why Google Stores Billions of Lines of Code in a Single Repository”. In: *Commun. ACM* 59.7 (2016), pp. 78–87. DOI: 10.1145/2854146.

[180] Progress. *Chef Software DevOps Automation Solutions*. https://chef.io. Accessed: 2023-11-29.

[181] Pulumi. *Crosswalk for AWS Guides*. https://www.pulumi.com/docs/clouds/aws/guides/. Accessed: 2023-11-29.

[182] Pulumi. *Integration Testing for Pulumi Programs*. https://www.pulumi.com/docs/using-pulumi/testing/integration/. Accessed: 2023-11-29.

[183] Pulumi. *Policies for AWS (AWSGuard)*. https://www.pulumi.com/docs/using-pulumi/crossguard/awsguard/. Accessed: 2023-11-29.

[184] Pulumi. *Policy as Code for Any Cloud with Pulumi: Pulumi CrossGuard*. https://www.pulumi.com/crossguard/. Accessed: 2023-11-29.

[185] Pulumi. *Pulumi Automation API*. https://www.pulumi.com/automation/. Accessed: 2023-12-20.

[186] Pulumi. *Pulumi: Infrastructure as Code in Any Programming Language*. https://github.com/pulumi/pulumi. Accessed: 2023-11-29.

[187] Pulumi. *Stacks: Stack References*. https://www.pulumi.com/docs/concepts/stack/#stackreferences. Accessed: 2023-11-29.

[188] Pulumi. *Testing of Pulumi Programs*. https://www.pulumi.com/docs/using-pulumi/testing/. Accessed: 2023-11-29.

[189] Puppet. *Puppet Infrastructure & IT Automation at Scale*. https://puppet.com/. Accessed: 2023-11-29.

[190] Giovanni Quattrocchi and Damian Andrew Tamburri. “Predictive Maintenance of Infrastructure Code Using "Fluid" Datasets: An Exploratory Study on Ansible Defect Proneness”. In: *J. Softw. Evol. Process.* 34.11 (2022). DOI: 10.1002/smr.2480.

[191] Akond Rahman, Farhat Lamia Barsha, and Patrick Morrison. “Shhh!: 12 Practices for Secret Management in Infrastructure as Code”. In: *IEEE Secure Development Conference, SecDev 2021, Atlanta, GA, USA, October 18-20, 2021*. 2021, pp. 56–62. DOI: 10.1109/SecDev51306.2021.00024.

[192] Akond Rahman, Effat Farhana, Chris Parnin, and Laurie A. Williams. “Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts”. In: *ICSE ’20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. 2020, pp. 752–764. DOI: 10.1145/3377811.3380409.

[193] Akond Rahman, Effat Farhana, and Laurie A. Williams. “The ’As Code’ Activities: Development Anti-patterns for Infrastructure as Code”. In: *Empir. Softw. Eng.* 25.5 (2020), pp. 3430–3467. DOI: 10.1007/s10664-020-09841-8.

[194] Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie A. Williams. “A Systematic Mapping Study of Infrastructure as Code Research”. In: *Inf. Softw. Technol.* 108 (2019), pp. 65–77. DOI: 10.1016/j.infsof.2018.12.004.

[195] Akond Rahman and Chris Parnin. “Detecting and Characterizing Propagation of Security Weaknesses in Puppet-based Infrastructure Management”. In: *IEEE Trans. Software Eng.* 49.6 (2023), pp. 3536–3553. DOI: 10.1109/TSE.2023.3265962.

[196] Akond Rahman, Chris Parnin, and Laurie A. Williams. “The Seven Sins: Security Smells in Infrastructure as Code Scripts”. In: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 2019, pp. 164–175. DOI: 10.1109/ICSE.2019.00033.

[197] Akond Rahman, Md. Rayhanur Rahman, Chris Parnin, and Laurie A. Williams. “Security Smells in Ansible and Chef Scripts: A Replication Study”. In: *ACM Trans. Softw. Eng. Methodol.* 30.1 (2021), 3:1–3:31. DOI: 10.1145/3408897.

[198] Akond Rahman, Shazibul Islam Shamim, Dibyendu Brinto Bose, and Rahul Pandita. “Security Misconfigurations in Open Source Kubernetes Manifests: An Empirical Study”. In: *ACM Trans. Softw. Eng. Methodol.* 32.4 (2023), 99:1–99:36. DOI: 10.1145/3579639.

[199] Akond Rahman and Tushar Sharma. “Lessons from Research to Practice on Writing Better Quality Puppet Scripts”. In: *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*. 2022, pp. 63–67. DOI: 10.1109/SANER53432.2022.00019.

[200] Akond Rahman and Laurie A. Williams. “Different Kind of Smells: Security Smells in Infrastructure as Code Scripts”. In: *IEEE Secur. Priv.* 19.3 (2021), pp. 33–41. DOI: 10.1109/MSEC.2021.3065190.

[201] Akond Rahman and Laurie A. Williams. “Source Code Properties of Defective Infrastructure as Code Scripts”. In: *Inf. Softw. Technol.* 112 (2019), pp. 148–163. DOI: 10.1016/j.infsof.2019.04.013.

[202] Rajiv Ranjan, Boualem Benatallah, Schahram Dustdar, and Michael P. Papazoglou. “Cloud Resource Orchestration Programming: Overview, Issues, and Directions”. In: *IEEE Internet Comput.* 19.5 (2015), pp. 46–56. DOI: 10.1109/MIC.2015.20.

