Deployment Coordination for Cross-Functional DevOps Teams

Daniel Sokolowski
sokolowski@cs.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

ABSTRACT

Software stability and reliability are the core concerns of DevOps. They are improved by tightening the collaboration between developers and operators in cross-functional teams on the one hand and by automating operations through *continuous integration* (CI) and *infrastructure as code* (IaC) on the other hand. Ideally, teams in DevOps are fully independent. Still, their applications often depend on each other in practice, requiring them to coordinate their deployment through centralization or manual coordination.

With this work, we propose and implement the novel IaC solution μIS ([mju:z] *"muse"*), which automates deployment coordination in a *decentralized* fashion. μIS is the first approach that is compatible with the DevOps goals as it enables truly independent operations of the DevOps teams. We define our research problem through a questionnaire survey with IT professionals and evaluate the solution by comparing it to other modern IaC approaches, assessing its performance, and applying it to existing IaC programs.

CCS CONCEPTS

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

KEYWORDS

DevOps, Infrastructure as Code, Resource Orchestration, Cloud

ACM Reference Format:
Daniel Sokolowski. 2021. Deployment Coordination for Cross-Functional DevOps Teams. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3468264.3473101

1 INTRODUCTION

The goal of agile development—improving satisfaction through iterative feedback and higher velocity—has led to DevOps, aiming to reduce the friction between software development and operations. It unites developers, operators, and others in cross-functional teams, improving the collaboration between the professions to increase reliability and stability [20]. These goals are expressed in the *software delivery and operational (SDO) performance* of the organization, which is measured by Forsgren et al. [8] by the (1) deployment frequency, (2) lead time between development and production of changes, (3) required time to restore service on failure, and (4) the rate of failed changes.

Technically, DevOps drives operations automation through *continuous integration* (CI) and *infrastructure as code* (IaC) [19]. The latter leverages code-based techniques for operations, enabling the application of well-developed best practices from software development like versioning, testing, or reviewing. Earlier examples of IaC approaches are Chef [7] and Ansible [3], which allow server configuration through *procedural* scripts, and Puppet [26] that is *declarative*, i.e., its scripts describe a target state and the system automatically derives the operations to achieve it, providing better stability and less maintenance [29]. Meanwhile, these systems are often referred to as *configuration as code* (CaC) because they cannot *provision* infrastructure, which is required to use modern cloud infrastructure effectively, e.g., serverless functions, containers, databases, and storage. This demand led to declarative IaC solutions like Terraform [11], AWS CloudFormation [34], or Azure Resource Manager (ARM) [18], which use JSON, YAML, or other DSLs for their configuration. Lately, IaC solutions like Pulumi [24] and AWS CDK [33] became available, using general-purpose programming languages like TypeScript, Python, C#, or Go to describe the infrastructure. They allow developers to use already known languages with all their well-developed abstractions and tools.

Ideally, DevOps teams are independent. However, their applications often depend on each other in practice, requiring coordinating the times and order of deployments and changes. With today's IaC solutions, this coordination requires either (1) *centralization*, which is against the independence of teams, or (2) *manual coordination*, e.g., via phone, email, or chat, which contradicts DevOps' automation paradigm and requires synchronization between the teams, reducing the flexibility, reliability, and stability. In this work, we address deployment coordination for DevOps organizations with cross-functional teams to automate the deployment coordination in a *decentralized* fashion. Such decentralized automation enables DevOps organizations to be compatible with DevOps goals in the presence of inevitable application dependencies across teams, improving their SDO performance.

To identify the research problem in practice, we perform a questionnaire survey under IT professionals, assessing the existence of dependencies between their applications and whether these dependencies impact the order of application deployments and undeployments. To automate the coordination of decentralized deployments, we propose μIS ([mju:z] *"muse"*), a novel IaC system that treats deployments as continuously running processes—in contrast to the common perception that deployments are one-off tasks. We connect such deployments and implement a protocol for the controlled

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3473101

sharing and reactive updating of resources, achieving automated coordination across separate deployments and, thus, across independent teams. We plan to compare μIS with other IaC solutions and apply it to existing IaC script.

In the following, we present related work (§2) and our IaC solution for cross-functional DevOps teams in §3. Then we describe our evaluation plan (§4) before reporting the already achieved results in §5 and providing evidence for our contribution to knowledge (§6).

