

Infrastructure as Code for Dynamic Deployments

Daniel Sokolowski
daniel.sokolowski@unisg.ch
University of St. Gallen
Switzerland

ABSTRACT

Modern DevOps organizations require a high degree of automation to achieve software stability at frequent changes. Further, there is a need for flexible, timely reconfiguration of the infrastructure, e.g., to use pay-per-use infrastructure efficiently based on application load. Infrastructure as Code (IaC) is the DevOps tool to automate infrastructure. However, modern *static IaC* solutions only support infrastructures that are deployed and do not change afterward. To implement infrastructures that change *dynamically* over time, static IaC programs have to be (updated and) re-run, e.g., in a CI/CD pipeline, or configure an external orchestrator that implements the dynamic behavior, e.g., an autoscaler or Kubernetes operator. Both do not capture the dynamic behavior in the IaC program and prevent analyzing and testing the infrastructure configuration jointly with its dynamic behavior.

To fill this gap, we envision *dynamic IaC*, which augments static IaC with the ability to define dynamic behavior within the IaC program. In contrast to static IaC programs, dynamic IaC programs run continuously. They re-evaluate program parts that depend on external signals when these change and automatically adjust the infrastructure accordingly. We implement D_{IaC} as the first dynamic IaC solution and demonstrate it in two realistic use cases of broader relevance. With dynamic IaC, ensuring the program's correctness is even harder than for static IaC because programs may define many target configurations in contrast to only a few. However, for this reason, it is also more critical. To solve this issue, we propose automated, specialized property-based testing for IaC programs and implement it in ProTI.

CCS CONCEPTS

• **Software and its engineering** → **Orchestration languages**; *Cloud computing*; Architecture description languages.

KEYWORDS

DevOps, Infrastructure as Code, Cloud, Software Evolution, Testing

ACM Reference Format:

Daniel Sokolowski. 2022. 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 '22)*, November 14–18, 2022, Singapore, Singapore. ACM, New York, NY, USA, 5 pages. <https://doi.org/10.1145/3540250.3558912>



This work is licensed under a Creative Commons Attribution 4.0 International License.

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

© 2022 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-9413-0/22/11.

<https://doi.org/10.1145/3540250.3558912>

1 INTRODUCTION

DevOps organizations unite development and operations in cross-functional teams to improve their *Software Delivery and Operational* (SDO) performance [12]. They aim at producing stable software (Dev) with reliable operations (Ops), requiring automation along the whole software delivery pipeline. In these efforts, the tool to automate infrastructure management is *Infrastructure as Code* (IaC) [22].

Early IaC solutions, e.g., Ansible [4] and Chef [7] used imperative scripts. Later, in *declarative* solutions, e.g., Puppet [28], developers only describe the desired infrastructure state, and the tool automatically derives the required actions to achieve it. Declarative IaC is often preferred because it promises better adaptability and robustness. More recent IaC solutions focus on declarative provisioning of virtualized cloud infrastructure. In AWS CloudFormation [3], ARM [21], and Terraform [15], developers describe the infrastructure in JSON, YAML, or similar tool-specific DSLs, e.g., HCL and Bicep. The AWS Cloud Development Kit (AWS CDK) [34] and CDK for Terraform (CDKTF) [14] allow to generate AWS CloudFormation and Terraform programs from IaC programs written in a general-purpose programming language. Thus, they are limited to the abilities of the underlying tools' JSON, YAML, or HCL DSLs. In contrast, Pulumi [26] performs the operations itself, enabling intertwined IaC program execution and deployment operations. This approach allows, e.g., arbitrary computations based on a resource's post-deployment configuration to configure other resources, providing a more integrated user experience. In this project, we focus on the most recent generation of IaC solutions, leveraging (imperative) general-purpose programming languages like Go, Python, and TypeScript to declaratively define the desired infrastructure state.

The core abstraction in all common declarative IaC solutions is the typed, directed, acyclic resource graph [44] (examples in Figure 1). Nodes are resources, i.e., deployable units like servers, files, or policies. Arcs are dependencies, typically describing *depends-on* or *contained-in* relationships. The arcs constrain the order in which resources have to be (un)deployed, i.e., a resource may only be deployed after all resources it depends on are available and must be undeployed before any of them. E.g., a server must exist before a file on it, and the file must be deleted before the server.

