Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry

Michele Guerriero,[1] Martin Garriga,[2] Damian A. Tamburri,[3] Fabio Palomba[4]
[1]Politecnico di Milano, Italy
[2]Jheronimus Academy of Data Science & Tilburg University, The Netherlands
[3]Jheronimus Academy of Data Science & Eindhoven University of Technology, The Netherlands
[4]University of Zurich, Switzerland
michele.guerriero@polimi.it, m.garriga@uvt.nl, d.a.tamburri@tue.nl, palomba@ifi.uzh.ch

Abstract—**Infrastructure-as-code (IaC) is the DevOps tactic of managing and provisioning infrastructure through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools. From a maintenance and evolution perspective, the topic has piqued the interest of practitioners and academics alike, given the relative scarcity of supporting patterns, best practices, tools, and software engineering techniques. Using the data coming from 44 semi-structured interviews in as many companies, in this paper we shed light on the state of the practice in the adoption of IaC and the key software engineering challenges in the field. Particularly, we investigate (i) how practitioners adopt and develop IaC, (ii) which support is currently available, i.e., the typically used tools and their advantages/disadvantages, and (iii) what are the practitioner's needs when dealing with IaC development, maintenance, and evolution. Our findings clearly highlight the need for more research in the field: the support provided by currently available tools is still limited, and developers feel the need of novel techniques for testing and maintaining IaC code.**

Index Terms—**Infrastructure-as-Code; DevOps; Software Maintenance & Evolution; Cloud Automation;**

I. Introduction

The current information technology (IT) market is increasingly focused towards the "need for speed": speed in deployment, faster release-cycles, speed in recovery, and more. This need is reflected in DevOps, a family of techniques which shorten the software development cycle and also intermix software development activities with IT operations [1], [2]. As part of the DevOps family of practices, *infrastructure-as-code* (IaC) [3] promotes managing the knowledge and experience inside reusable scripts of infrastructure code, instead of traditionally reserving it for the manual-intensive labour of system administrators which is typically slow, time-consuming, effort-prone, and often even error-prone.

While IaC represents an ever increasing widely adopted practice nowadays [2]–[4], little is known concerning how to best maintain, speedily evolve, and continuously improve the code behind the IaC strategy and yet it is picking up more and more traction in most if not all domains of society and industry: from Network-Function Virtualization (NFV) [5] to Software-Defined Everything [6] and more [7].

This paper targets at addressing that gap with empirical software engineering inquiry to aid and better focus the work of practitioners and academicians in the area. We conduct 44 semi-structured interviews in as many companies to distill: (1) how practitioners currently develop infrastructural code — that is, the best/bad practices experienced by the practitioners as IaC blueprints grow in size and maintenance costs [8], (2) what is the automatic support available — that is, pros and cons of using existing tools in practice, and (3) the challenges reported by the practitioners via direct experience — that is, the research and industrial avenues that practitioners perceive as worthy of investigation in the near future.

Data on the above points reveals a number of results as well as valuable avenues for further work. More specifically, our data reveals that 8-10 tools constitute equally-used alternatives in the DevOps IaC technical space, and they include Docker, Ansible, Vagrant, Kubernetes, and more. Also, the practitioners do not agree concerning the major *pro's* on these tools and their best usage are sparse while, on the *con's* front, practitioners identify testability, readability, consistency, and portability as major technical challenges which still need much attention from the state of the art and practice. Finally, although several best practices emerged from practitioners' insights (e.g., infrastructure programming to *break fast*, to speedily reveal broken code), they differ in applicability per technology.

The impact of our findings is considerable. First, software practitioners can use these to understand the best practices used by their peers when developing infrastructural code as well as the reported wrong design choices, thus possibly improving or refining their practice. Second, academicians in the field of software maintenance and evolution can better focus their research efforts towards the needs of industrial practitioners focusing on the evidence in this paper. Last but not least, for the sake of replicability, we make available our entire dataset such that others may replicate or further analyse our results for the benefit of theory and practice alike.