[203] Red Hat. *Ansible Is Simple IT Automation*. https://www.ansible.com/. Accessed: 2023-11-29.

[204] Sofia Reis, Rui Abreu, Marcelo d’Amorim, and Daniel Fortunato. “Leveraging Practitioners’ Feedback to Improve a Security Linter”. In: *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. 2022, 66:1–66:12. DOI: 10.1145/3551349.3560419.

[205] Manuel Rigger and Zhendong Su. “Intramorphic Testing: A New Approach to the Test Oracle Problem”. In: *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2022, Auckland, New Zealand, December 8-10, 2022*. 2022, pp. 128–136. DOI: 10.1145/3563835.3567662.

[206] Bernd Rücker. *trip-booking-saga-serverless*. https://github.com/berndruecker/trip-booking-saga-serverless. Accessed: 2023-12-02. 2019.

[207] Nuno Saavedra and João F. Ferreira. “GLITCH: Automated Polyglot Security Smell Detection in Infrastructure as Code”. In: *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. 2022, 47:1–47:12. DOI: 10.1145/3551349.3556945.

[208] Isac Sacchi e Souza, Daniel Pinheiro Franco, and João Pedro São Gregorio Silva. “Infrastructure as Code as a Foundational Technique for Increasing the DevOps Maturity Level: Two Case Studies”. In: *IEEE Softw.* 40.1 (2023), pp. 63–68. DOI: 10.1109/MS.2022.3213228.

[209] David Saff and Michael D. Ernst. “Mock Object Creation for Test Factoring”. In: *Proceedings of the 2004 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE’04, Washington, DC, USA, June 7-8, 2004*. 2004, pp. 49–51. DOI: 10.1145/996821.996838.

[210] Marcio Augusto Sekeff Sallem and Francisco José da Silva e Silva. “Adapta: A Framework for Dynamic Reconfiguration of Distributed Applications”. In: *Proceedings of the 5th workshop on Adaptive and reflective middleware, ARM 2006, Melbourne, Australia, November 27 - December 1, 2006*. 2006, p. 10. DOI: 10.1145/1175855.1175865.

[211] Guido Salvaneschi, Mirko Köhler, Daniel Sokolowski, Philipp Haller, Sebastian Erdweg, and Mira Mezini. “Language-integrated Privacy-aware Distributed Queries”. In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), 167:1–167:30. DOI: 10.1145/3360593.

[212] Guido Salvaneschi, Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. “On the Positive Effect of Reactive Programming on Software Comprehension: An Empirical Study”. In: *IEEE Trans. Software Eng.* 43.12 (2017), pp. 1125–1143. DOI: 10.1109/TSE.2017.2655524.

[213] Julio Sandobalin, Emilio Insfrán, and Silvia Abrahão. “An Infrastructure Modelling Tool for Cloud Provisioning”. In: *2017 IEEE International Conference on Services Computing, SCC 2017, Honolulu, HI, USA, June 25-30, 2017*. 2017, pp. 354–361. DOI: 10.1109/SCC.2017.52.

[214] Julio Sandobalin, Emilio Insfrán, and Silvia Abrahão. “On the Effectiveness of Tools to Support Infrastructure as Code: Model-driven Versus Code-centric”. In: *IEEE Access* 8 (2020), pp. 17734–17761. DOI: 10.1109/ACCESS.2020.2966597.

[215] Danilo Sato. *Carnary Release*. https://martinfowler.com/bliki/CanaryRelease.html. Accessed: 2023-12-02. June 2014.

[216] Danilo Sato. *Parallel Change*. https://martinfowler.com/bliki/ParallelChange.html. Accessed: 2023-12-02. May 2014.

[217] Julian Schwarz, Andreas Steffens, and Horst Lichter. “Code Smells in Infrastructure as Code”. In: *11th International Conference on the Quality of Information and Communications Technology, QUATIC 2018, Coimbra, Portugal, September 4-7, 2018*. 2018, pp. 220–228. DOI: 10.1109/QUATIC.2018.00040.

[218] Habib Seifzadeh, Hassan Abolhassani, and Mohsen Sadighi Moshkenani. “A Survey of Dynamic Software Updating”. In: *J. Softw. Evol. Process.* 25.5 (2013), pp. 535–568. DOI: 10.1002/SMR.1556.

[219] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices". In: *IEEE Access* 5 (2017), pp. 3909–3943. DOI: 10.1109/ACCESS.2017.2685629.

[220] Rian Shambaugh, Aaron Weiss, and Arjun Guha. "Rehearsal: A Configuration Verification Tool for Puppet". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 2016, pp. 416–430. DOI: 10.1145/2908080.2908083.

[221] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. "Does Your Configuration Code Smell?" In: *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*. 2016, pp. 189–200. DOI: 10.1145/2901739.2901761.

[222] Pavneet Singh Kochhar, Tegawendé F. Bissyandé, David Lo, and Lingxiao Jiang. "An Empirical Study of Adoption of Software Testing in Open Source Projects". In: *2013 13th International Conference on Quality Software, Najing, China, July 29-30, 2013*. 2013, pp. 103–112. DOI: 10.1109/QSIC.2013.57.