Today's IaC solutions implement *static IaC*, where the resource graph is static, i.e., the IaC program runs once, terminates, and the infrastructure remains unchanged until the next execution. To achieve dynamic infrastructure behavior, developers have to fall back on external tools like CI/CD pipelines or dynamic infrastructure resources like Kubernetes operators. They either re-run the IaC program with changed configuration or statically configure a dynamic orchestrator. This prevents central, holistic analysis and testing of the infrastructure configuration and its dynamic changes.

To solve this issue, we envision *dynamic IaC*, allowing to describe dynamic infrastructure behavior in the IaC program, i.e.,

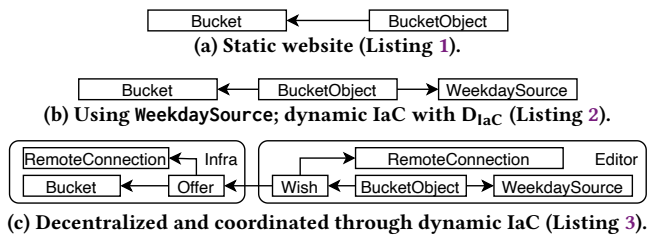


Figure 1: Resource graphs of the website IaC programs.

expressing dynamically changing resource graphs. For instance, to efficiently use pay-per-use infrastructure, developers could define that the infrastructure is different based on the load (i.e., an external signal), implementing automatic reconfiguration in the dynamic IaC program. In our previous work on automating the coordination of deployments, a topic that is motivated by our survey with 134 IT professionals [38], we already proposed an early, specialized dynamic IaC solution [36]. In our work on how to update components in modern distributed systems such that no distributed transaction breaks [37], we discussed another inherently dynamic infrastructure topic. Building upon these works, we develop the general dynamic IaC solution D_{IaC} (Section 4). We discuss D_{IaC} in both use cases that we studied in depth (Section 5) and evaluate its performance and benefits compared to current solutions (Section 7).

Dynamic IaC amplifies the need for easy and thorough testing because developers have to ensure the correctness of all infrastructures their dynamic IaC program configures—in contrast to static IaC programs, targeting only a single or few infrastructure configurations. Current testing of modern IaC programs is limited to standard unit testing (before deployment), property testing (during deployment), and integration test (after deployment) [27]. Property and integration testing is potentially costly and slow, and unit testing is typically example driven and effortful.

To close this gap, we propose ProTI (Section 6), a specialized, automated tool for property-based testing [11] (a technique pioneered by QuickCheck [8]) of static and dynamic IaC programs. ProTI automatically runs IaC programs in many different configurations before deployment, allowing out-of-the-box quick but thorough termination testing. Further, ProTI is an easy way to check application-specific properties in IaC programs. We plan to evaluate ProTI on all public Pulumi TypeScript programs on GitHub, and all IaC programs used in this project (Section 7).

2 RELATED WORK

Infrastructure as Code. Rahman et al. [29] provide a systematic overview of IaC research. Guerriero et al. [13] discovered through interviews that practitioners need better support and tooling for IaC program maintenance and evolution. Various works studied problems, code smells, evolution, and proposed linters for Ansible, Chef, and Puppet [24, 25, 30, 31, 33, 35]. Previous IaC research is centered around IaC solutions that focus on configuring resources without their provisioning, meanwhile often referred to as Configuration as Code (CaC). This work focuses on declarative IaC for virtualized cloud infrastructure, including provisioning.

Resource Orchestration. Weerasiri et al. [43] give an overview of cloud resource orchestration techniques, Ranjan et al. [32] on the

Listing 1: Static website showing the weekday in Pulumi.¹

```

1.1 const bucket = new aws.s3.Bucket("website", {
1.2   website: { indexDocument: "index.html" } });
1.3 const today = getWeekday(new Date());
1.4 new aws.s3.BucketObject("index", {
1.5   bucket: bucket, key: "index.html",
1.6   contentType: "text/html; charset=utf-8",
1.7   content: `<!DOCTYPE html>${today}` });

```

programming of resource orchestration operations, and COPE [20] is a distributed policy enforcement engine for resource orchestrators. Plenty of industrial-grade cloud resource orchestrators exist, e.g., Kubernetes and Mesos. Resource orchestration is about achieving infrastructure setup and changes. In contrast, IaC is about how developers configure the infrastructure. IaC solution runtimes are or use resource orchestrators when running IaC programs.