Structure of the paper. The remaining of this paper is structured as follows. Section II sets the background for our study. In Section III we outline our research design, while Section IV presents the results of our semi-structured interviews, while Section V discusses our findings and the implications for both industry and research. Section VI discusses the possible threats affecting the validity of our results. Section VII outlines related work on infrastructure code maintenance and evolution.

Finally, Section VIII concludes the paper.

II. Background

The DevOps methodology is radically changing the way software is designed and managed nowadays. DevOps entails the adoption of a set of organizational and technical practices, e.g. Continuous Integration (CI), Continuous Deployment (CD), blending development and operation teams. The goal is, essentially, to be able to survive as an organization in the modern digital ecosystem and digital market, which demands for fast and early releases, continuous software updates, constant evolution of market needs, and adoption of scalable technologies such as Cloud computing.

In this context, Infrastructure-as-Code (IaC) is the DevOps practice of describing complex and (usually) Cloud-based deployments by means of machine-readable code. The main enabler for IaC has been the advent of Cloud computing, which, thanks to virtualization technologies, for the first time allowed the provisioning, configuration and management of computational resources to be performed programmatically.

Subsequently, many different languages and corresponding platforms have been developed, each of which deals with a specific aspect of infrastructure management. From tools able to provision and orchestrate virtual machines (Cloudify, Terraform, etc.), to those doing a similar job with respect to container technologies (Docker Swarm, Kubernetes), to machine image management tools (Packer), to configuration management tools (Chef, Ansible, Puppet, etc.). Just to give an example, Listing 1 shows a piece of Kubernetes code, which provisions and deploys a Couchbase database. We can notice how we can configure various aspects such as the ports to be opened on the host container, the container image to be used or the desired number of replicas of the database.

Figure 1: An example of Kubernetes code.