2 RELATED WORK

Resource Orchestrators. Weerasiri et al. [42] provide an overview on resource orchestration for the cloud. Following their reference architecture, μIS is a rule engine (it performs rule-based operations for decisions) and a policy enforcement engine (it takes decisions based on policies and external signals). Ranjan et al. [31] summarize the programming of resource orchestration operations. Various centralized orchestration solutions for virtualized containers exist, e.g., Kubernetes, Kubernetes Federation, Mesos, and Docker Swarm [6]. DOCMA [12] 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 [16] is a distributed policy enforcement engine for cloud orchestrators, enforcing orchestration policies expressing constraints and service-level agreements (SLAs).

Resource orchestrators create, update and delete resources, which is also required in our IaC approach. μIS uses resource orchestrators internally for these tasks. However, the focus of this work is on coordinating dependency fulfillment in a *decentralized* way. In contrast, resource orchestrators rely on centralized mechanisms to fulfill dependencies if they handle them at all.

Infrastructure as Code. Infrastructure as code [19] uses machine-readable code to configure and provision, i.e., deploy, systems. It enables applying best practices from software engineering to software operations, e.g., version control, static analysis, or code reviews. Various industrial IaC solutions are available (cf. §1) and Rahman et al. [27] performed a mapping study on their discussion in academia. A Terraform based approach for repeatable auto-scaling infrastructure in scientific computing is proposed by Balis et al. [4]. Guerriero et al. [10] performed semi-structured interviews with 44 developers, noticing that maintenance and evolution of IaC require better support and tooling.

Code quality has been studied for CaC systems. Sharma et al. [35] analyzed ~ 5 K Puppet projects, identified common code smells, and showed that configuration and design smells often go along. Schwarz et al. [32] generalized these findings because such smells are independent of the used technology. Rahman and Williams [30] conducted an empirical study on defective CaC scripts, identifying source code properties that correlate with defects. They also identify code smells through qualitative studies of Ansible, Chef, and Puppet scripts and propose corresponding static linters [28, 29].

μIS is a representative of the youngest generation of IaC solutions: It is declarative and leverages a general-purpose programming language. μIS supports various cloud providers and is based on Pulumi. In contrast to all other IaC solutions, μIS enables decentralized deployments with automated coordination.

Modeling Languages. Modeling languages specify architectures and behavior. E.g., the OASIS standard TOSCA [21] models cloud applications and their management. It describes topologies as a graph of components with their relationships. Operational behavior is described in *management plans* using existing workflow modeling languages, e.g., BPEL or BPMN. Bellendorf and Mann [5] give an overview of TOSCA cloud orchestration techniques, extensions, and tools. TOSCA was also applied to DevOps to integrate heterogeneous automation artifacts [43, 44].

Wurster et al. [46] describe the *Essential Deployment Metamodel* (EDMM), a least denominator metamodel of IaC solutions. *TOSCA Light* [47] is the EDMM-compatible TOSCA subset that was shown to be deployable with 13 IaC solutions using *TOSCA Lightning* [45].

Declarative IaC solutions and modeling languages describe system architectures. However, modeling languages also describe operational behavior, which is automatically derived in declarative IaC. The resource graph model of μIS is inspired and compatible with the EDMM. Therefore, system descriptions in modeling languages like TOSCA could be automatically converted to μIS, which then provides the operationalization and runtime to execute these specifications in DevOps settings out-of-the-box. This also enables to use (graphical) modeling tools, e.g., from the TOSCA ecosystem, as description languages for μIS.

Architecture Description Languages. Architecture Description Languages (ADLs) define the component-level structure of an application. Medvidovic and Taylor [17] define that ADLs specify components, connections between them and their configuration, and they provide tools for development and evolution. ADLs exist on various levels, e.g., ArchJava [2] defines components in Java and ORS [15] treats entire services as components. Terra and Valente [38, 39] propose a DSL to enforce constraints on structural dependencies in object-oriented software.

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 (connections). 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.

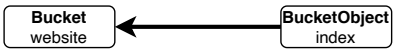
3 AUTOMATING DECENTRALIZED DEPLOYMENT COORDINATION

In descriptive IaC systems, users define a directed acyclic graph (DAG) 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 [46] between the two resources. 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.