Modeling Languages. Wurster et al. [44] proposed EDMM as a least denominator metamodel for IaC solutions and mapped it to a subset of TOSCA [45]. TOSCA [23] is an OASIS standard for cloud modeling. Bellendorf and Mann [6] provide an overview of related research and tools. TOSCA separates static, declarative topology (i.e., resource graph) descriptions from imperatively described dynamic infrastructure behavior, e.g., in BPMN workflows. In contrast, dynamic IaC with D_{IaC} allows defining the dynamic behavior declaratively united with the resource graph description.

Architecture Description Languages (ADLs). ADLs on various levels have been proposed, e.g., ArchJava [2] on the software component level and ORS [19] on the service level. Other work enables constraining such definitions [39]. ADLs allow describing and verifying architecture. However, unlike IaC, they are not executable specifications and typically do not cover dynamic behavior.

3 STATIC INFRASTRUCTURE AS CODE

To illustrate the difference between static IaC and dynamic IaC, we discuss a website deployment: A static HTML page that shows the weekday hosted in an AWS S3 bucket. Listing 1 is the website's IaC program in Pulumi TypeScript, constructing the resource graph in Subfigure 1a. Lines 1.1 and 1.2 define the S3 bucket hosting the website. Line 1.3 assigns the current weekday as string to the constant `today`, which is used in the content of the `index.html` page that is an object within the bucket (Lines 1.4 to 1.7).

After executing this IaC program, the website displays the correct weekday. However, the webpage will be wrong the next day because Pulumi is static IaC, i.e., the IaC program executes once, terminates, and the infrastructure does not change afterward. With static IaC, users have two options to ensure that the website displays the correct weekday every day. First, implementing the dynamic behavior externally, e.g., a CI/CD pipeline re-runs the IaC program daily. This separation of static and dynamic concerns prevents a holistic view and possibility that is beneficial to analyze and test jointly (cf. Sections 4 and 6). Second, implementing the dynamic behavior in infrastructure that is configured statically. E.g., AWS could offer a bucket object resource that replaces the string `$$WEEKDAY$$` in the content with the current weekday when the object is accessed. Such

¹For brevity, imports, export, and the bucket policy to allow public access are omitted.

a resource would enable the static configuration of the website, e.g., removing Line 1.3 and replacing the content value in Line 1.7 with "`<!DOCTYPE html>$$WEEKDAY$$`". However, this approach is limited to the dynamic behavior the infrastructure provider supports.

4 DYNAMIC INFRASTRUCTURE AS CODE

Listing 2 is an alternative IaC program for the weekday website. In contrast to Listing 1, Line 2.2 defines a `WeekdaySource` resource, providing the current weekday in its `today` property. Given an implementation of `WeekdaySource`, Listing 2 is a valid Pulumi program and configures the static infrastructure in Subfigure 1b. The `apply` method on resource outputs provides access to their values, which are only available after the resource is deployed. E.g., `apply` in Line 2.3 provides the weekday value in the inline function (Lines 2.3 to 2.6). Listing 2 does not generate the dynamically changing value but retrieves it from a resource. That resource is long-living and knows when the weekday changes. Unfortunately, with Pulumi, we can only retrieve the value once, and then the IaC program terminates, preventing reactively updating `index.html`.

To solve this issue, we propose D_{IaC} , an extension of Pulumi TypeScript for dynamic IaC. In D_{IaC} , resource outputs are not limited to values that are resolved once. Instead, they are streams, i.e., values that can change over time. IaC programs in D_{IaC} are continuously running—in contrast to static IaC programs in Pulumi that run once and terminate. On the code level, D_{IaC} evaluates and deploys the whole program once, like Pulumi. However, unlike Pulumi, D_{IaC} continues to watch for updates on resource outputs. Whenever a resource output changes, D_{IaC} re-evaluates the parts of the IaC program that depend on the updated output, resulting in an updated resource graph, D_{IaC} applies to the infrastructure.