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: couchbase-rc
  labels:
    name: couchbase-rc
    context: iac-example
spec:
  replicas: 1
  template:
    metadata:
      name: couchbase-rc-pod
      labels:
        name: couchbase-rc-pod
        context: iac-example
    spec:
      containers:
      - name: couchbase-rc-pod
        image: devs/iac-example:latest
        ports:
        - containerPort: 8091
        - containerPort: 8092
        - containerPort: 8093
        - containerPort: 11210
```

Currently, the landscape of IaC languages and tools is jeopardized by the technology heterogeneity and by the huge number of available solutions. On the one hand this is the result of the great interest that IaC has raised. On the other hand, it complicates the understanding and adoption of this new technology. Shedding light on the IaC current adoption, issues and challenges is thus fundamental towards bringing IaC to maturity and ease its further development.

III. Research Methodology

The *goal* of the study is to empirically investigate the state-of-the-practice in the development of infrastructural code, with the *purpose* of eliciting the developer's perspective when it turns to (i) current development practices, (ii) pros and cons of available tools, and (iii) challenges when developing infrastructural code. The *perspective* is of both practitioners and researchers: the former are interested in gaining a wider knowledge of how the general population of developers develop infrastructural code, while the latter are interested in assessing the limitations of the current state-of-the-practice and sketch the developer's needs that have to be further addressed by the research community.

A. Research Questions

The specific research questions driving our investigation are the following:

- **RQ$_1$.** *How do practitioners currently develop infrastructural code?*
- **RQ$_2$.** *How do currently available tools support practitioners when developing infrastructural code?*
- **RQ$_3$.** *What are the challenges that practitioners face when developing infrastructural code?*

With the first research question (**RQ$_1$**), we aim at understanding how practitioners develop infrastructural code in terms of good and bad development practices they follow and encounter, respectively. Then, in **RQ$_2$** we focus on the (semi-)automatic support available for practitioners when developing infrastructural code. Finally, with **RQ$_3$** we focus on the main issues and challenges practitioners perceive during IaC development.

Addressing these three specific angles, we aim at providing a broad view of the state-of-the-practice and future challenges that the research community should pursue to help practitioners in the development of high-quality infrastructural code. In the following sections, we describe methodological steps and data analysis methods used to address our research questions.

B. Subjects of the Study

To address our **RQ**s, we need to select subjects able to provide us with an authoritative opinion with respect to IaC practices, tools, and challenges. To this aim, the four authors of this paper first internally discussed and selected a set of candidate companies to involve in the study: this was done by considering (1) the list of companies that, in the last five years, have been granted with European projects related to the development of IaC technologies (*e.g.,* the DICE project[1]) and (2) personal contacts of the authors. This step led to the identification of 86 companies which are supposed to be active

[1]http://www.dice-h2020.eu

Table I: Schema of the semi-structured interviews conducted on IaC development practices, tools, and challenges.

Question ID	Question
Background	
Q1	How many years of working experience do you have in DevOps?
Q2	Do you actively develop Infrastructure-as-Code?
Q3	How many hours per week do you work on developing IaC?
Q4	Do you also develop standard (non-IaC) software?
Theme: IaC Development Practices	
Q5	Describe the main differences between writing standard code and IaC.
Q6	Do you have anything you consider a bad practice in IaC development? Please specify.
Q7	Do you have any good practice to develop IaC? Please specify.
Theme: IaC Development Tooling	
Q8	When you develop IaC, which specific software development tools do you use? (e.g. IDEs, static analyzers, debuggers, testing frameworks, documentation frameworks, etc.)?
Q9	Which DevOps tool(s) are you experienced with?
Q10	For each tool you are experienced with, could you name some pros and cons in terms of the programming language for writing IaC it provides?
Theme: IaC Development Challenges	
Q11	Could you list from 2 to 5 common issues that you currently face when you develop IaC?
Q12	How much desirable are IaC-specific Integrated Development Environments (IDEs)?
Q13	How much desirable are IaC-specific Integrated testing frameworks?
Q14	How much desirable is IaC language standardization?
Q15	How much desirable are IaC-specific static analysis tools?