[223] Marigianna Skouradaki, Vasilios Andrikopoulos, and Frank Leymann. "Representative BPMN 2.0 Process Model Generation from Recurring Structures". In: *IEEE International Conference on Web Services, ICWS 2016, San Francisco, CA, USA, June 27 - July 2, 2016*. 2016, pp. 468–475. DOI: 10.1109/ICWS.2016.67.

[224] Marigianna Skouradaki, Dieter H. Roller, Frank Leymann, Vincenzo Ferme, and Cesare Pautasso. "On the Road to Benchmarking BPMN 2.0 Workflow Engines". In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, Austin, TX, USA, January 31 - February 4, 2015*. 2015, pp. 301–304. DOI: 10.1145/2668930.2695527.

[225] Daniel Sokolowski. "Deployment Coordination for Cross-functional DevOps Teams". In: *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. 2021, pp. 1630–1634. DOI: 10.1145/3468264.3473101.

[226] Daniel Sokolowski. "Infrastructure as Code for Dynamic Deployments". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. 2022, pp. 1775–1779. DOI: 10.1145/3540250.3558912.

[227] Daniel Sokolowski, Jan-Patrick Lehr, Christian H. Bischof, and Guido Salvaneschi. "Leveraging Hybrid Cloud HPC with Multitier Reactive Programming". In: *IEEE/ACM International Workshop on Interoperability of Supercomputing and Cloud Technologies, SuperCompCloud@SC 2020, Atlanta, GA, USA, November 11, 2020*. 2020, pp. 27–32. DOI: 10.1109/SUPERCOMPCLOUD51944.2020.00010.

[228] Daniel Sokolowski and Guido Salvaneschi. "Towards Reliable Infrastructure as Code". In: *20th International Conference on Software Architecture, ICSA 2023 - Companion, L'Aquila, Italy, March 13-17, 2023*. 2023, pp. 318–321. DOI: 10.1109/ICSA-C57050.2023.00072.

[229] Daniel Sokolowski, David Spielmann, and Guido Salvaneschi. “Automated Infrastructure as Code Program Testing”. In: *IEEE Transactions on Software Engineering* (2024). DOI: 10.1109/TSE.2024.3393070.

[230] Daniel Sokolowski, David Spielmann, and Guido Salvaneschi. *PIPr: A Dataset of Public Infrastructure as Code Programs*. Version 1.1. Nov. 2023. DOI: 10.5281/zenodo.10173400.

[231] Daniel Sokolowski, David Spielmann, and Guido Salvaneschi. *ProTI: Automated Unit Testing of Pulumi TypeScript Infrastructure as Code Programs*. 2023. DOI: 10.5281/zenodo.10028479.

[232] Daniel Sokolowski, David Spielmann, and Guido Salvaneschi. “The PIPr Dataset of Public Infrastructure as Code Programs”. In: *21th IEEE/ACM International Conference on Mining Software Repositories, MSR 2024, Lisbon, Portugal, April 15-16, 2024*. 2024, pp. 498–503. DOI: 10.1145/3643991.3644888.

[233] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. “Automating Serverless Deployments for DevOps Organizations”. In: *ESEC/FSE ’21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. 2021, pp. 57–69. DOI: 10.1145/3468264.3468575.

[234] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. “Change Is the Only Constant: Dynamic Updates for Workflows”. In: *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. 2022, pp. 350–362. DOI: 10.1145/3510003.3510065.

[235] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. “Decentralizing Infrastructure as Code”. In: *IEEE Software* 40.1 (2023), pp. 50–55. DOI: 10.1109/MS.2022.3192968.

[236] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. *Dependencies in DevOps Survey 2021*. Version 2.0 (Until April 15, 2021). Apr. 2022. DOI: 10.5281/zenodo.6372120.

[237] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. *Evaluation of Safe Dynamic Updating on Collaborative BPMN Workflows with a Discrete-event Simulation: Dataset, Implementation, Measurements, and Analysis*. Version 1.0. Jan. 2022. DOI: 10.5281/zenodo.5864684.

[238] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. *Pulumi TypeScript Projects Using Stack References*. Version 1.0. June 2021. DOI: 10.5281/zenodo.4878577.

[239] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. *Pulumi TypeScript Stack References to µs Converter*. Version v1.0.0. June 2021. DOI: 10.5281/zenodo.4902171.

[240] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. *µs Infrastructure as Code*. 2021. DOI: 10.5281/zenodo.4902312.

[241] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. *µs Performance Evaluation*. Version 1.0. June 2021. DOI: 10.5281/zenodo.4902330.

[242] Fritz Solms and Linda Marshall. “Contract-based Mocking for Services-oriented Development”. In: *Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists, SAICSIT 2016, Johannesburg, South Africa, September 26-28, 2016*. 2016, 40:1–40:8. DOI: 10.1145/2987491.2987534.

[243] Thodoris Sotiropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. “Practical Fault Detection in Puppet Programs”. In: *ICSE ’20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. 2020, pp. 26–37. DOI: 10.1145/3377811.3380384.

[244] Davide Spadini, Mauricio Finavaro Aniche, Magiel Bruntink, and Alberto Bacchelli. “To Mock or Not to Mock?: An Empirical Study on Mocking Practices”. In: *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*. 2017, pp. 402–412. DOI: 10.1109/MSR.2017.61.

