Automating Serverless Deployments for DevOps Organizations

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

Pascal Weisenburger
pascal.weisenburger@unisg.ch
University of St. Gallen
Switzerland

Guido Salvaneschi
guido.salvaneschi@unisg.ch
University of St. Gallen
Switzerland

ABSTRACT

DevOps unifies software development and operations in cross-functional teams to improve *software delivery and operations* (SDO) performance. Ideally, cross-functional DevOps teams *independently* deploy their services, but the correct operation of a service often demands other services, requiring coordination to ensure the correct deployment order. This issue is currently solved either with a central deployment or manual out-of-band communication across teams, e.g., via phone, chat, or email. Unfortunately, both contradict the independence of teams, hindering SDO performance—the reason why DevOps is adopted in the first place.

In this work, we conduct a study on 73 IT professionals, showing that, in practice, they resort to manual coordination for correct deployments even if they expect better SDO performance with fully automated approaches. To address this issue, we propose μIS ([mju:z] *"muse"*), a novel IaC system automating deployment coordination in a fully decentralized fashion, still retaining compatibility with the DevOps practice—in contrast to today's solutions. We implement μIS, demonstrate that it effectively enables automated coordination, introduces negligible definition overhead, has no performance overhead, and is broadly applicable, as shown by the migration of 64 third-party IaC projects.

CCS CONCEPTS

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

KEYWORDS

DevOps, Infrastructure as Code, Cloud, Serverless Computing

ACM Reference Format:
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 (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3468264.3468575

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

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468575

1 INTRODUCTION

While agile methods had a deep influence on software in IT organizations, software *development* and *operations* are traditionally separated. Operations summarizes all activities after the development, including configuration, resource provisioning and deployment, monitoring, alarming, reporting, and support. The widespread adoption of agile methods [17] set the focus on changing requirements and software quality, aiming for minimal change response time. Operations, however, focuses on stability and reliability, which are typically assumed to be threatened by frequent change. *DevOps* aims to mitigate this tension: (1) Organizationally, DevOps strengthens collaboration between development and operations staff, often, by unifying both tasks in cross-functional teams [27]. (2) Technically, DevOps leverages code-based tools for operations, too, achieving a high degree of automation (*continuous delivery*). Both lead to a smoother workflow [33], improving service quality [27] and change management performance [13].

In DevOps, *Infrastructure as Code* (IaC) is a core technique [32] to automate software deployment by 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 the domain of "traditional" application code. IaC has shown a tremendous impact on the speed of change. For example, introducing IaC scripts in Ambit Energy has increased deployment frequency by a factor of 1,200 [38]. Intercontinental Exchange (ICE) uses IaC to maintain 75% of its 20 K servers, reducing the time to provision development environments from 1–2 days to 21 minutes [39].

We focus on IaC for serverless computing, e.g., as in AWS CloudFormation, Azure Resource Manager, or Terraform. Serverless computing [9] is a paradigm for deploying cloud applications close to the cloud's original pay-as-you-go concept and simplifies operational concerns regarding application availability, scalability, and fault tolerance [15]. Serverless infrastructure is therefore an excellent fit for DevOps because it reduces the required operations knowledge, making specialized operations teams obsolete [16]. We refer to serverless computing to include not only function-as-a-service (FaaS) but all offerings where the cloud provider transparently manages the provisioning and scaling of underlying servers, in agreement with the current use of "serverless computing" at major cloud providers [7, 20, 31], e.g., serverless containers, serverless databases, or serverless storage solutions.

Implementing DevOps with cross-functional teams following a service-based architecture ideally implies that each team independently deploys its own services. However, the correct operation of a service often depends on the availability of other services. If these dependencies span across teams, their deployments have to be coordinated to ensure that dependencies are not violated.

Unfortunately, existing IaC solutions to orchestrate multiple deployments (e.g., AWS CloudFormation, StackSet, or Sceptre) are centralized and cannot ensure that dependencies between services in deployments of different teams are satisfied. Hence—as we show in this paper—such dependencies require *manual* coordination of deployments, i.e., teams have to communicate *out-of-band* to ensure the correct deployment order, e.g., via phone, chat, or email. This approach is problematic because it (1) slows down software evolution [13, 27] and (2) is error-prone because of potential miscommunication between teams [33]. This also applies to approaches from academia, where work so far either focuses on centralization (cf. Section 10.1, 10.2, and 10.4), which counteracts the decoupling of the teams, or does not provide *executable* descriptions of the infrastructure (cf. Section 10.2 and 10.3), which hinders automation. Beyond previous work, we strive to provide decentralized and automated management for dependencies among deployments.

In this paper, we conduct an empirical, cross-sectional, online questionnaire survey on 73 IT professionals, highlighting the issue of manual coordination and showing the need for better automation in DevOps. We close this gap by proposing μS ([mju:z] *"muse"*), a novel IaC system. μS fully automates deployment coordination in a decentralized fashion, enabling teams to execute their deployments independently and asynchronously with no need for manual coordination between teams. μS deployments decentrally ensure that services are only deployed when their dependencies are available and undeployed otherwise. We show that μS effectively automates the deployment coordination for a representative service-based application on AWS, we run benchmarks indicating no performance overhead compared to industrial-strength competitors, and we port 64 third-party projects to μS. In summary:

- We survey 73 IT professionals showing that application dependencies are very common and that, in practice, developers resort to manual coordination for correct deployments.
- We design μS, an IaC approach that ensures correct deployment across teams with neither centralization nor manual coordination, ensuring safe deployments in DevOps organizations.
- We implement μS as an IaC language and novel continuous deployment runtime by extending the Pulumi SDK.
- We evaluate μS, showing that it is effective in coordinating decentralized deployments, it is efficient, and it is easily applicable.

2 THE DEPLOYMENT COORDINATION ISSUE

To better understand the issues related to deployment coordination, we conducted a cross-sectional, self-administered, online questionnaire survey with 73 IT professionals[1] about software dependencies and deployment order at their organization. The raw data and a technical report with its full description and results, including the adherence to the survey standards of the ACM SIGSOFT community [1], are publicly available [45] under CC BY 4.0.

The respondents have various professional experience levels (Figure 1d). They were reached through snowball sampling [23] in the authors' personal network (90%) and via social media (10%). Their companies cover a broad spectrum of sizes (Figure 1e) and sectors (Figure 1a). Most participants have development or operations

[1] Assuming 55.3 million IT professionals worldwide [24] and a confidence interval of 95% entails an error margin of 11%.