This design yields that Listing 2 is already a valid dynamic IaC program for D_{IaC} . Yet, `weekdays.today` is now a stream of weekday strings, not a future-like string that is only resolved once. D_{IaC} executes all code registered to the stream—here, the inline function Lines 2.3 to 2.6—whenever a new value is available, i.e., directly after the IaC program's start and every midnight.

At first sight, the solution for dynamic IaC is similar to the opposed workarounds for static IaC (cf. Section 3). Yet, as shown in the example, the dynamic behavior is not externalized, e.g., to the `WeekdaySource`. The external resource is only used as a signal to (a) trigger partial re-evaluation and (b) provide new data, i.e., the current weekday. The dynamic behavior—re-configuring *another* resource based on the changed weekday—is encoded in the IaC program, making it a central source for holistic reasoning about the infrastructure, including its dynamic behavior.

5 EXAMPLE USE CASES FOR DYNAMIC IAC

We now present two practical use cases of broader relevance.

Automated Decentralized Deployment Coordination. In modern DevOps organizations, cross-functional teams aim to operate their application(s) as independently as possible. Still, in practice, applications depend on one another, and these dependencies carry on to the applications' (un)deployment. For example, consider that the "infra" team is responsible for the bucket of the weekday website and the "editor" team for `index.html`. They have to coordinate to

Listing 2: Weekday website using `WeekdaySource` resource.¹

```
2.1 const bucket = new aws.s3.Bucket("website", { /* ... */ });
2.2 const weekdays = new WeekdaySource("weekdays")
2.3 const weekdays.today.apply((today) => {
2.4     new aws.s3.BucketObject("index", { /* ... */
2.5         content: `<!DOCTYPE html>${today}` });
2.6 });
```

Listing 3: Weekday website decentralized across two independent teams with automated (un)deployment coordination.¹

(a) The infra team's dynamic IaC program.

```
3a.1 const editor = new RemoteConnection("editor", { /* ... */ });
3a.2 const bucket = new aws.s3.Bucket("website", { /* ... */ });
3a.3 new Offer(editor, "bucket", bucket);
```

(b) The editor team's dynamic IaC program.

```
3b.1 const infra = new RemoteConnection("infra", { /* ... */ });
3b.2 const wish = new Wish<aws.s3.Bucket>(infra, "bucket");
3b.3 const wdays = new WeekdaySource("weekdays");
3b.4 join(wish.offer, wdays.today).apply((bucket, today) => {
3b.5     new aws.s3.BucketObject("index", { /* ... */
3b.6         bucket: infraBucket,
3b.7         content: `<!DOCTYPE html>${today}` });
3b.8 });
```

ensure that (a) infra deploys the bucket before editor deploys `index.html`, (b) editor undeploys `index.html` before infra undeploys the bucket, and (c) editor updates `index.html` when infra changes the bucket. This coordination can be automated with dynamic IaC in D_{IaC} . For instance, Listing 3 and Subfigure 1c show both teams' dynamic IaC programs and the resource graphs. They explicitly define their connection to the other deployment (Lines 3a.1 and 3b.1). The infra team *offers* in Line 3a.3 the bucket to the editor team, which explicitly *wishes* for it (Line 3b.2). The offer is deployed when the wish signals its use and undeployed once the wish confirmed that it is not used anymore. This achieves that `index.html` (Lines 3b.5 to 3b.7) is only deployed when the offer is available, always updated with new weekday values, and undeployed when `wish.offer` indicates the infra team signals it withdraws its offer.