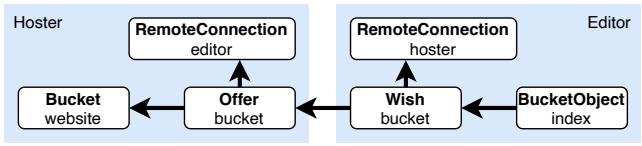
We use a simple static website with a single `index.html` page hosted in an AWS S3 bucket as a running example. The deployment description is in Listing 1 and defines Figure 1a; the index must be deployed after and undeployed before the bucket. While this is unrealistically simple, it suffices to showcase our approach.

Listing 1: Centralized deployment description of the website.

```
1.1 const bucket = new aws.s3.Bucket('website', {
1.2   website: { indexDocument: 'index.html' }
1.3 }); // Creates the S3 bucket for the static website
1.4 const index = new aws.s3.BucketObject('index', {
1.5   bucket, content, key: 'index.html'
1.6 }); // Saves the index.html page in the website's bucket
```



(a) Centralized.



(b) Decentralized using μS offers and wishes.

Figure 1: Resource graph of the website deployment.

3.1 Decentralized Deployment

In DevOps, ideally, each team operates its resources independently, including deployment. In the website example, the *hoster* could be 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 manually coordinate whenever the bucket is deployed, updated or undeployed. This decreases the flexibility of the teams and wastes time due to synchronization. To decouple the deployment times, we propose to treat deployments not as one-off tasks—as all common IaC systems do—but as continuously running processes, updating the deployed resources reactively based on the deployment description *and* external signals, e.g., changes in other deployments. The editor can start their deployment independently and run it continuously. Whenever the hoster starts or updates their deployment, the editor's deployment automatically deploys, updates, or undeploys the index page, without manual intervention.

To enable such behavior, the connections between deployments and inter-deployment resource dependencies must be explicit. For this we propose three new resource types: `RemoteConnection` for connections to other deployments, `Offer` to provide resources or information to a connected deployment, and `Wish` to access the offer of a connected deployment. Using them, the hoster and editor specify their connection (Lines 2.1 and 3.1). The hoster offers their bucket (Lines 2.2 to 2.4) to the editor's deployment in Line 2.5. The editor specifies their expectation of the offer by defining a wish in Line 3.2, allowing to use the offered bucket via `wish.offer` (Line 3.4). Together Listings 2 and 3 define the resource graph Figure 1b.

3.2 Enabled Use Cases

Asynchronous deployment across teams. μS enables teams to start their deployments independently and deploys resources asynchronously once their dependencies are fulfilled. E.g., the editor starts their deployment before the hoster. First, the index is not deployed because its dependency on the bucket is unsatisfied. Once the hoster deploys the bucket, μS automatically deploys the index.

Listing 2: μS deployment description of the hoster.

```
2.1 const editor = new RemoteConnection('editor');
2.2 const bucket = new aws.s3.Bucket('website', {
2.3   website: { indexDocument: 'index.html' }
2.4 });
2.5 new Offer(editor, 'bucket', bucket);
```

Listing 3: μS deployment description of the editor.

```
3.1 const hoster = new RemoteConnection('hoster');
3.2 const wish = new Wish<aws.s3.Bucket>(hoster, 'bucket');
3.3 const index = new aws.s3.BucketObject('index', {
3.4   bucket: wish.offer, content, key: 'index.html'
3.5 });
```

Safe undeployment across teams. μS ensures that all resources depending on a resource R are not deployed anymore when R is undeployed. If R shall be undeployed, the undeployment of all resources depending on it is triggered and R is only undeployed after their undeployment is completed. E.g., when the hoster undeploys the bucket, μS automatically undeploys the index page before.

Reactive updates across teams. μS automatically transports configuration changes across the teams' deployments and triggers reactive updates. E.g., the editor might show the bucket's name (`wish.offer.name`) in the page's content. If the hoster updates the bucket name, this change is transported to the editor's deployment, automatically updating the index page's content.

3.3 System Architecture and Design

The runtime of a μS deployment comprises three components: (1) The *interpreter* generates the *target state* from the deployment description and a snapshot of the current values of all external signals, e.g., remote offers. The target state is a snapshot of the resource graph described by the deployment description where all resources depending on *unsatisfied* wishes are pruned, e.g., if the bucket offer is not available, the index page is absent in the target state of the editor's deployment. (2) The *driver* reads the current deployment state from persistent storage, deploys, updates, and undeploys resources to achieve the target state, and persists the new deployment state for its next invocation. (3) The *reactive engine* is the only continuously executing component and records changes of external signals, e.g., when a remote offer changes. It triggers the interpreter on changes, which subsequently runs the driver.