[245] David Spielmann, Daniel Sokolowski, and Guido Salvaneschi. “Extensible Testing for Infrastructure as Code”. In: *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH 2023, Cascais, Portugal, October 22-27, 2023*. 2023, pp. 58–60. DOI: 10.1145/3618305.3623607.

[246] Dominic Steinhöfel and Andreas Zeller. “Input Invariants”. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. 2022, pp. 583–594. DOI: 10.1145/3540250.3549139.

[247] Gareth Paul Stoyle, Michael W. Hicks, Gavin M. Bierman, Peter Sewell, and Iulian Neamtiu. “Mutatis Mutandis: Safe and Predictable Dynamic Software Updating”. In: *ACM Trans. Program. Lang. Syst.* 29.4 (2007), p. 22. DOI: 10.1145/1255450.1255455.

[248] Ting Su, Yichen Yan, Jue Wang, Jingling Sun, Yiheng Xiong, Geguang Pu, Ke Wang, and Zhendong Su. “Fully Automated Functional Fuzzing of Android Apps for Detecting Non-crashing Logic Bugs”. In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021), pp. 1–31. DOI: 10.1145/3485533.

[249] Suriya Subramanian, Michael W. Hicks, and Kathryn S. McKinley. “Dynamic Software Updates: A VM-centric Approach”. In: *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. 2009, pp. 1–12. DOI: 10.1145/1542476.1542478.

[250] Sebastian Sztwiertnia, Maximilian Grübel, Amine Chouchane, Daniel Sokolowski, Krishna Narasimhan, and Mira Mezini. “Impact of Programming Languages on Machine Learning Bugs”. In: *AISTA 2021: Proceedings of the 1st ACM International Workshop on AI and Software Testing/Analysis, Virtual Event, Denmark, 12 July 2021*. 2021, pp. 9–12. DOI: 10.1145/3464968.3468408.

[251] Kunal Taneja, Yi Zhang, and Tao Xie. “MODA: Automated Test Generation for Database Applications via Mock Objects”. In: *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*. 2010, pp. 289–292. DOI: 10.1145/1858996.1859053.

[252] Ricardo Terra and Marco Túlio de Oliveira Valente. “A Dependency Constraint Language to Manage Object-oriented Software Architectures”. In: *Softw. Pract. Exp.* 39.12 (2009), pp. 1073–1094. DOI: 10.1002/SPE.931.

[253] Ricardo Terra and Marco Túlio de Oliveira Valente. “Towards a Dependency Constraint Language to Manage Software Architectures”. In: *Software Architecture, Second European Conference, ECSA 2008, Paphos, Cyprus, September 29 - October 1, 2008, Proceedings*. Vol. 5292. 2008, pp. 256–263. DOI: 10.1007/978-3-540-88030-1_19.

[254] Ken Tindell. “Dynamic Code Replacement and Ada”. In: *Ada Lett.* X.7 (1990), pp. 47–54. ISSN: 1094-3641. DOI: 10.1145/101120.101133.

[255] Foivos Tsimpourlas, Ajitha Rajan, and Miltiadis Allamanis. “Supervised Learning over Test Executions as a Test Oracle”. In: *SAC ’21: The 36th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, Republic of Korea, March 22-26, 2021*. 2021, pp. 1521–1531. DOI: 10.1145/3412841.3442027.

[256] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D’Hondt. “Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates”. In: *IEEE Trans. Software Eng.* 33.12 (2007), pp. 856–868. DOI: 10.1109/TSE.2007.70733.

[257] Vasudev Vikram, Rohan Padhye, and Koushik Sen. “Growing a Test Corpus with Bonsai Fuzzing”. In: *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. 2021, pp. 723–735. DOI: 10.1109/ICSE43902.2021.00072.

[258] Simon Friis Vindum and Emil Holm Gjørup. *Hareactive*. https://github.com/funkia/hareactive. Accessed: 2023-11-29. 2019.

[259] Denis Weerasiri, Moshe Chai Barukh, Boualem Benatallah, Quan Z. Sheng, and Rajiv Ranjan. “A Taxonomy and Survey of Cloud Resource Orchestration Techniques”. In: *ACM Comput. Surv.* 50.2 (2017), 26:1–26:41. DOI: 10.1145/3054177.

[260] Johannes Wettinger, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. “Streamlining DevOps Automation for Cloud Applications Using TOSCA as Standardized Metamodel”. In: *Future Gener. Comput. Syst.* 56 (2016), pp. 317–332. DOI: 10.1016/J.FUTURE.2015.07.017.

[261] Johannes Wettinger, Uwe Breitenbücher, and Frank Leymann. “Standards-based DevOps Automation and Integration Using TOSCA”. In: *Proceedings of the 7th IEEE/ACM International Conference on Utility and Cloud Computing, UCC 2014, London, United Kingdom, December 8-11, 2014*. 2014, pp. 59–68. DOI: 10.1109/UCC.2014.14.

[262] 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”. In: *SICS Softw.-Intensive Cyber Phys. Syst.* 35.1-2 (2020), pp. 63–75. DOI: 10.1007/S00450-019-00412-X.