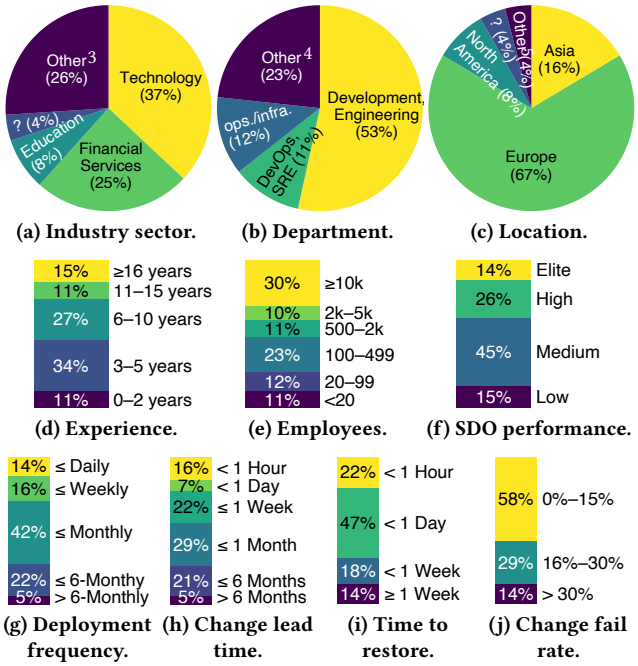


Figure 1: Survey demographics and SDO performance.

tasks and are located in Europe (Figures 1b and 1c). Albeit, our data does not indicate significantly different responses in other regions.[2]

We measure the companies' *software delivery and operational* (SDO) performance with an instrument by Forsgren et al. [19], 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. Our results for these questions are in Figures 1g to 1j. Forsgren et al. found that the metrics correlate and form four clusters (low, medium, high, and elite). In Figure 1f, we apply this clustering to our respondents' companies using minimal euclidean distance, mainly having medium and high SDO performance (71%). Our study leads to the following research insights.

RI1: Most applications depend on other applications. Only 16% of the participants' primary applications do not require another application for their correct operation, while 2–5 dependencies are common (42%). 19% depend on more than 10 other applications (Figure 2a). Dependencies among applications are very common.

RI2: Dependencies between applications constrain the order of their deployment. Only 12% of the participants state that dependencies do not constrain the deployment order, while 21% answer that they do; the answers in between (67%) mean that some dependencies imply a deployment order (Figure 2b). The likelihood of such deployment order has a significant negative correlation[6]

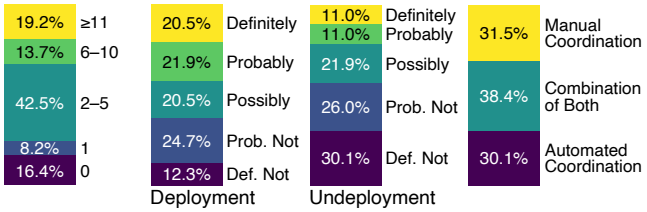
[2] Using Kruskal-Wallis test with posthoc Wilcoxon signed-rank test with $\alpha = 5\%$.

[3] Product Manager (8%), Manager (4%), Consultant, Coach or Trainer (4%), Information Security (3%), C-level Executive (1%), Release Engineering (1%), Other (1%).

[4] Industrials & Manufacturing (7%), Retail/Consumer/e-Commerce (5%), Telecommunications (4%), Media/Entertainment (3%), Non-profit (3%), Healthcare & Pharmaceuticals (1%), Government (1%), Energy (1%).

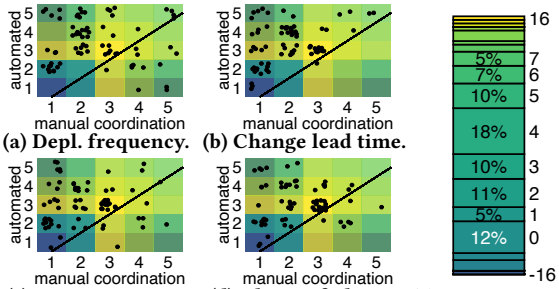
[5] Africa (1%), Oceania (1%), South America (1%).

[6] Using both Kendall and Spearman rank correlation coefficients with $\alpha = 5\%$.



(a) Number of dependencies. **(b) Dependency implies deployment order.** **(c) Coordination method.**

Figure 2: State of dependencies and deployment order.



(a) Depl. frequency. **(b) Change lead time.** **(c) Time to restore.** **(d) Change fail rate.** **(e) Total difference.**

Figure 3: Assumed SDO performance compared to no coordination (3: similar, <3: worse, >3: better). Jitter added to the data points for better visualization on the discrete scales.

with the company's SDO performance, 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. Still, the responses for deployment and undeployment have a significant positive correlation.[6] In summary, our survey indicates 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 [18].

RI3: Deployments across teams need manual coordination. 70% of the participants rely on manual coordination to ensure the correct deployment order across teams (Figure 2c). 32% do not support the deployment coordination with automation at all. The more respondents rely on manual coordination, the stronger they agree that dependencies constrain the deployment order in RI2.[6]

RI4: IT professionals think that automated coordination has better SDO performance than manual coordination. For each SDO performance metric, we asked separately for automated and for manual coordination, whether they are likely to be similar (=3), better (>3), or worse (<3) compared to a scenario with no required coordination (Figures 3a to 3d). In Figure 3a, as an example, a data point $(3, 3)$ means that both coordination methods lead to similarly frequent deployments, and a point $(2, 4)$ means that manual coordination has less frequent deployments and automated coordination has more frequent ones. The majority of participants believe that automated coordination has similar or better SDO performance than manual coordination, i.e., most data points are above the diagonal. The sum of the differences between the scores for automated and manual coordination over the four SDO metrics yields the total difference in $[-16, 16]$ for each participant (Figure 3e), showing that

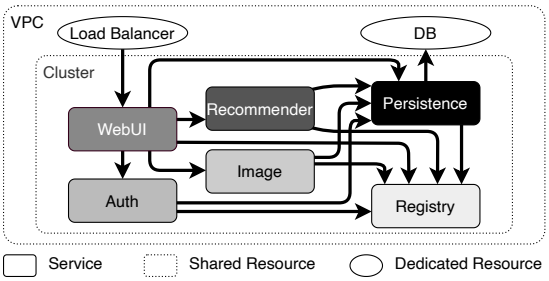


Figure 4: Components and functional dependencies of the TeaStore. Every service is managed by a dedicated team.

79% assume better SDO performance with automated coordination (>0) and only 8% assume worse SDO performance (<0).

Our survey confirms that applications depend on each other (RI1), often constraining their deployment order (RI2). In most cases, manual coordination is used to ensure the correct deployment order across teams (RI3)—even though IT professionals assume better SDO performance with automated coordination solutions (RI4). These results indicate the need to automate the deployment coordination among teams, which we address in the following section.

3 DEPLOYMENT COORDINATION IN DEVOPS

To illustrate deployment coordination techniques for DevOps organizations, we consider the TeaStore application [53], a case study on online retailing for benchmarking and modeling service software that we use as a running example throughout the paper. TeaStore is a representative case study according to RI1–RI4 (cf. Section 2). TeaStore consists of six services (Figure 4): (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 services backed by a *Database* (DB), (6) *Registry* lists all service instances for load balancing. All components reside in a single *virtual private cloud* (VPC). They run as serverless container services (as offered, e.g., by AWS Fargate [6]) within the same *container cluster* and the DB is serverless, too (as offered, e.g., by AWS RDS [4]).