Safe Dynamic Software Updating (Safe DSU). Updating software is important, e.g., to fix vulnerabilities or introduce features. However, updating a component in a distributed system may break running distributed transactions. If these transactions are frequent or take long, perhaps even days, breaking and repeating them is infeasible. Safe DSU is about identifying *when* a component can be updated such that no transaction breaks, i.e., they do not have to be repeated and interruption is minimal. The safety criterion is *version consistency*: a component participates in, at most, one version in each transaction. Safe DSU requires for each component update that sequentially (1) a safe update interval is enforced, then (2) the update is performed, and, after completion, (3) the safe update interval is released. In static IaC programs, this protocol cannot be implemented. Dynamic IaC with D_{IaC} can implement this protocol by (re-)configuring the component's inbound proxies and the component itself based on the component version and the transaction monitoring insights of the component's inbound proxies.

6 TESTING DYNAMIC IAC PROGRAMS

Ensuring IaC programs work correctly is important and dynamic IaC amplifies the need for easy and thorough testing (cf. Section 1). To solve this issue, we propose ProTI, a specialized, automated tool for property-based testing of static and dynamic IaC programs.

ProTI randomly executes the IaC program many times with different values to find errors systematically. Every resource is replaced with an auto-generated mock, and the mocks validate each occurring resource input value. Resource outputs are so-called *arbitrarities*, random generators for concrete values of the outputs' specified types. To replace other non-resource objects with mocks, ProTI provides an annotation. This suffices to randomly execute the IaC program thousands of times with different values, providing out-of-the-box high confidence that the IaC program always terminates. Additionally, users can add annotations providing *application-specific* specifications that ProTI validates and uses for more precise value generation. A first version of ProTI targets static Pulumi TypeScript and dynamic D_{IaC} programs, leveraging TypeScript decorators and fast-check [10] for property-based testing.

For instance, testing the editor team's IaC program (Sublisting 3b) with ProTI runs the program many times. ProTI checks that resource input values comply with their specified type, e.g., only string values are assigned to content in Line 3b.7. Arbitrary values are generated for all resource outputs, respecting their type. E.g., `wish.offer` generates different buckets and `undefined`, signaling the offer is unavailable, and `wdays.today` generates random string values. Further, we can add application-specific specifications, e.g., by annotating content in Line 3b.7 with `@inNarrow([/* 7 weekdays */].map((d) => `<!DOCTYPE html>${d}`))`, narrowing down its accepted values from any string to seven concrete strings. Now the ProTI tests fail because `wdays.today` may provide any string value. To fix the tests, `WeekdaySource` in Line 3b.3 is annotated with `@outNarrow({ today: [/* 7 weekdays */] })`, narrowing down the values of its today output.

7 EVALUATION

To evaluate D_{IaC} , we first synthesize typical static IaC programs and compare their deployment with D_{IaC} , Pulumi, and AWS CDK, showing that D_{IaC} does not introduce significant overhead. To evaluate D_{IaC} 's dynamic IaC capabilities, we thoroughly demonstrate its applicability in the two presented use cases of broader relevance:

To motivate automated coordination of decentralized deployments, we organize a cross-sectional online survey with IT professionals from industry, complying with common standards, including the ACM SIGSOFT guidelines [1, 16, 17]. The survey assesses the state of application dependencies in practice, whether they constrain deployment orders, and how these are coordinated. We then demonstrate in depth on the TeaStore microservices application [42] how dynamic IaC can be leveraged for such deployment coordination. Further, we assess the scaling behavior of the coordination through dynamic IaC using synthetic decentralized deployment benchmarks. To confirm general applicability, we automatically convert all public Pulumi TypeScript projects using stack references to D_{IaC} with automated deployment coordination. Stack references are the only mean in Pulumi to depend on another deployment; however, they do suffice for automated coordination.

For safe DSU, we show how state-of-the-art approaches [5, 18, 37, 40] can be applied to modern workflow applications. First, we evaluate this application through discrete-event simulation of updates in all 106 realistic collaborative BPMN workflows from RePROSitory [9]. We then generate dynamic IaC programs in D_{IaC} for these workflows and repeat the experiments using the D_{IaC} dynamic IaC programs, continuous-time simulation for the workflow engines, and Kubernetes for the execution of the workflow tasks.

To demonstrate the benefits of dynamic IaC's holistic view, we select projects using static IaC with Pulumi and CI/CD on GitHub. On a few representative projects with dynamic infrastructure behavior in the CI/CD pipeline, we demonstrate and discuss the replacement with a D_{IaC} IaC program. We expect to achieve simplified CI/CD pipelines, the ability to jointly reason and test the infrastructure and its behavior, and reduced infrastructure update latency.