[263] Michael Wurster, Uwe Breitenbücher, Lukas Harzenetter, Frank Leymann, and Jacopo Soldani. “TOSCA Lightning: An Integrated Toolchain for Transforming TOSCA Light into Production-ready Deployment Technologies”. In: *Advanced Information Systems Engineering - CAiSE Forum 2020, Grenoble, France, June 8-12, 2020, Proceedings*. Vol. 386. 2020, pp. 138–146. DOI: 10.1007/978-3-030-58135-0_12.

[264] Michael Wurster, Uwe Breitenbücher, Lukas Harzenetter, Frank Leymann, Jacopo Soldani, and Vladimir Yussupov. “TOSCA Light: Bridging the Gap Between the TOSCA Specification and Production-ready Deployment Technologies”. In: *Proceedings of the 10th International Conference on Cloud Computing and Services Science, CLOSER 2020, Prague, Czech Republic, May 7-9, 2020*. 2020, pp. 216–226. DOI: 10.5220/0009794302160226.

[265] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. “Dynamic Code Evolution for Java”. In: *Proceedings of the 8th International Conference on Principles and Practice of Programming in Java, PPPJ 2010, Vienna, Austria, September 15-17, 2010*. 2010, pp. 10–19. DOI: 10.1145/1852761.1852764.

[266] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *Fuzzing: Breaking Things with Random Inputs*. https://www.fuzzingbook.org/html/Fuzzer.html. Accessed: 2023-11-30. 2023.

[267] Hengcheng Zhu, Lili Wei, Valerio Terragni, Yepang Liu, Shing-Chi Cheung, Jiarong Wu, Qin Sheng, Bing Zhang, and Lihong Song. “StubCoder: Automated Generation and Repair of Stub Code for Mock Objects”. In: *ACM Trans. Softw. Eng. Methodol.* (2023). ISSN: 1049-331X. DOI: 10.1145/3617171.

# Appendix A
# Questionnaire of the Dependencies in DevOps Survey 2021

SI A.1 In this survey, we say *application* for all kinds of operated software products, e.g., software applications, services, middleware or databases.

SQ A.1.1 For the primary application you work on, how often does your organization deploy code to production or release it to end users? *Single choice:* ● Fewer than once per six months ● Between once per month and once every six months ● Between once per week and once per month ● Between once per day and once per week ● Between once per hour and once per day ● On demand (multiple deploys per day)

SQ A.1.2 For the primary application you work on, what is your lead time for changes (i.e., how long does it take to go from code committed to code successfully running in production)? *Single choice:* ● More than six months ● Between one month and six months ● Between one week and one month ● Between one day and one week ● Less than one day ● Less than one hour

SQ A.1.3 For the primary application you work on, how long does it generally take to restore service when a service incident or a defect that impacts users occurs (e.g., unplanned outage or service impairment)? *Single choice:* ● More than six months ● Between one month and six months ● Between one week and one month ● Between one day and one week ● Less than one day ● Less than one hour

SQ A.1.4 For the primary application you work on, what percentage of changes to production or released to users result in degraded service (e.g., lead to service impairment or service outage) and subsequently require remediation (e.g., require a hotfix, rollback, fix forward, patch)? *Single choice:* ● 76% – 100% ● 61% – 75% ● 46% – 60% ● 31% – 45% ● 16% – 30% ● 0% – 15%

SQ A.1 *Hidden virtual question combining results of SQ A.1.1 to SQ A.1.4*

SI A.2 In this survey, we say *application* for all kinds of operated software products, e.g., software applications, services, middleware or databases.

SQ A.2.1 How many other applications need to be deployed so that the primary application you work on provides all its functions? *Single choice:* • 11+ • 6 – 10 • 2 – 5 • 1 • 0

SQ A.2.2 Suppose that the primary application A you work on, to provide all its functions, uses another application B. At your current organization, must application B be deployed before application A? *Single choice:* • Definitely • Probably • Possibly • Probably Not • Definitely Not

SQ A.2.3 Suppose that the primary application A you work on, to provide all its functions, uses another application B. At your current organization, when application B is stopped, must application A be stopped first? *Single choice:* • Definitely • Probably • Possibly • Probably Not • Definitely Not

SQ A.2.4 Suppose that the primary application A you work on, to provide all its functions, uses another application B. At your current organization, if application B is operated by another group of persons and must be deployed before application A, do you use manual or automated coordination to ensure the correct order of the deployments? *Single choice:* • manual coordination (e.g., via chat, phone, or email) • automated coordination (e.g., with a CI/CD pipeline that does *not* require any form of manual coordination) • combination of both (e.g., automated with CI/CD, but coordination via phone, chat, or email is still required)

SQ A.2.5 (Optional) More specifically, at your current organization, which tools or methods would you use to coordinate the order of the deployments? *Free text*

SI A.3 For *all questions on this page*, imagine a fictive company *ACorp*, which has multiple applications, each operated by a distinct group of persons, i.e., a dedicated team. In order to provide all their functions, some of these applications require that other applications are deployed before them. To ensure that the deployments are performed in the correct order, the persons operating the applications at *ACorp* must coordinate the deployments.

For automated coordination at *ACorp*, suppose that the fictive deployment solution *DPL* automatically ensures that deployments are performed in the correct order. Thus, the persons operating an application A can deploy it independently; without manually coordinating with persons who operate the other applications, application A depends on.