TeaCorp, TeaStore's company, adopts DevOps: each service in Figure 4 is developed and operated by a dedicated team managing the service's infrastructure and deployment. Also, the WebUI team manages the load balancer, the Persistence team its service's DB, and the Registry team maintains the shared VPC and cluster.

3.1 Dependencies and Coordination

The arrows in Figure 4 are dependencies between services, e.g., Persistence depends on DB and Registry. Apart from Registry, every TeaStore service requires two to five (in case of WebUI) other applications for its correct operation—a common case (RI1). Dependencies between services constrain their deployment order (RI2) because each service requires that its dependencies are satisfied for correct operation. For instance, before Persistence is deployed, DB and Registry must be up and running. Symmetrically, DB and Registry should not be undeployed before Persistence. Such dependencies are not limited to services but may refer to any infrastructure entity, e.g., Persistence also depends on the cluster. Ensuring

that a service is only deployed when its dependencies are satisfied requires deployment coordination, as we discuss next.

Manual Coordination. Most commonly, deployments are coordinated manually (RI3). Current IaC solutions provide limited support for this approach: each team independently maintains a script for deploying its services without any guarantee that dependencies are satisfied across teams (Section 10.4). As a result, the teams in TeaCorp manually coordinate tasks and have to 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 [37], a TeaCorp team can give other teams access to its deployment state, ensuring that a service is only deployed when its dependencies to other teams' deployments are met. Yet, this guarantee is not provided for the undeployment order of services, potentially leaving the system in an inconsistent state. Also, teams still have to manually coordinate their operations to deploy and undeploy in a 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 reduce the organization's SDO performance.

Automated Coordination. Automated coordination promises better SDO performance (RI4). Yet, existing automated deployment solutions, e.g., AWS CloudFormation, AWS CDK, Terraform or Pulumi, are centralized (Section 10.4): all teams delegate the deployment of their services 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 the *"you build it, you run it"* principle of DevOps. It is likely to reduce the SDO performance, as communication across teams is required for (1) all changes and (2) application improvements based on operational insights.

3.2 μS: Automated Coordination for DevOps

We propose μS, which solves the deployment issue for DevOps organizations by *decentralizing* the automated deployment coordination. With μS, teams independently specify their deployments in the μS_ℓ language—similar to what they may do today with, e.g., AWS CDK or Pulumi. In contrast to these solutions, however, μS provides a mechanism to satisfy dependencies across deployments without manual coordination. In μS_ℓ, developers define a *wish* from another deployment and deploy their services dependent on its satisfaction. For example, in TeaCorp, the Auth team defines wishes for Registry and Persistence and lets its service depend on them so that the Auth service is only deployed when the wishes are satisfied. The Registry and Persistence teams satisfy these wishes by defining a corresponding *offer* in their deployments.

To automate deployment coordination, the execution of each team's deployment is a continuously running process of the μS runtime and not—as common today—a one-off task. The deployments communicate and ensure that services depending on wishes are only deployed when the wishes are satisfied by corresponding offers. For example, the Auth team's deployment automatically deploys its service when the offers from Registry and Persistence are available. Further, when an offer is withdrawn and a wish becomes unsatisfied, μS ensures that the Auth service is undeployed first.

Our solution guarantees the correct deployment and undeployment order for dependencies across deployments of different teams without introducing a central authority nor requiring manual coordination. Thus, μS enables safe deployments in DevOps organizations with cross-functional teams.

4 DEPLOYMENT DEFINITIONS

We now present μS_ℓ, μS' definition language for deployments.

4.1 Deployment Graphs

A *deployment graph* is a directed acyclic graph (DAG) where nodes r are *resources*, i.e., infrastructure entities like containerized services, load balancers, or network security policies. Arcs (r, r') represent *dependencies* between resources, describing *requires* or *hosted-by* relationships. Dependencies are transitive and constrain the deployment order, i.e., for each arc, the target r' is deployed before the source r. For instance, if a container depends on a cluster, the cluster is deployed before the container and the container is undeployed before the cluster. To ensure that the deployment order is decidable, the deployment graph must be acyclic.

Crucially, μS safely connects independent deployment graphs by *inter-deployment dependencies*, i.e., arcs between nodes in different deployment 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 deployment graph allows another deployment 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 deployment graphs, enabling transitive dependencies among resources across separate deployments. To retain the decidability of the deployment order, inter-deployment dependencies may not introduce cyclic dependencies.

For example, at TeaCorp, each team maintains a separate deployment graph, modeling the team's resources and dependencies. Figure 5 shows the Auth team's deployment 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 deployment graphs through inter-deployment dependencies forms the *global deployment 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 deployment graphs of all teams.

4.2 Deployment Definitions in μS_ℓ

μS_ℓ enables developers to define deployment graphs in *deployment definitions*. They describe the topology of the graph and the configuration of its resources. μS_ℓ uses TypeScript as a host language, retaining all TypeScript features, including OOP abstractions like classes and inheritance. Further, it extends Pulumi, achieving full

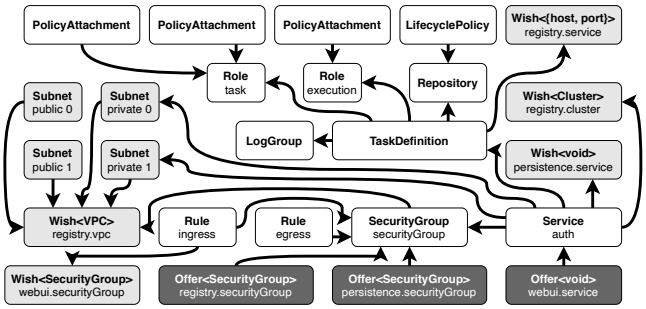


Figure 5: The Auth team's deployment graph.

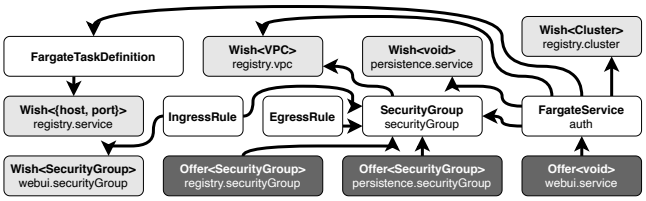


Figure 6: Simplified developer view on Figure 5.

compatibility with existing Pulumi TypeScript projects. Listing 1 shows the Auth team's deployment definition for Figure 5.