Q16	How much desirable are tools and methods to support IaC security and privacy related issues?
Q17	How much desirable is to have IaC support for heterogeneous infrastructures (GPUs, FPGAs, Cloud, HPC)?
Q18	Could you mention any other challenge and future direction in the context of IaC development?

in the development of infrastructure code and, perhaps more importantly, are medium to large. In the second place, we sent a study participation invitation to the senior developers of those companies, who were identified through the company websites. We got response from 44 of them, who confirmed their willingness to conduct a semi-structured interview on IaC themes. In case they had no experience at all with infrastructural code, we excluded and asked them to indicate another person of the company who was willing to conduct the interview.

C. Gathering the Practitioner's Perspective

To acquire knowledge on the state-of-the-practice in the development of infrastructural code, we conducted semi-structured interviews, a form of interactive discussion often used in exploratory investigations to understand phenomena and seek new insights [9]. The general structure of the interviews is presented in Table I; as shown, after some background questions aimed at characterizing the sample of practitioners interviewed, we organized the semi-structured interview around three main themes, one for each research question.

The first theme, *i.e., 'IaC Development Practices'*, revolves around the main differences observed by practitioners when developing infrastructural code with respect to standard source code and the best/bad practices they could recognize in IaC development. Through this set of questions, we aimed at shedding lights on the ways practitioners develop infrastructural code as well as the key elements making IaC different from standard code. With the second theme, *i.e., 'IaC Development Tooling'*, we instead focused on the tools currently available for writing infrastructural code and their main advantages and disadvantages from the practitioner's perspective: this angle had the goal of discovering potential limitations that tool vendors and researchers could address in the future. Finally, the last theme, *i.e., 'IaC Development Challenges'*, is the one where we aimed at extracting a set of issues and challenges in the development of infrastructural code. As shown, we inquired practitioners on a broad set of software maintenance and evolution perspectives, ranging from language standardization to testing- and security-related matters. All these aspects are of interest for both tool vendors and researchers, called to improve the automatic support provided to practitioners.

The 44 interviews were conducted by two authors of this paper via SKYPE. They were all recorded and transcribed for analysis, and took $\approx$1 hour per practitioner.

D. Data Analysis

Once concluded all the semi-structured interviews, we analyzed the transcripts by means of content analysis [10]. Specifically, the four authors of the paper first independently went through each transcript and assigned preliminary codes to all relevant pieces of information. In a second step, they opened a joint discussion where they discussed the codes assigned so far and came up with a final set of pieces of information that we discuss in the following section.

IV. Analysis of the Results

In this section, we analyze the results of our study. For the sake of comprehensibility, the analysis that follows is structured according to the structure of the semi-structured interviews defined in Table I. In particular, Section IV-A presents the demographic information of the practitioners involved. Following, Section IV-B addresses $\textbf{RQ}_1$: IaC Development Practices. Then,

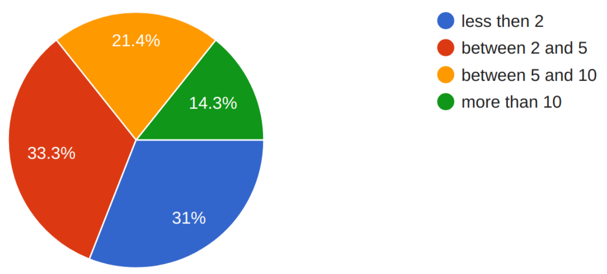

Figure 2: Distribution of the IaC development experience of interviewees (**Q1**).

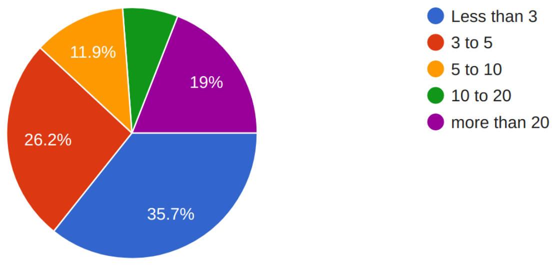

Figure 3: Distribution of the number of hours per week spent developing IaC (**Q2**).

Section IV-C addresses **RQ**$_2$: IaC Tooling; and Section IV-D discusses **RQ**$_3$: IaC challenges and directions.

A. Background and Demographics

First, we asked questions regarding activity and experience in developing IaC. Figure 2 reports the results for experience (**Q1**), where 64.3% of industry practitioners have less than 5 years of experience developing IaC, whilst only 14.3% have more than 10 years of experience. This somehow confirms that IaC is a relatively new trend, and practitioners are now acquiring the required expertise to develop IaC. Regarding activity (**Q2**), 88.4% of practitioners actively develop IaC, which gives us confidence on the answers given to subsequent questions, i.e., the majority of our interviewees are *actual* IaC developers.