In *all questions on this page*, we compare *manual/automated coordination* at *ACorp* with an environment where all applications can be deployed in any order and *no coordination* is required.

| all applications are independent | ***ACorp***<br>some applications require other applications | |
|---|---|---|
| **no coordination** | **manual coordination** | **automated coordination** |
| no deployment order needs to be ensured | deployment order ensured via phone, chat, or email | deployment order ensured through ***DPL*** |

SQ A.3.1 In contrast to *no coordination* (no deployment order needs to be ensured), how often do you think the teams at *ACorp* will deploy code to production or release it to end users if they use…

SQ A.3.1.1 …*manual coordination*, e.g., via phone, chat, or email? *Single choice:* ● Much More Often ● More Often ● Similarly Often ● Less Often ● Much Less Often

SQ A.3.1.2 …*automated coordination* with *DPL*? *Single choice:* ● Much More Often ● More Often ● Similarly Often ● Less Often ● Much Less Often

SQ A.3.1D *Hidden virtual question: difference of SQ A.3.1.1 and SQ A.3.1.2*

SQ A.3.2 In contrast to In contrast to *no coordination* (no deployment order needs to be ensured), how long do you think will the lead time for changes be at *ACorp* (i.e., how long will it take to go from code committed to code successfully running in production) if they use…

SQ A.3.2.1 …*manual coordination*, e.g., via phone, chat, or email? *Single choice:* ● Much Longer ● Longer ● Similarly Long ● Shorter ● Much Shorter

SQ A.3.2.2 …*automated coordination* with *DPL*? *Single choice:* • Much Longer • Longer • Similarly Long • Shorter • Much Shorter

SQ A.3.2D *Hidden virtual question: difference of SQ A.3.2.1 and SQ A.3.2.2*

SQ A.3.3 In contrast to *no coordination* (no deployment order needs to be ensured), how long do you think will it generally take at *ACorp* to restore service when a service incident or a defect that impacts users occurs (e.g., unplanned outage or service impairment) if they use…

SQ A.3.3.1 …*manual coordination*, e.g., via phone, chat, or email? *Single choice:* • Much Longer • Longer • Similarly Long • Shorter • Much Shorter

SQ A.3.3.2 …*automated coordination* with *DPL*? *Single choice:* • Much Longer • Longer • Similarly Long • Shorter • Much Shorter

SQ A.3.3D *Hidden virtual question: difference of SQ A.3.3.1 and SQ A.3.3.2*

SQ A.3.4 In contrast to *no coordination* (no deployment order needs to be ensured), how often do you think changes to production will result in degraded service at *ACorp* (e.g., lead to service impairment or service outage) and subsequently require remediation (e.g., require a hotfix, rollback, fix forward, patch) if they use…

SQ A.3.4.1 …*manual coordination*, e.g., via phone, chat, or email? *Single choice:* • Much More Often • More Often • Similarly Often • Less Often • Much Less Often

SQ A.3.4.2 …*automated coordination* with *DPL*? *Single choice:* • Much More Often • More Often • Similarly Often • Less Often • Much Less Often

SQ A.3.4D *Hidden virtual question: difference of SQ A.3.4.1 and SQ A.3.4.2*

SQ A.3D *Hidden virtual question: sum of SQ A.3.1D to SQ A.3.4D*

SQ A.4.1 How many years of professional experience do you have in the field of developing and operating software? *Single choice:* • 0 to 2 years • 3 to 5 years • 6 to 10 years • 11 to 15 years • 16 years or more • I prefer not to answer

SQ A.4.2 At the moment, to which kind of department do you belong? *Single choice:* ● Development or Engineering ● DevOps or SRE ● Manager ● IT operations or infrastructure ● Consultant, Coach or Trainer ● C-level Executive ● Product Manager ● Professional Services ● SQuality Engineering and Assurance ● Information Security ● Release Engineering ● Other ● I prefer not to answer

SQ A.4.3 How many employees does your company have? *Single choice:* ● 1 – 4 ● 5 – 9 ● 10 – 19 ● 20 – 99 ● 100 – 499 ● 500 – 1,999 ● 2,000 – 4,999 ● 5,000 – 9,999 ● 10,000+ ● I prefer not to answer

SQ A.4.4 In which region are you located? *Single choice:* ● Asia ● Africa ● Central America/Caribbean ● Europe ● North America ● Oceania ● South America ● I prefer not to answer

SQ A.4.5 To which industry does your company belong? *Single choice:* ● Technology ● Financial Services ● Retail/Consumer/e-Commerce ● Healthcare & Pharmaceuticals ● Government ● Media/Entertainment ● Insurance ● Education ● Industrials & Manifacturing ● Telecommunications ● Energy ● Non-profit ● I prefer not to answer