In μS_ℓ, resources are objects inheriting from `Resource`. Instantiating `Resource` creates a node r in the deployment graph, e.g., Lines 1.7 and 1.8 instantiate a resource since `SecurityGroup` inherits from `Resource`. Referencing another resource r' from a resource r defines a dependency, hence, an arc (r, r') in the deployment graph. There are two ways to establish such dependencies. First, dependencies can be defined using `dependsOn`, e.g., the Auth service (Lines 1.15 to 1.28) requires the Persistence service (Line 1.28). Second, resources can declare *input* and *output* values: using a resource or its output as another resource's input defines a dependency. Inputs are configuration values used when instantiating a resource, e.g., in Line 1.8, the security group is configured with its VPC and an egress rule. Outputs are values resulting from the deployment and are accessible via object properties (e.g., a security group's id in Line 1.10) or getter methods (e.g., VPC's private subnets' ids in Line 1.17). For instance, the Auth service also depends on a cluster (Line 1.16), the VPC (Line 1.17), the Registry service (Lines 1.24 and 1.25), and the security group (Line 1.27).

Inheritance enables reuse and refinement of resource definitions. Inherited definitions simplify the developer view on the deployment graph because parts of it are implicitly defined in the superclass. Hence, inheritance can be used to collapse a region of the graph into a node, hiding the dependencies within the region. For example, the deployment graph in Figure 5 is simplified to the developer view in Figure 6 (described in Listing 1) because `Remote`, `SecurityGroup` and `FargateSevice` inherit from superclasses: (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`.

4.3 Connecting Deployment Definitions

To define *inter-deployment* dependencies using offers and wishes, μS_ℓ deployment definitions reference remote deployment definitions as *remote objects* (cf. instances of `Remote` in Lines 1.3 to 1.5). The

Listing 1: The Auth team's μS_ℓ deployment definition.

```
1.1  interface RegistryWishes { vpc: Vpc, cluster: Cluster,
1.2    svc: { host: string, port: number } }
1.3  const reg = new Remote<RegistryWishes>(registryKey);
1.4  const persistence = new Remote<{ svc: void }>(persistncKey);
1.5  const webui = new Remote<{ secG: SecurityGroup }>(webuiKey);
1.6
1.7  const securityGroup = new SecurityGroup[7]('auth', {
1.8    vpc: reg.wishes.vpc, egress: [anywhereViaTcp] });
1.9  securityGroup.createIngressRule[7]('webui-inbound', {
1.10   location: { sourceSecurityGroupId: webui.wishes.secG.id },
1.11   ports: new TcpPorts(8080)});
1.12 [reg, persistence].foreach(remote =>
1.13   new Offer(remote, 'securityGroup', securityGroup));
1.14
1.15 const auth = new FargateService[7]('auth', {
1.16   cluster: reg.wishes.cluster,
1.17   subnets: reg.wishes.vpc.getSubnetsIds('private'),
1.18   taskDefinitionArgs: { container: {
1.23     environment: [
1.24       { name: 'REG_HOST', value: reg.wishes.svc.host },
1.25       { name: 'REG_PORT', value: `${reg.wishes.svc.port}`[8] }
1.26   ] } },
1.27   securityGroups: [securityGroup],
1.28 }, { dependsOn: [persistence.wishes.svc] });
1.29 new Offer(webui, 'service', undefined, { dependsOn: auth });
```

Listing 2: The Registry team's `svc` offer to the Auth team.

```
2.1  new Offer(auth, 'svc', { host: regHost, port: regPort },
2.2    { dependsOn: registry });
```

remote objects in the Auth team's deployment definition connect to Registry, Persistence, and WebUI.

The type parameter of a remote object defines wishes towards remote deployment definitions, mapping the names of the expected offers to their expected value types. For instance, Auth defines three wishes (`vpc`, `cluster`, and `svc` in Lines 1.1 and 1.2) from Registry (Line 1.3), an empty `svc` wish from Persistence (Line 1.4), and a security group `secG` from WebUI (Line 1.5).

Developers can access the wishes satisfied by an offer of a remote deployment via the `wishes` property of the remote object, which maps the wish 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 deployment definition. A wish's type may refer to a resource, e.g., in Line 1.8 `vpc` 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 `svc` offer (Lines 1.24 and 1.25). For other types, the wish resource provides the typed value as `value` or, in case of type `void`, has no output property, like for Persistence's `svc` offer (Line 1.28).

Offers to remote deployments are instances of `Offer`. They are configured with the beneficiary's remote object, a unique name among the offers to that remote deployment, and by the content to be offered. In Line 1.29, an empty offer depending on the Auth

[7] AWS resource interfaces presented as reused in our μS_ℓ implementation from Pulumi. Arguably better alternative typing would be possible.

[8] `` `${...}` `` used to convert `Output<number>` to expected `Input<string>`.

service is offered to WebUI as `service`. Lines 1.12 and 1.13 provide the security group as separate `securityGroup` offers to Registry and Persistence. Listing 2 shows the Registry team's `svc` offer of an object with host and port, depending on the Registry service.

4.4 Deployment Compatibility

A wish is *satisfiable* if it corresponds to an offer across separate deployment definitions. If a resource depends on an unsatisfiable wish, it is not deployed—like if it was not defined at all. As unsatisfiable wishes threaten availability, μIS allows checking the *compatibility* with connected deployments before the deployment is executed, i.e., whether all wishes are satisfiable by the connected deployments.

To check compatibility, μIS generates *offer excerpts* for each remote object in a deployment definition. An excerpt describes all offers for a particular remote with their types. These excerpts can then be used to validate the wishes in the remote deployment definition, checking that all wished offers exist and that their types are subtypes of the wished type.

5 DEPLOYMENT EXECUTION

A *deployment* is a process of the μIS *runtime* and is configured by a μIS$_\ell$ deployment definition. The deployment iteratively runs through three phases (Figure 7) as described in the following. Each deployment can be connected to arbitrary remote deployments, constituting a distributed system.

5.1 Configuration Phase

Each μIS$_\ell$ deployment definition is executed in the *interpreter* of its own μIS deployment runtime during the *configuration phase* (Figure 7a). The interpreter receives the deployment definition and the offers to the deployment as input from remote deployments, and generates the *target state*, i.e., the targeted deployment graph, plus the *resource configurations*. For each resource, the configuration comprises only simple values and output values of resources on which it directly depends. Upon deploying the resource, such output values are available because the dependency implies that the resources associated with the output values 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 deployment graph, ensuring that they are not in the target state and, thus, not deployed.

5.2 Deployment Phase

In the *deployment phase*, the *driver* updates the infrastructure based on the target state and the *current state* (Figure 7b). The current state is the deployment graph of the currently deployed resources. It also contains the resource configurations, but they are fully resolved to values—in contrast to the target state. The current state is initially empty, and the driver updates it according to the performed operations. It is saved in persistent storage from where it is read in consecutive runs to initialize the current state.

Operations on Resources. The driver implements CRUD operations (create, read, update, delete) for all supported resources. Reading a resource accesses its configuration from the infrastructure

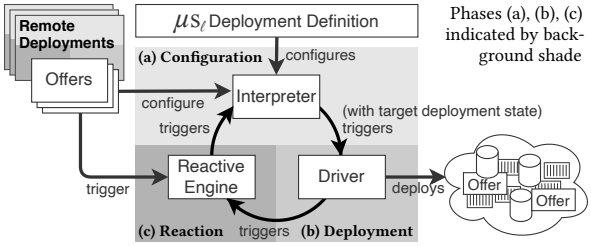


Figure 7: μIS deployment architecture.