To ensure correct behavior, the interpreter and driver may not execute multiple times in parallel. Thus, the reactive engine buffers all changes between invocation of the interpreter and completion of the driver. If during this period a change was observed, the interpreter is directly invoked again. Otherwise the invocation is delayed until the next external change is observed.

Each team can exchange its deployment description, allowing the update of the deployment. Such updates take effect on the next run of the interpreter and driver. Also, the resources deployed by a deployment continue to exist when the deployment stops or crashes. Thus, the deployed applications continue operating when their deployment goes down. However, they will not be updated until the deployment starts again. Across teams, the unavailability

of another deployment delays the deployment of its resources. Also, unavailability delays the undeployment of a resource on which resources of the unavailable deployment (transitively) depend.

4 EVALUATION

We evaluate the existence of the research problem and μIS' effectiveness, performance, and applicability to existing IaC scripts.

To identify the demand for decentralized automated deployment coordination, we perform an empirical study with IT professionals that follows the ACM SIGSOFT guidelines for questionnaire surveys [1] and advice from Kasunic [13] and Kitchenham and Pfleeger [14]. We want to find out: (1) How many dependencies do applications have to other applications? (2) Do such dependencies constrain their order of deployment and undeployment? (3) How are deployments coordinated in practice? (4) Do practitioners believe that automated coordination provides better SDO performance than manual coordination? (5) Does the organization's SDO performance influence the answers to the previous questions? For the SDO performance-related questions, validated instruments from Forsgren et al. [8] are reused.

To evaluate the effectiveness of μIS, we re-implement the deployment of existing applications and compare them with deployments in systems like Pulumi or AWS CDK. The amount of coordination overhead reduction shall be evaluated, and code-based metrics indicating the complexity shall be compared, e.g., lines of code, resource objects, etc. To evaluate μIS' performance, we implement a set of microbenchmarks in μIS and other systems like Pulumi and AWS CDK, covering simple, typical deployments. Pulumi's examples repository [23] is the starting point. With this benchmark suite, we compare the deployment duration, obtain insight into μIS' behavior, measure the delay to adapt to changes reactively, and assess the resource consumption. To assess the adaptability, we migrate existing IaC programs to μIS. We start with Pulumi TypeScript programs using stack references [25], Pulumi's feature to explicitly model dependencies across deployments. These scripts are compatible with μIS out-of-the-box; however, to leverage automated coordination, the stack references need to be converted to offers and wishes. To evaluate the migration from other IaC solutions, focusing on their features explicitly modeling dependencies across deployments, we use import and conversion tools from the Pulumi community [22] to obtain a μIS compatible version. Then we enable μIS' automated deployment coordination by applying transformations based on information from the original programs.

5 ACHIEVED RESULTS

We organized the *Dependencies in DevOps Survey 2021* from January to April 2021. It was filled by 134 IT professionals working in industry, who were advertised through snowball sampling [9] on a DevOps mailing list, social media, and in the authors' personal networks. The central insights are that (1) the majority of applications depend on others, (2) such dependencies usually impact the deployment and undeployment order, and (3) deployments across teams typically rely on manual coordination, even though (4) automated coordination promises better SDO performance. The results show the demand for automated deployment coordination.

A first version of μIS is implemented as a TypeScript library using Pulumi as driver and Hareactive [40] for the reactive engine [37] and presented in [36]. With it, a set of initial evaluations was performed. First, the deployment of the TeaStore [41] was implemented in μIS, Pulumi, and AWS CDK. μIS required 14% and 35% more lines of code. μIS enables decentralized automated deployment coordination with small definition overhead. Second, in a microbenchmark μIS' deployment duration for a single microservice was compared, showing similar performance to Pulumi and better performance than AWS CDK. Third, the microservice was deployed multiple times with a serial and a parallel chain of dependencies between them. μIS deploys the services—as expected—in sequence and parallel, each requiring roughly the same time measured in the single-service experiment. Fourth, we transformed stack references in 64 Pulumi projects from Github to μIS offers and wishes. It is easy to adopt μIS' automated deployment coordination in decentralized Pulumi deployments using stack references.