SQ A.4.6 (Optional) Is there anything you want to share with us? *Free text*

# Appendix B
# Coding Tables of the Dependencies in DevOps Survey 2021

| Code | SQ A.1 | SQ A.1.4 | SQ A.4.1 | SQ A.4.3 |
|---|---|---|---|---|
| 1 | low | 76 %–100 % | 0 to 2 years | 1–4 |
| 2 | medium | 61 %–75 % | 3 to 5 years | 5–9 |
| 3 | high | 46 %–60 % | 6 to 10 years | 10–19 |
| 4 | elite | 31 %-–45 % | 11 to 15 years | 20–99 |
| 5 | | 16 %–30 % | 16 years or more | 100–499 |
| 6 | | 0 %–15 % | | 500–1 999 |
| 7 | | | | 2 000–4 999 |
| 8 | | | | 5 000–9 999 |
| 9 | | | | 10 000+ |
| | | | I prefer not to answer | I prefer not to answer |

| Code | SQ A.1.1 | SQ A.1.2 and SQ A.1.3 |
|---|---|---|
| 1 | Fewer than once per six months | More than six months |
| 2 | Between once per month and once every six months | Between one month and six months |
| 3 | Between once per week and once per month | Between one week and one month |
| 4 | Between once per day and once per week | Between one day and one week |
| 5 | Between once per hour and once per day | Less than one day |
| 6 | On demand (multiple deploys per day) | Less than one hour |

| Code | SQ A.2.1 | SQ A.2.2 and SQ A.2.3 | SQ A.2.4 |
|---|---|---|---|
| 1 | 0 | Definitely Not | manual coordination |
| 2 | 1 | Probably Not | combination of both |
| 3 | 2–5 | Possibly | automated coordination |
| 4 | 6–10 | Probably | |
| 5 | 11+ | Definitely | |

| Code | SQ A.3.1 | SQ A.3.2 and SQ A.3.3 | SQ A.3.4 |
|---|---|---|---|
| 1 | Much Less Often | Much Longer | Much More Often |
| 2 | Less Often | Longer | More Often |
| 3 | Similarly Often | Similarly Long | Similarly Often |
| 4 | More Often | Shorter | Less Often |
| 5 | Much More Often | Much Shorter | Much Less Often |

# Curriculum Vitæ

| | |
|---|---|
| Name | Daniel Sokolowski |
| ORCID | 0000-0003-2911-8304 |
| Website | https://dsoko.de |
| Citizenship | German |

## Education

Nov. 2021 – June 2024 Ph.D. in Computer Science *University of St. Gallen, Switzerland*
Continuation of Ph.D. studies. Thesis title: Reliable Infrastructure as Code for Decentralized Organizations. Advisor: Prof. Dr. Guido Salvaneschi.

May 2019 – Oct. 2021 Ph.D. Student in Computer Science *TU Darmstadt, Germany*
Beginning of Ph.D. studies. Advisor: Prof. Dr. Guido Salvaneschi.

Oct. 2016 – April 2019 M.Sc. in Internet- and Web-based Systems *TU Darmstadt, Germany*
Thesis title: Performance in Privacy-Aware Distributed Stream Processing. Advisor: Prof. Dr. Guido Salvaneschi.

Jan. 2018 – July 2018 Visiting Student in Computer Science and Engineering *IIT Delhi, India*
Studied an equivalent of 28 ECTS credits.

Oct. 2013 – Sept. 2016 B.Eng. in Information Technology *DHBW Mannheim, Germany*
Cooperative study program (Duales Studium) with Lufthansa Systems and Lufthansa Industry Solutions. Advisor: Prof. Dr. Harald Kornmayer.

June 2013 Abitur (A-Levels) *Internatsschule Schloss Hansenberg, Germany*

## Work Experience

Nov. 2021 – June 2024 Research Assistant *University of St. Gallen, Switzerland*
Programming Group led by Prof. Dr. Guido Salvaneschi.

May 2019 – Oct. 2021 Research Assistant *TU Darmstadt, Germany*
Reactive Programming Technology led by Prof. Dr. Guido Salvaneschi.

Oct. 2016 – Dec. 2017 Web Developer at Lufthansa Industry Solutions *Frankfurt a.M., Germany*
Business unit Air Cargo.

| | | |
|---|---|---|
| Feb. 2015 – | Co-Founder & CTO at actified | *Neu-Isenburg, Germany* |
| Dec. 2016 | Social network startup connecting people for sports. | |
| April 2015 – | Trainee at Lufthansa Industry Solutions | *Frankfurt a.M., Germany* |
| Sept. 2016 | Cooperative study program (Duales Studium). 4 three-month internships. | |
| Sept. 2013 – | Trainee at Lufthansa Systems | *Kelsterbach, Germany* |
| Mar. 2015 | Cooperative study program (Duales Studium). 2 three-month internships. | |
| Mar. 2013 | Intern at CERN | *Geneva, Switzerland* |
| | 2 weeks internship in research organization. | |
| Oct. 2011 – | Intern at Lufthansa German Airlines Beijing Office | *Beijing, China* |
| Nov. 2011 | 4 weeks internship in marketing and sales. | |

## Summer Schools

| | | |
|---|---|---|
| Aug. 2023 | Marktoberdorf Summer School | *Marktoberdorf, Germany* |
| | 2 weeks on safety and security through formal methods. | |
| July 2023 | TAROT Summer School | *London, UK* |
| | 1 week on software testing, verification, and validation. | |
| July 2022 | Advanced Functional Programming (AFP) Summer School | *Utrecht, NL* |
| | 1 week on advanced functional programming in Haskell. | |
| June 2019 | Oregon Programming Languages Summer School (OPLSS) | *Eugene, USA* |
| | 2 weeks on programming languages theory and security. | |

## Awards

| | | |
|---|---|---|
| May 2022 | Best Artifact Award at ICSE 2022 | *Pittsburgh, USA* |
| May 2021 | Best Student Volunteer Award at ICSE 2021 | *Online* |
| Mar. 2017 | 1st at Switch-Up Challenge powered by Cisco | *Eschborn, Germany* |
| Oct. 2014 | 10th at Gründer-Garage Business Idea Competition | *Berlin, Germany* |
| June 2012 | 1st (state) & 3rd (nation) at Deutscher Gründerpreis für Schüler | *Germany* |

## Service to the Research Community

Since 2020 Active (shadow) reviewer in the ACM and IEEE Software Engineering and Programming Languages research communities.