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.

Deployment Algorithm. For deployment, 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 resource configuration as in the target state.

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

Wishes and Offers. 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 delete 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).

5.3 Reaction Phase

To enable DevOps, deployments should be started and updated independently, i.e., without (synchronous) coordination among teams. Thereby, it is critical to maintain all dependency constraints across deployments at all time to ensure correct operation. In μIS, the *reactive engine* of the deployment runtime triggers the interpreter and consecutively the driver whenever offers from other deployments change (Figure 7c). Thus, a μIS deployment is a long-running service continuously adapting the infrastructure rather than a one-off

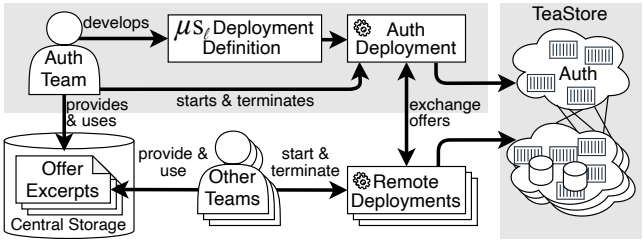


Figure 8: The Auth team's μS setup at TeaCorp.

task setting up the infrastructure. As a result, in μS, deployments are decoupled and can be started and updated independently.

The μS 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 deployment definition, 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.

5.4 Combining All Three Phases: μS in Action

We are now ready to present how μS can be used in a DevOps organization where deployments—similar to conventional application code—change over time. As programs and their infrastructure evolve, deployment definitions 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 deployment definition. Termination of a deployment does *not* undeploy its managed resources, but 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—it waits for the completion of the deployment phase. After restart, the deployment continues with the latest current state and the new deployment definition.

Figure 8 shows the Auth team's setup at TeaCorp. μS executes the Auth team's μS$_\ell$ deployment definition (cf. Listing 1) as a continuously running service that communicates with the deployments of the other teams to exchange the 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 any 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 deployment definition 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 did not update its offer excerpts yet. Otherwise, the Auth team's offer excerpts are generated from the new deployment definition and updated in the company-wide storage. Finally, the deployment is terminated and restarted with the new deployment definition.

6 DISTRIBUTION IN μS

In this section, we discuss aspects that do not impact μS' *principles* but are relevant in a real-world distributed system.

6.1 Availability

Connected μS deployments construct a distributed system where faults can occur anytime. μS has to account for another deployment's (temporary) unavailability because availability cannot be guaranteed. 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.

The safety protocol (cf. Section 6.3) ensures that resources are only deployed when their dependencies are—also across multiple deployments. 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.

6.2 Consistency

Several inconsistencies can arise in a distributed deployment system between the specification and the current state or among replicas.

Deployment Definition vs. Target State. The target state results from interpreting a μS$_\ell$ deployment definition (cf. Section 5.1). It depends on the environment, i.e., the offers from other deployments and possibly other external state and side effects since μS$_\ell$ supports all features of its host language. Thus, multiple executions of a μS$_\ell$ script may result in different target states, as the environment may change. Thus, the consistency of the target state with the deployment definition is only guaranteed at generation time.

Current State vs. Remote View on Offers. In μS, we ensure eventual consistency between each deployment's state and remote deployment's view on the deployed offers: The reactive engine triggers the re-execution whenever the environment changes. By default, μS considers changes of offers. If developers use other environmental elements in a deployment definition, e.g., external state, they have to inform the reactive engine whenever there is a change.

Current State vs. Infrastructure. Generally, the infrastructure might *drift* over time, i.e., become inconsistent with the current state, e.g., through external or manual administration operations. However, ideally, infrastructure managed purely by μS does not drift. Existing drift can be eliminated by reading the infrastructure's state into the μS deployment's current state and triggering the re-execution of the deployment phase (cf. Section 5.2).

6.3 Trust Model

Deployments can (necessarily) access information from or influence each other when there are inter-deployment dependencies.

Shared Information. Information between deployments is only shared via offers and wishes. An offer discloses a concretely defined

Table 1: Evaluation overview.

Research Hypothesis		Evaluation Method	Hypothesis Confirmed
RH1.1	μIS can automate decentralized deployments.	Implementing TeaCorp's TeaStore deployment from Section 3.	✓
RH1.2	μIS introduces only negligible coding overhead over its competitors.	Comparing SLOC of TeaStore deployments.	✓
RH2.1	μIS deployments are not slower than deployments with competitors.	Comparing duration of a standard deployment.	✓
RH2.2	μIS' deployment time is constant for independent dependencies.	Measuring joint duration for multiple independent deployments.	✓
RH2.3	μIS' deployment time scales linearly for sequential dependencies.	Measuring joint duration for a chain of dependent deployments.	✓
RH3.1	μIS is applicable to existing IaC programs.	Executing existing Pulumi TypeScript programs in μIS.	✓
RH3.2	Existing distributed IaC programs connected with explicit interfaces can be converted to μIS_ℓ.	Automatically converting 64 Pulumi TypeScript programs connected through stack references to μIS_ℓ.	✓

share of information to its beneficiary deployment defined as the offered object in the μIS_ℓ deployment definition. In the opposite direction, the beneficiary discloses the case in which no resources depend on the offer (at offer withdrawal, cf. Section 5.2).

Safety Protocol. As resources depending on a wish are deployed only 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.

7 IMPLEMENTATION

To the best of our knowledge, all existing IaC tools compatible with serverless infrastructure are executed as one-off tasks. Thus, μIS cannot be directly implemented using existing solutions. We implement μIS as a TypeScript library, which internally uses the Pulumi SDK [36]. μIS_ℓ deployment definitions are executed in μIS' runtime, which is also implemented in TypeScript and based on Hareactive [52], a functional reactive programming library to ensure continuous reactivity to deployment changes (Section 5.3).

We decided to implement the μIS framework as a minimal extension to Pulumi and base it on 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 remote connections, offers, and wishes. Pulumi provides μIS' interpreter and driver, enabling that all Pulumi IaC programs implemented in TypeScript are valid in μIS_ℓ and, thus, can be used with μIS out-of-the-box—without any changes. Our code is Apache 2.0 licensed and public on GitHub [48]. μIS_ℓ supports three resource types (`RemoteConnection`, `Offer` and `Wish`) implemented using dynamic resource providers [29]. `Remote` is a component resource to define a `RemoteConnection` and its `Wish` resources jointly (cf. Listing 1). In addition, all resource types available as a library from or for Pulumi can be used in μIS_ℓ.

The μIS runtime executes and deploys (cf. Section 5.1 and Section 5.2) a μIS_ℓ deployment definition using Pulumi's Automation API. 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 5.3). The reaction runtime ensures sequential execution of deployment rounds.