To evaluate ProTI, we apply it to all publicly available Pulumi TypeScript programs on GitHub, showing that it is easy to apply and that we effectively find termination issues. Additionally, we apply it to the dynamic IaC programs used in our evaluation for D_{IaC} . In contrast to out-of-the-box termination testing, the deeper insight into these case studies allows us to evaluate the applicability of D_{IaC} for checking application-specific properties.

8 CONTRIBUTIONS AND ACHIEVED RESULTS

By now, research on IaC in the SE community either discussed CaC or modeling. We **extend the discussion to modern IaC solutions**, especially solutions leveraging general-purpose programming languages, and contribute novel ideas, i.e., for dynamic IaC, testing, deployment coordination, and safe updating.

We **organized the Dependencies in DevOps Survey** [38] on 134 IT professionals. The survey shows that (1) applications often depend on one another, (2) dependencies often constrain the order of (un)deployment operations, and (3) coordination is typically manual, even though (4) automation promises better performance.

We **implemented and evaluated μS** ([mjuz] "muse") [36], a dynamic IaC extension of Pulumi [26] using Hareactive [41]. μS 's performance is better than AWS CDK's and similar to Pulumi's. The scaling behavior is as expected. μS is the basis for D_{IaC} .

We **motivated safe DSU, applied approaches to asynchronous workflows, and suggested Essential Safety** [37]. The evaluation simulates 106 realistic BPMN workflows from RePROSitory [9], confirming the applicability to workflows on distributed systems. It is the basis for the evaluation of D_{IaC} on Kubernetes.

We consolidate our work on dynamic IaC in **implementing and evaluating D_{IaC}** , the first IaC solution featuring dynamic IaC in a general fashion. We **implement and evaluate ProTI**, a novel, automated, systematic testing approach for modern IaC programs based on auto-mocking and property-based testing.

ACKNOWLEDGMENTS

The author's advisor is Prof. Dr. Guido Salvaneschi, an associate professor at the University of St. Gallen. This work has been co-funded by the Swiss National Science Foundation (SNSF, No. 200429), by the German Research Foundation (DFG, No. 383964710, SFB 1119), by the Hessian LOEWE initiative (emergenCITY and Software-Factory 4.0), and by the University of St. Gallen (IPF, No. 1031569).