Oct. 2023 Student Volunteer at SPLASH 2023 *Cascais, Portugal*

July 2021 Program Committee Member at ISC 2021 SuperCompCloud *Online*

May 2021 Student Volunteer at ICSE 2021 *Online*

Oct. 2019 Student Volunteer at SPLASH 2019 *Athens, Greece*

## Student Supervision

Supervised 2 Master's and 6 Bachelor's theses (student names omitted to respect privacy):

MT 2023 Formal Verification of Network Access in Infrastructure as Code Programs. Examiner: Mira Mezini. Supervisors: Daniel Sokolowski, Ragnar Mogk. *TU Darmstadt, Germany*

MT 2023 Safe Dynamic Updates for Workflows on Kubernetes. Examiner: Mira Mezini. Supervisor: Daniel Sokolowski. *TU Darmstadt, Germany*

BT 2023 Challenges in Modern Infrastructure as Code. Examiner: Guido Salvaneschi. Supervisor: Daniel Sokolowski. *University of St. Gallen, Switzerland*

BT 2021 Multitier Reactive Programming for Hybrid Cloud HPC. Examiner: Mira Mezini. Supervisor: Daniel Sokolowski. *TU Darmstadt, Germany*

BT 2020 Dynamic Placement in Serverless Computing. Examiner: Guido Salvaneschi. Supervisor: Daniel Sokolowski. *TU Darmstadt, Germany*

BT 2020 Design and Formal Verification of a Distributed Consensus Algorithm for ScalaLoci. Examiner: Guido Salvaneschi. Supervisor: Daniel Sokolowski. *TU Darmstadt, Germany*

BT 2020 Evaluation of Safe Dynamic Updating of Microservices. Examiner: Guido Salvaneschi. Supervisor: Daniel Sokolowski. *TU Darmstadt, Germany*

BT 2019 Multitier Programming in HPC. Examiner: Guido Salvaneschi. Supervisor: Daniel Sokolowski. *TU Darmstadt, Germany*

## Teaching Experience

Teaching Assistant (exercises and exams) *University of Zurich, Switzerland*
Lecture: Foundations of Programming Languages & Static Analysis.
Spring 2022, Spring 2023: ~20 students. Spring 2024: ~40 students.

Teaching Assistant (exercises, projects, and exams) *University of St. Gallen, Switzerland*
Lecture: Programming Methodology. Spring 2023: ~30 students.

Teaching Assistant (exercises, projects, and exams) *University of St. Gallen, Switzerland*
Lecture: Introduction to Programming. Fall 2022, Fall 2023: ~30 students.

Guest Lecture on “Building Intelligent Systems” *University of St. Gallen, Switzerland*
Lecture: Engineering Software Systems. Spring 2022, Spring 2023: ~20 students.

Teaching Assistant (project supervision) *University of St. Gallen, Switzerland*
Project: Integrative Master’s Project. Fall 2022: ~20 students.

Guest Lecture on “Building Intelligent Systems” *University of St. Gallen, Switzerland*
Lecture: Emerging Trends in Information Technology. Fall 2021, Fall 2022: ~15 students.

Co-Organizer *TU Darmstadt, Germany*
Seminar: Software Engineering for Artificial Intelligence. Summer 2020: ~35 students.
Winter 2020/2021, Winter 2021/2022: ~10 students.

Teaching Assistant (project supervision) *TU Darmstadt, Germany*
Project: Implementation of Programming Languages. Winter 2019/2020, Summer 2020,
Winter 2020/2021, Summer 2021, Winter 2021/2022: ~15 students.

Teaching Assistant (seminar paper supervision) *TU Darmstadt, Germany*
Seminar: Design and Implementation of Programming Languages. Winter 2019/2020,
Summer 2020, Winter 2020/2021, Summer 2021, Winter 2021/2022: ~15 students.

Teaching Assistant (exercises and exams) *TU Darmstadt, Germany*
Lecture: Software Engineering. Winter 2019/2020: ~450 students.

## Publications

Section 1.4 lists all publications of this dissertation. Google Scholar provides a complete publication list: https://dsoko.de/pubs. The most significant publications so far:

[229] Daniel Sokolowski, David Spielmann, and Guido Salvaneschi. “Automated Infrastructure as Code Program Testing”. In: *IEEE Transactions on Software Engineering* (2024). DOI: 10.1109/TSE.2024.3393070

[234] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. “Change Is the Only Constant: Dynamic Updates for Workflows”. In: *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. 2022, pp. 350–362. DOI: 10.1145/3510003.3510065

[233] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. “Automating Serverless Deployments for DevOps Organizations”. In: *ESEC/FSE ’21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. 2021, pp. 57–69. DOI: 10.1145/3468264.3468575

In our field, conferences are usually considered more prestigious than journals. The most prestigious venues include ICSE, FSE, ASE, TSE, TOSEM (Software Engineering), and PLDI, POPL, OOPSLA, ICFP, ECOOP, TOPLAS (Programming Languages).