The resource implementations communicate via gRPC with their μIS runtime to update and retrieve the state of offers, wishes, and remote connections. μIS deployments also use gRPC for connections between them, defined by `RemoteConnection` resources. Between consecutive runs, the μIS 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.

8 EVALUATION

In this section, we evaluate the design and the implementation of μIS. First, we are interested in whether μIS is effective in ensuring safe, decentralized deployments for service-based applications. This leads to the research question:

RQ1: Does μIS effectively support deployment automation in DevOps organizations? It is crucial to assess whether μIS can automate deployment in a context where current solutions need manual coordination. Second, we are interested in the run time performance of μIS, leading to the research question:

RQ2: How does μIS' performance compare to state-of-the-art, industrial-strength deployment solutions? Performance is important to ensure that μIS' automation does not come at the detriment of slow deployments. Finally, we are interested in the applicability of μIS to existing projects. We ask:

RQ3: Can μIS be applied to existing IaC projects? It is important to ensure that μIS can be applied to real-world IaC projects and assess the required migration effort.

We break the research questions down into research hypotheses (RH) in Table 1 and state the evaluation method to confirm each of them. For the experiments, we use the Amazon Web Services cloud with AWS Fargate containers [6]. We select Pulumi and AWS CDK as a baseline because they are industrial-strength, recent IaC solutions, and offer features comparable to μIS (Section 10.4).

8.1 Effective Deployments in DevOps

To answer RQ1, we implement three versions of the TeaStore application's deployment (Section 3), with μIS, with Pulumi, and with AWS CDK. We compare automation and definition overhead.

First, we consider the support for decentralized deployments. With Pulumi, AWS CDK, and μIS, each team can have a separate

Table 2: Size of the teams' deployments at TeaCorp (SLOC).

Team	Auth	Image	Pers.	Recomm.	Registry	WebUI	Total
μS$_\ell$	61	63	88	63	75	144	494
Pulumi	53	56	80	56	59	129	433
CDK	47	48	91	47	59	73	365

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 (cf. Section 3), confirming RH1.1.

Second, to evaluate the coding overhead required by μS (RH1.2), we compare the size for each IaC solution. Table 2 reports the SLOC for each team's service 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 [5] require less configuration than the best practices implemented in Pulumi's Crosswalk for AWS library [35].

In summary, to answer RQ1, 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.

8.2 Performance

To answer RQ2, we first measure the duration of a containerized HTTP web service deployment with μS and compare it to AWS CDK and Pulumi. Second, we assess the run time of the automated coordination of multiple depending deployments in a parallel and in a sequential setup. For the experiments, we deploy AWS Fargate container services with the web service instance [22] into an existing VPC and cluster.

In the first experiment, we assess the performance of a deployment with μS compared to AWS CDK and Pulumi. We repeat the measurements of deployment duration 15 times. Each measurement starts at process start and ends for μS 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 9 and indicate that Pulumi and μS deployments take similarly long, while deployment with AWS CDK takes on average 20% longer. Hence, μS is not slower than AWS CDK nor Pulumi, confirming RH2.1.

In the second experiment, we assess the performance of μS' automated coordination. In the parallel setup, all deployments depend on the same *lead deployment* and can be deployed in parallel once the lead deployment is available. In the sequential setup, the deployments' dependencies build a chain towards the lead deployment. Hence, deployments take place sequentially after the lead deployment. We measure both setups with 3, 6, and 12 services and repeat each experiment 3 times. Figure 10 shows the number of deployed resources after starting (un)deployment of the lead deployment over time. As expected by RH2.3, in the sequential

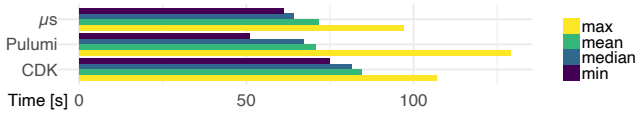


Figure 9: Required time to deploy a single service.

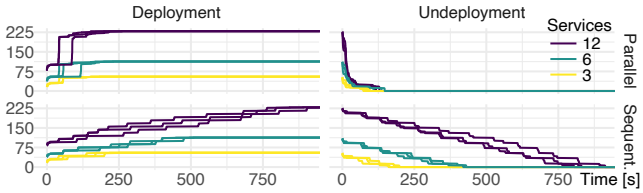


Figure 10: Number of resources deployed with μS when (un)deploying services in parallel and sequentially.

setup, the time increases linearly with the number of services, i.e., three need ~3.5 minutes and 12 services 4× as much (~14 minutes). The parallel setup (RH2.2) requires, independently of the number of services, roughly double the time (for the lead deployment and the deployments that depend on it), ~2.5 minutes, compared to the single service experiment in Figure 9. Deployment and undeployment show the same behavior; undeployment is faster. The results show no significant overhead of automated coordination, entailing the behavior expected in RH2.2 and RH2.3.

The experiments answer RQ2, showing that deployment duration is comparable or better than with state-of-the-art IaC systems, and μS' automated coordination does not introduce significant delay.

8.3 Applicability

To answer RQ3, we applied μS to third-party open-source projects. First, by the design of our system, every Pulumi TypeScript program is a valid μS$_\ell$ deployment definition, making μS virtually compatible with any third-party, Pulumi-based TypeScript project, satisfying RH3.1.Of course, when starting from a centralized Pulumi script, the correct splitting into separate deployments, one for each DevOps team, requires manual intervention because the DevOps structure of the organization is not explicit in the code. Hence, naively porting a Pulumi program to μS does not benefit from μS' automated coordination out-of-the-box.

To demonstrate μS' deployment automation, we focus on a subset of Pulumi programs already partitioned and used in a decentralized way. These Pulumi programs use *stack references* [37] to access remote deployments, making the boundary between deployments and their interfaces explicit. Such scripts can be migrated to support automated deployment by (1) defining offers for the supplied resources and (2) wishes for accessing remote resources by replacing Pulumi's stack references. We built a dataset [46] 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 μS$_\ell$ by applying a simple script [47] that translates stack references and their access to remote, wish and offer resources based on AST transformation. Our migration of the dataset replaces 197 stack references with 556

wishes and shows that that RH3.2 applies: μS decentralized coordination can automatically be leveraged if distributed deployments are connected through explicit interfaces, e.g., stack references.

The experiments answer RQ3, showing that μS_ℓ can be applied to 64 real-world projects. While μS is compatible with any existing Pulumi TypeScript program, we further show 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.

9 LIMITATIONS

While this work focuses on serverless computing, μS is not limited to serverless resources. We only require that a resource is controllable via a CRUD API. This is naturally the case in serverless computing but nothing prevents that a similar approach is 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, which includes, for example, virtual servers (e.g., AWS EC2 [3]). 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.

Even if μS decouples the operations of teams, offers and wishes cause *architectural* coupling because they must be compatible for dependency satisfaction, requiring to exchange the interfaces during development. However, this communication is not critical for operations because it can be performed asynchronously with no impact on the running system. Further, μS offers an asynchronous verification mechanism for compatibility (cf. Section 4.4). Currently, μS focuses on 1-to-1 offer–wish dependencies, where the offering and the wishing deployment directly reference each other. To increase decoupling, we could consider offers to and wishes from *any* deployment. This mechanism would require a middleware (e.g., a publish/subscribe system) to mediate indirect dependencies.