REFERENCES

- [1] ACM Special Interest Group on Software Engineering. 2021. Empirical Standards: Questionnaire Surveys. <https://github.com/acmsigsoft/EmpiricalStandards/blob/master/docs/QuestionnaireSurveys.md> (Accessed: 2021-05-05).
- [2] Jonathan Aldrich, Craig Chambers, and David Notkin. 2002. ArchJava: Connecting Software Architecture to Implementation. In *Proceedings of the 24th International Conference on Software Engineering (Orlando, Florida) (ICSE '02)*. Association for Computing Machinery, New York, NY, USA, 187–197. <https://doi.org/10.1145/581339.581365>
- [3] Amazon Web Services. 2021. AWS CloudFormation. <https://aws.amazon.com/cloudformation/> (Accessed: 2021-05-06).
- [4] Ansible and Red Hat. 2022. Ansible is Simple IT Automation. <https://www.ansible.com/> (Accessed: 2022-07-12).
- [5] Luciano Baresi, Carlo Ghezzi, Xiaoxing Ma, and Valerio Panzica La Manna. 2017. Efficient Dynamic Updates of Distributed Components Through Version Consistency. *IEEE Transactions on Software Engineering* 43, 4 (2017), 340–358. <https://doi.org/10.1109/TSE.2016.2592913>
- [6] Julian Bellendorf and Zoltán Ádám Mann. 2020. Specification of cloud topologies and orchestration using TOSCA: a survey. *Computing* 102, 8 (2020), 1793–1815. <https://doi.org/10.1007/s00607-019-00750-3>
- [7] Chef. 2022. Chef Software DevOps Automation Tools & Solutions. <https://chef.io> (Accessed: 2022-07-12).
- [8] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 268–279. <https://doi.org/10.1145/351240.351266>
- [9] Flavio Corradini, Fabrizio Fornari, Andrea Polini, Barbara Re, and Francesco Tiezzi. 2019. RePROSitory: a Repository Platform for Sharing Business PROcess models. *BPM (PhD/Demos)* (2019), 149–153.
- [10] Nicolas Dubien. 2022. fast-check: Property based testing framework for JavaScript/TypeScript. <https://github.com/dubzzz/fast-check> (Accessed: 2022-07-12).
- [11] George Fink and Matt Bishop. 1997. Property-Based Testing: A New Approach to Testing for Assurance. *SIGSOFT Softw. Eng. Notes* 22, 4 (jul 1997), 74–80. <https://doi.org/10.1145/263244.263267>
- [12] Nicole Forsgren, Dustin Smith, Jez Humble, and Jessie Frazelle. 2019. 2019 Accelerate State of DevOps Report. <https://services.google.com/fh/files/misc/state-of-devops-2019.pdf> (Accessed: 2022-04-07).
- [13] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba. 2019. Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 580–589. <https://doi.org/10.1109/ICSME.2019.00092>
- [14] HashiCorp. 2022. CDK for Terraform. <https://www.terraform.io/cdktf> (Accessed: 2022-07-12).
- [15] HashiCorp. 2022. Terraform. <https://www.terraform.io/> (Accessed: 2022-07-12).
- [16] Mark Kasunic. 2005. Designing an Effective Survey. <https://doi.org/10.1184/R1/6573062.v1>
- [17] Barbara A. Kitchenham and Shari L. Pfleeger. 2008. Personal Opinion Surveys. In *Guide to Advanced Empirical Software Engineering*. Forrest Shull, Janice Singer, and Dag I. K. Sjøberg (Eds.). Springer, Chapter 3, 63–92.
- [18] Jeff Kramer and Jeff Magee. 1990. The evolving philosophers problem: dynamic change management. *IEEE Transactions on Software Engineering* 16, 11 (1990), 1293–1306. <https://doi.org/10.1109/32.60317>
- [19] Ingolf Krüger, Barry Demchak, and Massimiliano Menarini. 2012. *Dynamic Service Composition and Deployment with OpenRichServices*. Springer Berlin Heidelberg, Berlin, Heidelberg, 120–146. https://doi.org/10.1007/978-3-642-30835-2_9
- [20] Changbin Liu, Boon Thau Loo, and Yun Mao. 2011. Declarative Automated Cloud Resource Orchestration. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (Cascais, Portugal) (SOCC '11)*. Association for Computing Machinery, New York, NY, USA, Article 26, 8 pages. <https://doi.org/10.1145/2038916.2038942>
- [21] Microsoft. 2022. Azure Resource Manager. <https://azure.microsoft.com/en-us/features/resource-manager/> (Accessed: 2022-07-12).
- [22] Kief Morris. 2021. *Infrastructure as Code: Dynamic Systems for the Cloud Age* (second ed.). O'Reilly Media, Inc., Sebastopol, CA, USA.
- [23] OASIS. 2013. Topology and Orchestration Specification for Cloud Applications Version 1.0. OASIS Standard, <http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html> (Accessed on 2020-09-25).
- [24] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. 2022. Smelly Variables in Ansible Infrastructure Code: Detection, Prevalence, and Lifetime. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. 61–72. <https://doi.org/10.1145/3524842.3527964>