6 CONTRIBUTION TO KNOWLEDGE

Research on IaC so far either focuses on modeling approaches like TOSCA or CaC solutions like Ansible, Chef, or Puppet (cf. §2). Recent IaC systems that support infrastructure provisioning and leverage general-purpose programming languages, e.g., Pulumi and AWS CDK, are not discussed in scientific research yet. μIS is an entrance into this direction, and we presume the discussion of various problems in this field. For instance, such IaC solutions start dissolving the separation of infrastructure and application code, which could be further blurred for, e.g., safe updating and dynamic adaption. Also, we expect work on specializing debugging and testing techniques of general-purpose languages for IaC, improving the currently underdeveloped field of debugging and testing IaC [10].

The proposed decentralized mechanism to automate the deployment coordination is novel and enables further decoupling of cross-functional teams, improving DevOps. It can be further contributed back to modeling languages and implemented at the level of resource orchestrators, enabling decentralized federations suitable to span across organizations, which is not the case for today's centralized orchestrator federation solutions.

7 CONCLUSION

DevOps aims for decoupled cross-functional teams, each independently developing and operating their applications. However, applications depend on other teams' applications, requiring *decentralized*, asynchronous deployment coordination. We evaluate the practical relevance of this problem and solve it by proposing the μIS IaC system, which automates decentralized deployment coordination. We implement μIS and describe its evaluation for effectiveness, performance, and applicability to existing IaC scripts.

ACKNOWLEDGMENTS

The author is advised by Prof. Dr. Guido Salvaneschi, who is associate professor at the University of St. Gallen. This work has been co-funded by the German Research Foundation (DFG, No. 383964710, SFB 1119) and by the Hessian LOEWE initiative (emergenCITY and Software-Factory 4.0).

REFERENCES

[1] ACM Special Interest Group on Software Engineering . 2021. Empirical Standards: Questionnaire Surveys. https://github.com/acmsigsoft/EmpiricalStandards/blob/master/docs/QuestionnaireSurveys.md, last accessed on 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] Ansible and Red Hat. 2021. Ansible is Simple IT Automation. https://www.ansible.com/, last accessed: 2021-05-06.

[4] Bartosz Balis, Michal Orzechowski, Krystian Pawlik, et al. 2020. Cloud Infrastructure Automation for Scientific Workflows. In *Parallel Processing and Applied Mathematics*, Roman Wyrzykowski, Ewa Deelman, Jack Dongarra, et al. (Eds.). Springer International Publishing, Cham, 287–297. https://doi.org/10.1007/978-3-030-43229-4_25

[5] 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

[6] Brendan Burns, Brian Grant, David Oppenheimer, et al. 2016. Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade. *Queue* 14, 1 (Jan. 2016), 70–93. https://doi.org/10.1145/2898442.2898444

[7] Chef. 2021. Chef Software DevOps Automation Tools & Solutions. https://chef.io, last accessed: 2021-05-06.

[8] Nicole Forsgren, Dustin Smith, Jez Humble, et al. 2019. 2019 Accelerate State of DevOps Report. https://services.google.com/fh/files/misc/state-of-devops-2019.pdf, last accessed on 2021-05-05.

[9] Leo A. Goodman. 1961. Snowball Sampling. *The Annals of Mathematical Statistics* 32, 1 (1961), 148–170. http://www.jstor.org/stable/2237615

[10] M. Guerriero, M. Garriga, D. A. Tamburri, et al. 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

[11] HashiCorp. 2021. Terraform. https://www.terraform.io/, last accessed: 2021-05-06.

[12] L. L. Jiménez and O. Schelén. 2019. DOCMA: A Decentralized Orchestrator for Containerized Microservice Applications. In *2019 IEEE Cloud Summit*. 45–51. https://doi.org/10.1109/CloudSummit47114.2019.00014

[13] Mark Kasunic. 2005. Designing an Effective Survey. https://doi.org/10.1184/R1/6573062.v1

[14] Barbara A. Kitchenham and Shari L. Pfleeger. [n.d.]. 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.

[15] 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

[16] 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

[17] Nenad Medvidovic and Richard N. Taylor. 2000. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering* 26, 1 (2000), 70–93. https://doi.org/10.1109/32.825767

[18] Microsoft. 2021. Azure Resource Manager. https://docs.microsoft.com/en-us/azure/azure-resource-manager/management/overview, last accessed: 2021-05-06.