Another important aspect is that μS treats deployed applications as a black box. Thus, deployment updates do not consider the *internal state* of deployed applications. In contrast, techniques like *safe dynamic updating* [11, 51] enable only safe resource updates, i.e., no distributed transaction occurs across a component update. Such mechanism can be implemented on top of μS.

Concerning performance, as shown in Section 8.2, μS' results are comparable to competitors. Still, deployment time is dominated by the infrastructure platforms managing the deployed resources, causing deployment iterations to take at least seconds and rather minutes. This process might be too slow for adaptive systems where deployment changes must be applied very frequently. μS' approach is applicable to such scenarios in combination with a faster resource orchestration and a lower-latency driver than Pulumi.

Finally, μS' distributed behaviors (cf. Section 6) necessarily require a trust relationship between the teams who connect their deployments with offers and wishes. Also, as changes require that the deployment and the resource orchestration platform are available, μS does not fit scenarios where either is regularly unreachable.

10 RELATED WORK

We now outline the research gap covered by μS before providing detailed insight into the related work.

Resource orchestrators (Section 10.1) manage resource creation, setup, and deletion. In contrast to μS, they do not coordinate deployments over time, nor do they ensure correct dependencies across resources. μS can interface with resource orchestrators and leverage them for the actual infrastructure changes.

Modeling languages (Section 10.2) describe the system architecture *and* its operational behavior but do not allow the automatic derivation of operations, as μS does. Architecture description languages (ADLs) (Section 10.3) specify software architectures and can be used to check architectural conformance. μS is similar, as it provides a model of the components' dependencies, but the goal of μS is to define an executable specification, ensuring that deployment dependencies are fulfilled even in a decentralized setting.

μS features its own definition language μS_ℓ, which is executable code, in contrast to modeling languages or ADLs. Still, decentralized coordination could also be defined in modeling languages like TOSCA (cf. Section 10.2). In particular, μS_ℓ code can be generated from definitions in modeling languages (cf. Section 10.3). Similarly, μS_ℓ' coordination based on offers and wishes could be defined as an extension to an ADL (instead of extending Pulumi). These approaches would relinquish some advantages of our solution that combines Pulumi and TypeScript (cf. Section 7) but would enable reuse of existing analysis and architecture verification techniques.

Existing solutions for IaC (Section 10.4) are centralized, and each team contributes its part. Instead, μS is decentralized, preserving compatibility with the separation into DevOps teams.

10.1 Resource Orchestrators

For an overview of cloud resource orchestration, the reader may refer to the survey by Weerasiri et al. [54]. According to their reference architecture, μS combines a policy enforcement engine and a rule engine. The former derives decisions based on policies and monitoring; the latter performs deployment operations for each decision based on defined rules. A survey of programming resource orchestration operations is in the work of Ranjan et al. [42].

Solutions like Kubernetes, Kubernetes Federation, Mesos, and Docker Swarm manage software containers [14], an OS-level virtualization technique for simplified, standardized software distribution with guaranteed isolation. Container orchestrators deploy containers across clusters and data centers and provide features for fault tolerance, load balancing, and automated scaling. In contrast to μS, all systems above only support central configuration and are limited to the management of container-based applications.

Motivated by the challenges in IoT, where applications and services are deployed not only in data centers but also at the edge, DOCMA [25] is a distributed and decentralized orchestrator for containerized microservice applications. However, DOCMA applications are centralized: an application globally defines everything (all services/containers) within its scope. In contrast to μS, separation into multiple entities and dependencies among applications are not supported. Liu et al. [28] present COPE (Cloud Orchestration Policy Engine), a distributed platform to automate cloud resource orchestration. COPE is declarative: the provider specifies

constraints and goals with a policy language that, in combination with the current system state, is used to determine the compute, storage and network resource allocations to meet customer SLAs.

10.2 Modeling Languages

Modeling languages express a system structure (e.g., nodes and relations) following a consistent set of codified rules. TOSCA [34] is an OASIS standard for modeling cloud applications and their management. The application topology is described as a graph of *nodes* (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 [12] provide a survey on cloud orchestration methodologies using TOSCA, its language extensions, and tools for manipulating TOSCA models. Wettinger et al. [55, 56] apply TOSCA to DevOps using it as a metamodel to integrate heterogeneous automation artifacts. The *Essential Deployment Metamodel* (EDMM) [58] is the least denominator metamodel of popular declarative deployment technologies to ensure that metamodel instances easily map to such technologies. *TOSCA Light* [59] is an EDMM-compliant subset of TOSCA, whose models can be deployed with 13 popular deployment technologies using the *TOSCA Lightning* [57] toolchain.

μ*I*S draws direct inspiration from the EDMM, describing deployments as a graph of typed components and directed relations, where artifacts are defined as component properties and operations are derived from the deployment description. We envision a clear synergy between μ*I*S and modeling languages. For example, a specification in μ*I*S could be derived from TOSCA with a two level-approach: The TOSCA specification provides a centralized view on a system with components implementing well-defined interfaces, and our system provides the operationalization and the runtime to execute such specification as a decentralized deployment. The TOSCA specification serves a whole-system view for the design phase, while DevOps teams use μ*I*S for decentralized deployment definitions.

10.3 Architecture Description Languages

Architecture Description Languages (ADL) describe the high-level structure of applications on a component level. According to the classification of Medvidovic and Taylor [30], ADLs must explicitly model components, their connection with the respective configurations, and they require tools for development and evolution.

ArchJava [2] 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*, *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 [26] treat services as first-class composition units and separates the application from infrastructure concerns. It features a sub-language to define the deployment and to allow dynamic system changes. Terra and Valente [49, 50] propose 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.

10.4 Infrastructure as Code

Infrastructure as code [32] refers to the management and provisioning of computing resources through machine-readable code to enable deployment automation. The advantage of IaC is that designing, implementing, and deploying infrastructure can leverage known software best practices such as version control and code reviews. Various industrial frameworks support IaC, including CloudFormation, CloudFormation-based DSLs and Orchestrators, AWS CDK, ARM, Terraform and Pulumi. They configure and provision infrastructure and software and differ in the language they support (e.g., custom DSLs vs. existing language like JavaScript), programming model, and the targeted infrastructure [40].

Balis et al. [10] investigate an approach based on Terraform to provide repeatable cloud infrastructures for scientific computing to automate research experiments in scientific workflows. They enable a strict separation of infrastructure provisioning and workflow description and support auto-scaling of scientific workflows. Guerriero et al. [21] conduct semistructured interviews with senior developers to investigate the state of IaC adoption, concluding that available tools offer limited automation, lack portability (each tool is based on a different language) and miss support for analysis techniques (e.g., linters). TOSCA supports IaC in DevOps by providing a standard notation and language for infrastructure [8].