- [25] Ruben Opdebeeck, Ahmed Zerouali, Camilo Velázquez-Rodríguez, and Coen De Roover. 2021. On the practice of semantic versioning for Ansible galaxy roles: An empirical study and a change classification model. *Journal of Systems and Software* 182 (2021), 111059. <https://doi.org/10.1016/j.jss.2021.111059>
- [26] Pulumi. 2022. Universal Infrastructure as Code. <https://github.com/pulumi/pulumi> (Accessed: 2022-07-12).
- [27] Pulumi. 2022. Testing. <https://www.pulumi.com/docs/guides/testing/> (Accessed: 2022-07-14).
- [28] Puppet. 2022. Powerful Infrastructure Automation and Delivery. <https://puppet.com/> (Accessed: 2022-07-12).
- [29] Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie Williams. 2019. A systematic mapping study of infrastructure as code research. *Information and Software Technology* 108 (2019), 65–77. <https://doi.org/10.1016/j.infsof.2018.12.004>
- [30] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The Seven Sins: Security Smells in Infrastructure as Code Scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 164–175. <https://doi.org/10.1109/ICSE.2019.00033>
- [31] Akond Rahman, Md Rayhanur Rahman, Chris Parnin, and Laurie Williams. 2021. Security Smells in Ansible and Chef Scripts: A Replication Study. *ACM Trans. Softw. Eng. Methodol.* 30, 1, Article 3 (Jan. 2021), 31 pages. <https://doi.org/10.1145/3408897>
- [32] Rajiv Ranjan, Boualem Benatallah, Schahram Dustdar, and Michael P. Papazoglou. 2015. Cloud Resource Orchestration Programming: Overview, Issues, and Directions. *IEEE Internet Computing* 19, 5 (2015), 46–56. <https://doi.org/10.1109/MIC.2015.20>
- [33] Julian Schwarz, Andreas Steffens, and Horst Lichter. 2018. Code Smells in Infrastructure as Code. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*. 220–228. <https://doi.org/10.1109/QUATIC.2018.00040>
- [34] Amazon Web Services. 2022. AWS Cloud Development Kit. <https://aws.amazon.com/cdk/> (Accessed: 2022-07-12).
- [35] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does Your Configuration Code Smell?. In *Proceedings of the 13th International Conference on Mining Software Repositories (Austin, Texas) (MSR '16)*. Association for Computing Machinery, New York, NY, USA, 189–200. <https://doi.org/10.1145/2901739.2901761>
- [36] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. 2021. Automating Serverless Deployments for DevOps Organizations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (Athens, Greece) (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 57–69. <https://doi.org/10.1145/3468264.3468575>
- [37] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. 2022. Change is the Only Constant: Dynamic Updates for Workflows. In *Proceedings of the 44th International Conference on Software Engineering (Pittsburgh, Pennsylvania) (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 350–362. <https://doi.org/10.1145/3510003.3510065>
- [38] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. 2022. Dependencies in DevOps Survey 2021: Version 2.0 (Until April 15, 2021). <https://doi.org/10.5281/zenodo.6372120>
- [39] Ricardo Terra and Marco Tulio Valente. 2009. A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience* 39, 12 (2009), 1073–1094. <https://doi.org/10.1002/spe.931>
- [40] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. 2007. Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates. *IEEE Transactions on Software Engineering* 33, 12 (2007), 856–868. <https://doi.org/10.1109/TSE.2007.70733>
- [41] Simon Friis Vindum and Emil Holm Gjørup. 2019. Hareactive. <https://github.com/funkia/hareactive> (Accessed: 2022-07-12).
- [42] Jöakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. 2018. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 223–236. <https://doi.org/10.1109/MASCOTS.2018.00030>
- [43] Denis Weerasiri, Moshe Chai Barukh, Boualem Benatallah, Quan Z. Sheng, and Rajiv Ranjan. 2017. A Taxonomy and Survey of Cloud Resource Orchestration Techniques. *ACM Comput. Surv.* 50, 2, Article 26 (May 2017), 41 pages. <https://doi.org/10.1145/3054177>
- [44] Michael Wurster, Uwe Breitenbücher, Michael Falkenthal, Christoph Krieger, Frank Leymann, Karoline Saatkamp, and Jacopo Soldani. 2020. The Essential Deployment Metamodel: A Systematic Review of Deployment Automation Technologies. *SICS Software-Intensive Cyber-Physical Systems* 35 (2020), 63–75. <https://doi.org/10.1007/s00450-019-00412-x>
- [45] Michael Wurster, Uwe Breitenbücher, Lukas Harzenetter, Frank Leymann, Jacopo Soldani, and Vladimir Yussupov. 2020. 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 - Volume 1: CLOSER*. INSTICC, SciTePress, 216–226. <https://doi.org/10.5220/0009794302160226>