[19] Kief Morris. 2016. *Infrastructure as Code: Managing Servers in the Cloud* (1st ed.). O'Reilly Media, Inc.

[20] Kristian Nybom, Jens Smeds, and Ivan Porres. 2016. On the Impact of Mixing Responsibilities Between Devs and Ops. In *Agile Processes, in Software Engineering, and Extreme Programming*, Helen Sharp and Tracy Hall (Eds.). Springer International Publishing, Cham, 131–143. https://doi.org/10.1007/978-3-319-33515-5_11

[21] 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, last accessed on 2020-09-25.

[22] Pulumi. 2021. Adopting Pulumi. https://www.pulumi.com/docs/guides/adopting, last accessed on 2021-05-05.

[23] Pulumi. 2021. Pulumi Examples. https://github.com/pulumi/examples, last accessed on 2021-05-05.

[24] Pulumi. 2021. Pulumi: Modern Infrastructure as Code. https://github.com/pulumi/pulumi, last accessed: 2021-02-25.

[25] Pulumi. 2021. Stacks: Stack References. https://www.pulumi.com/docs/intro/concepts/stack/#stackreferences, last accessed: 2021-05-05.

[26] Puppet. 2021. Powerful Infrastructure Automation and Delivery. https://puppet.com/, last accessed: 2021-05-06.

[27] 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

[28] 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

[29] Akond Rahman, Md Rayhanur Rahman, Chris Parnin, et al. 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

[30] Akond Rahman and Laurie Williams. 2019. Source code properties of defective infrastructure as code scripts. *Information and Software Technology* 112 (2019), 148 – 163. https://doi.org/10.1016/j.infsof.2019.04.013

[31] Rajiv Ranjan, Boualem Benatallah, Schahram Dustdar, et al. 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

[32] 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

[33] Amazon Web Services. 2021. AWS Cloud Development Kit. https://aws.amazon.com/cdk/, last accessed: 2021-05-06.

[34] Amazon Web Services. 2021. AWS CloudFormation - Infrastructure as Code & AWS Resource Provisioning. https://aws.amazon.com/cloudformation/, last accessed: 2021-05-06.

[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 European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE '21)*. ACM, New York, NY, USA. https://doi.org/10.1145/3468264.3468575

[37] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. 2021. *µs Infrastructure as Code*. https://doi.org/10.5281/zenodo.4902323

[38] Ricardo Terra and Marco Tulio de Oliveira Valente. 2008. Towards a Dependency Constraint Language to Manage Software Architectures. In *Software Architecture*, Ron Morrison, Dharini Balasubramaniam, and Katrina Falkner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 256–263. https://doi.org/10.1007/978-3-540-88030-1_19

[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] Simon Friis Vindum and Emil Holm Gjørup. 2019. Hareactive: Purely Functional Reactive Programming Library. https://github.com/funkia/hareactive, last accessed: 2021-02-25.

[41] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, et al. 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

[42] Denis Weerasiri, Moshe Chai Barukh, Boualem Benatallah, et al. 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

[43] Johannes Wettinger, Uwe Breitenbücher, and Frank Leymann. 2014. Standards-Based DevOps Automation and Integration Using TOSCA. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC '14)*. IEEE Computer Society, USA, 59–68. https://doi.org/10.1109/UCC.2014.14

[44] Johannes Wettinger, Uwe Breitenbücher, Oliver Kopp, et al. 2016. Streamlining DevOps automation for Cloud applications using TOSCA as standardized metamodel. *Future Generation Computer Systems* 56 (2016), 317 – 332. https://doi.org/10.1016/j.future.2015.07.017

[45] Michael Wurster, Uwe Breitenbücher, Lukas Harzenetter, et al. 2020. TOSCA Lightning: An Integrated Toolchain for Transforming TOSCA Light into Production-Ready Deployment Technologies. In *Advanced Information Systems Engineering*, Nicolas Herbaut and Marcello La Rosa (Eds.). Springer International Publishing, Cham, 138–146. https://doi.org/10.1007/978-3-030-58135-0_12

[46] Michael Wurster, Uwe Breitenbücher, Michael Falkenthal, et al. 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

[47] Michael Wurster., Uwe Breitenbücher., Lukas Harzenetter., et al. 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