Similar to traditional code, IaC can be of poor quality. Sharma et al. [44] propose a catalog of configuration smells that violate best practices for IaC, analyze ~ 5 K Puppet repositories and show that design smells and configuration smells tend to occur together. Schwarz et al. [43] extend this research to show that IaC smells are agnostic to the specific technology and can be defined at a more abstract level. Rahman and Williams [41] conduct an empirical study to identify the characteristics of defective IaC scripts through a qualitative analysis of project commits validated by an assessment from practitioners. They identify several properties that correlate with defects (e.g., executing external modules and hard-coded strings).

11 CONCLUSION

Our study on 73 IT professionals confirms that most applications depend on others and that such dependencies constrain the order of the applications' deployment, for which typically manual coordination is required—even though developers agree that automation promises better SDO performance. Yet, today's IaC solutions cannot automate deployments in DevOps organizations because they require centralization or manual out-of-band coordination, e.g., via phone, chat, or email. We propose μ*I*S, an IaC system to automate deployment coordination in a *decentralized* fashion, ensuring compatibility with the DevOps practice. We implement μ*I*S and show that it ensures decentralized deployments without manual coordination, introduces negligible performance overhead, and is broadly applicable to IaC projects.

ACKNOWLEDGMENTS

This work has been co-funded 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, 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] Amazon Web Services. 2021. Amazon EC2. https://aws.amazon.com/ec2/, last accessed on 2021-05-28.

[4] Amazon Web Services. 2021. Amazon RDS: Cloud Relational Database. https://aws.amazon.com/rds/, last accessed on 2021-02-14.

[5] Amazon Web Services. 2021. AWS Cloud Development Kit (AWS CDK): Constructs. https://docs.aws.amazon.com/cdk/latest/guide/constructs.html, last accessed: 2021-02-26.

[6] Amazon Web Services. 2021. AWS Fargate: Serverless Compute Engine. https://aws.amazon.com/fargate/, last accessed on 2021-02-14.

[7] Amazon Web Services. 2021. Serverless Computing. https://aws.amazon.com/serverless/, last accessed on 2021-05-21.

[8] Matej Artac, Tadej Borovssak, Elisabetta Di Nitto, Michele Guerriero, and Damian Andrew Tamburri. 2017. DevOps: Introducing Infrastructure-as-Code. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 497–498. https://doi.org/10.1109/ICSE-C.2017.162

[9] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. 2017. *Serverless Computing: Current Trends and Open Problems*. Springer Singapore, Singapore, 1–20. https://doi.org/10.1007/978-981-10-5026-8_1

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

[11] 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

[12] 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

[13] Alanna Brown, Nigel Kersten, and Michael Stahnke. 2020. 2020 State of DevOps Report. https://puppet.com/resources/report/2020-state-of-devops-report/, last accessed on 2020-11-27.

[14] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 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

[15] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2019. The Rise of Serverless Computing. *Commun. ACM* 62, 12 (Nov. 2019), 44–54. https://doi.org/10.1145/3368454

[16] Daniel Cukier. 2013. DevOps Patterns to Scale Web Applications Using Cloud Services. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity* (Indianapolis, Indiana, USA) *(SPLASH '13)*. Association for Computing Machinery, New York, NY, USA, 143–152. https://doi.org/10.1145/2508075.2508432

[17] Digital.ai. 2020. 14th Annual State of Agile Report. https://explore.digital.ai/state-of-agile/14th-annual-state-of-agile-report, last accessed on 2020-11-30.

[18] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. *Microservices: Yesterday, Today, and Tomorrow*. Springer International Publishing, Cham, 195–216. https://doi.org/10.1007/978-3-319-67425-4_12

[19] Nicole Forsgren, Dustin Smith, Jez Humble, and Jessie Frazelle. 2019. *2019 Accelerate State of DevOps Report*. Technical Report. https://services.google.com/fh/files/misc/state-of-devops-2019.pdf, last accessed on 2021-05-05.

[20] Google Cloud. 2021. Serverless Computing. https://cloud.google.com/serverless, last accessed on 2021-05-21.

[21] 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

[22] HashiCorp. 2016. In-memory web server that echos back the arguments given to it. https://hub.docker.com/r/hashicorp/http-echo/, last accessed: 2021-02-25.

[23] Douglas D. Heckathorn. 1997. Respondent-Driven Sampling: A New Approach to the Study of Hidden Populations. *Social Problems* 44, 2 (1997), 174–199. https://doi.org/10.2307/3096941

[24] IDC and Statista. 2019. Worldwide Technology Employment Impact Guide. IDC; Statista. https://www.statista.com/statistics/1126677/it-employment-worldwide/, last accessed on 2021-02-03.

[25] Lara Lorna Jiménez and Olov 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

[26] 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

[27] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. 2019. A Survey of DevOps Concepts and Challenges. *ACM Comput. Surv.* 52, 6, Article 127 (Nov. 2019), 35 pages. https://doi.org/10.1145/3359981

[28] 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

[29] Praneet Loke. 2020. Pulumi: Dynamic Providers. https://www.pulumi.com/blog/dynamic-providers/, last accessed: 2021-02-22.

[30] 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

[31] Microsoft Azure. 2021. Serverless Computing. https://azure.microsoft.com/en-us/overview/serverless-computing/, last accessed on 2021-05-21.

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

[33] 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

[34] 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.

[35] Pulumi. 2021. Pulumi Crosswalk for AWS. https://www.pulumi.com/docs/guides/crosswalk/aws/, last accessed: 2021-02-26.

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

[37] Pulumi. 2021. Stacks: Stack References. https://www.pulumi.com/docs/intro/concepts/stack/#stackreferences, last accessed: 2021-02-23.

[38] Puppet. 2019. Ambit Energy's Competitive Advantage? It's Really a DevOps Software Company. https://media.webteam.puppet.com/uploads/2019/11/puppet-cs-ambit.pdf, last accessed on 2021-02-22.

[39] Puppet. 2019. NYSE and ICE: Compliance, DevOps and Efficient Growth with Pupper Enterprise. https://media.webteam.puppet.com/uploads/2019/11/puppet-CS-NYSE-ICE.pdf, last accessed on 2021-02-22.

[40] 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

[41] 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

[42] 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

[43] 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

[44] 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

[45] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. 2021. *Dependencies in DevOps Survey 2021*. https://doi.org/10.5281/zenodo.4873909

[46] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. 2021. *Pulumi TypeScript Projects using Stack References*. https://doi.org/10.5281/zenodo.4878577

[47] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. 2021. *Pulumi TypeScript Stack References to µs Converter*. https://doi.org/10.5281/zenodo.4902171

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

[49] 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

[50] 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

[51] 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

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

[53] Jóakim 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

[54] 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

[55] 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

[56] Johannes Wettinger, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. 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

[57] Michael Wurster, Uwe Breitenbücher, Lukas Harzenetter, Frank Leymann, and Jacopo Soldani. 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

[58] 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

[59] 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