We then wanted to understand whether IaC developers are specialized personnel or actually common developers or operators which, as part of their job, do write IaC (**Q3**) and standard production code (**Q4**) alike. Figure 3 shows that 61.9% of respondents spend less than 5 hours per week writing IaC, which can be regarded as a quite small amount of time. Indeed, assuming a developer works on average 40 hours per week, less than 12.5% of its time is devoted to IaC development. On the other hand, only 19% of respondents spend more than 20 hours per week in IaC development.

These results suggest IaC developers are pretty versatile, as it turned out that 88.1% of respondents do also develop standard code (**Q4**). This is also in line with the belief that IaC is a DevOps practice; with DevOps engineers being highly versatile workers which deal not only with infrastructures and applications operations, but also with standard application development as well, covering the entire application lifecycle.

*B. **RQ**$_1$: IaC Development Practices*

When addressing **RQ**$_1$, we firstly aimed at understanding the particularities of writing IaC code with respect to standard production code (**Q5**). Table II depicts the main differences pointed out by interviewees, ordered by frequency. Interestingly, IaC turns out to be, in general, declarative and Tree-alike vs. the imperative and Graph-alike features of standard code. As an example, one of our interviewees explained that:

> *"Product development is more akin to developing a 'graph' of modules while IaC is a tree of nodes".*

Table II: Differences between IaC code and standard code (**Q5**), ordered by the number of mentions (#) among all answers.

Diff	**Description**	**#**
Impossible Testing	The lack of standard practices for testing and proper (maybe local) testing environments makes testing painful	8
Declarative	Standard programming is in terms of class, functions, flow. With IaC the reasoning is declarative, i.e., express what is needed, not how to do it.	7
Graph vs. Tree model	Production code is shaped as a graph of modules whereas IaC code resembles a tree of nodes.	7
Impossible Debugging	Similar to testing, the lack of standard practices and the distributed nature makes debugging painful.	6
IaC error-prone	The tools for code checking are more shallow for IaC (e.g., lack of type-checking).	5
Longer feedback loop	The developer has to wait for the whole infrastructure to be deployed in order to understand the correctness of the IaC code.	3
Unmaintainable	As the infrastructure evolves and the computer resources changes, IaC code cannot cope with those on a seamless way.	2

This substantially changes the way in which practitioners approach and design IaC code and blueprints, and circumscribes the practices and patterns that they can migrate from their experience with standard code. The other differences mentioned were mostly in the form of IaC disadvantages, mainly regarding testing and debugging. As such, more discussion on this is presented in Section IV-D.

Following, we collected what are perceived as bad practices when developing IaC (question **Q6**). Table III summarizes the answers for those bad practices that were mentioned more than once in the answers. The most common one is *hardcoding* which not only hinders maintainability (e.g., by not using environment variables) but also can cause critical security breaches, e.g., by embedding passwords, ssh keys or access codes on the blueprints.

This was followed by too Polyglot IaC, which means blueprints in various languages for the same environment. This goes against understandability and maintainability of the systems. For instance, one practitioner explained that:

> *"Code which is too polyglot is unreadable, sloppy to deploy, often slow, and difficult to debug".*

Table III: Bad practices when Developing IaC (**Q6**) ordered by number of mentions.

Bad-practice	**Description/Effects**	**#**
Hardcoding	Hardcoding values on the script such as credential or constants.	5
Too Polyglot	Using many languages in interrelated node definitions.	4
Blob blueprints	Generating too large scripts.	3
Non idempotent code	Writing scripts with side effects can lead to undesired states.	3
Poor documenting	Hinders understandability and maintainability	3
Manual infrastructure	Some parts of the configuration are made manually, outside of IaC scripts.	2
Nodes too deep	The tree of nodes generated from a single script is too deep.	2

Moving to best practices to develop IaC (**Q7**), we were able to distill four practices from the practitioners' comments. The most common (12 mentions) was the *Secret-Injection* that fosters parametric configuration of IaC scripts. For instance, by loading environment variables from a separate `.env` file thanks to the `dotenv` module[2].

Following, break-fast allows one to test and fail as soon as possible, which counterfeits one of the disadvantages of IaC: the longer feedback loop, as one has to wait for the deployment to take place in order to see the results of the scripts. This was best expressed in one of the comments given by our interviewees:

> *"Develop and test in small increments (make added features small and as orthogonal as possible), setup testing environment early on, make your deploy as deterministic as possible".*

Last but not least, reuse by abstraction and low nesting both aim to modularize templates (*a la* object-oriented code) which also makes them reusable, simple and shallow (not nesting many nodes).

In summary, there are some development practices inherited from standard code that seem to be applicable to IaC with some considerations due to its singularities – e.g., being declarative and tree-alike.

C. *RQ*$_2$. *IaC Development Tooling*

Moving to tool support, we first assessed the use of common development tools (**Q8**) when developing IaC. Table V summarizes the types of tools mentioned by interviewees along with some examples also extracted from their answers. The most common ones are IDEs (11), followed by text editors, linters – static code analyzers that detect potential errors – and models such as UML diagrams and dependencies graphs (4 each). What we can observe is that the automated support relates to various types of tools, which may be a sign that

[2]https://github.com/bkeepers/dotenv

Table IV: Best practices for developing IaC (**Q7**), ordered by number of mentions.

Best-practice	**Description**	**#**
Secret-Injection	Keep all contents of the blueprint or IaC scripts parametric so that the orchestrator or users can inject the desired results at will	12
Break-fast	program the infrastructure to be buildable as fast as possible and hopefully as fast-breaking as possible furthermore, infrastructure circuit-breakers are needed to minimize waste	8
Reuse by Abstraction	Making templates and scripts also recall each-other to allow for interdependency but also interchangeability and possibly reuse, e.g., object orientation	6
Low-Nesting	Keep nodes nesting to at most one level of recall (i.e., tree of height 2)	5

Table V: Summary of most common support tools for IaC (**Q8**).

Type of Tool	**#**
IDEs (IntelliJ, Visual Studio Code)	11
Editors (Sublime, emacs, vim)	4
Linters (pylint, cfn-lint)	4
Models (UML, dependencies graphs)	4
Testing (molecule, pytest)	3
Own Scripts	3
Monitoring (grafana, Amazon x-ray)	2
Ontologies (no examples given)	2
Auto-documentation (sphinx)	1

practitioners have not an objective idea of what an automated support for IaC would be.

Next, we wanted to get insights regarding IaC-specific tools and languages that are common among practitioners (**Q9**). Indeed, the IaC technology ecosystem is currently characterized by a plethora or different and often overlapping (in terms of their goals) tools and languages. Thus, it is important to study and understand their adoption, plus identifying those that nowadays represent the *de facto* standard way of writing IaC. Results of this analysis are reported in Table VI, showing that no IaC tool is currently used by more than 60% of respondents, with Docker being the most used technology with 59.5% of respondents using it, confirming it as the *de facto* standard containerization technology. This also confirms the observation that the IaC technology ecosystem is currently very scattered, heterogeneous and not fully understood, with no single tool dominating the market.

We can also observe that 69% of interviewees declared to use three or more different tools. This shows the lack of a single tool covering the various aspects of IaC in an unified way. As a consequence, developers tend to select a stack of tools, each having a different purpose, whose combination enables full IaC development. This seems to be confirmed also by looking at the top tools being adopted by more than 30% of survey respondents, namely Kubernetes, Vagrant, Chef, Terraform, Ansible and Docker. Indeed, each of these tools

Table VI: Results on the adoption of IaC tools, ordered by decreasing frequency (**Q9**).

Tool	#	Usage %
Docker	26	59.0%
Ansible	23	52.2%
Vagrant	19	43.1%
Kubernetes	18	40.9%
Chef	16	36.3%
Terraform	15	34.1%
Puppet	13	29.5%
Apache Brooklyn	9	20.0%
Packer	9	20.0%
CloudFormation	9	20.0%
TOSCA	8	18.2%
Salt	6	13.6%
Shell scripts	4	09.0%
Cloudify	3	06.8%
Octopus Deploy	1	02.3%
Azure DevOps	1	02.3%

deals with a different aspect of IaC development: Kubernetes enables orchestration of containers of any kind; Vagrant allows to define and manage virtual machines; Chef and Ansible deal with configuration management of services; Terraform aims to orchestrate services deployed on different infrastructures (VMs, containers, public and private clouds); Docker is the weapon-of-choice when building containers. Another observation stemming from the results on IaC tools adoption regards language standardization: Despite the standardization effort behind TOSCA [3], TOSCA-enabled orchestrators are still far from being the standard solution for IaC development.

Afterwards, in **Q10** we aimed to disguise the pros and cons for each DevOps language/tool. For doing this we elicited six categories from the answers obtained, that help understand the main concerns of the practitioners when using the tools. These may be categorized as follow:

- **Coding**: Refers to the ease of building scripts for a given tool/language (e.g., being declarative or imperative);
- **Portability**: Refers to the portability of the scripts to different target infrastructures (e.g., configuring Linux nodes vs configuring Windows nodes transparently);
- **Automation**: Refers to the degree of automation on the infrastructure building (e.g., one can create a VM or a complete cluster automatically with the same scripts);
- **Usability**: Refers to the usability of a given language/tool (e.g., if it provides a local development/testing environment or higher abstractions to design infrastructure);
- **Extensibility**: Refers to the degree in which a given language/tool can be used outside of its original context (e.g., by providing hooks to other tools/platforms);
- **Maturity**: Refers to the stability and activity of the language/tool and its broad adoption by the community (e.g., by having active repositories on github or a big number of contributors).

Table VII summarizes the results for the answers to **Q10** according to the categories above. We grouped positive and

[3]https://www.oasis-open.org/committees/tosca

Table VII: Categorized pros and cons comments for the IaC languages/tools (**Q10**), ordered by number of mentions.

	Coding		Portability		Automation		Usability		Extensibility		Maturity			
Tool	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**Tot**	**Ratio**
Docker (compose)	1	2	2				1	2					8	0.00
Chef		2		2					1		2		7	-0.14
TOSCA		1	3				1		1			1	7	0.42
Ansible	1	2	2						1				6	0.33
Puppet		2		1				1					4	-1.00
Terraform		1			1						1	1	4	0.00
Vagrant							1	1	1		1		4	0.50
Shell-Script		1	1		2								4	0.50
Cloudify	1	2							1				4	0.00
CloudFormation		1		1	1								3	-0.33
Apache Brooklyn			1							1	1		3	0.33
Kubernetes		1	1								1		3	0.33
Saltstack	1										1		2	1.00
PowerShell					1								1	1.00
Packer					1								1	1.00

negative comments into the six categories and then present a total of the different mentions (nevertheless positive or negative) and a ratio per tool ($ratio_{tool} = (pos - neg)/total$), which shows whether the overall feeling for the tool among practitioners is positive/negative. The tools whose number of mentions are above average are Ansible, Chef, Docker and TOSCA, while the most positively perceived tools are Powershell, Packer and Saltstack – however those have only a few mentions which indicates low popularity. The negatively perceived tools are Puppet and Chef, although we notice that more comments on the tools (either negative or positive) indicate more adoption or popularity.

All in all, our findings suggest that there is no *one-size-fits-all* tool, as also explained by one of our interviewees:

> *"All the tools have cons as far as I'm concerned, its a matter of combining them effectively through experience and canceling those cons out".*

In summary, these results clearly point out the need for further research and industry efforts on the tool support provided to IaC developers. Besides, the negative perception of some tools such as Puppet and Chef could be indeed due to the higher popularity of these tools, as practitioners complain more of tools that they use more. This result calls for more studies that corroborate this finding in a more quantitative manner.

D. *$\boldsymbol{RQ}_3$: Challenges and Directions for IaC Development*

For the last research question **RQ**$_3$, we started by identifying current issues when developing IaC (**Q11**). As shown in Table VIII, the most critical issue for developers is Testability (14 mentions). This finding is aligned with the most prevalent disadvantage with respect to standard code, namely impossible testing (recall Table II). In particular, as reported by one of our interviewees:

> *"Issues are mostly related to setting up a testing environment, since this is usually quite a complex problem. Combine that with no standard practices when it comes to testing and you have one big mess".*

The last set of questions in Table I addresses what practitioners consider as a priority regarding the support of IaC

Table VIII: Most common IaC issues (**Q11**).

Issue	Description	#	Relation
Testability	Testing is impossible and verifyability as well as peer-review or code-review is too basic, no review guidelines	14	
Readability/ Polyglot	infrastructure triaging involves many different formats which are often obscure to most and need specialised personnel	10	
Inconsisten-cy	Versions often break back-ward compatibility and templating is different; new versions of the language may not be supported by the tools	6	Lack of IDE
Runtime Automa-tion	Automating the infrastructure at runtime is still very limited and managed events are often reduced to elasticity/scalability properties only	4	
Portability	Difficult to port the nodes between technologies especially when there are multiple technologies in a polyglot topology with multiple hooks of different types in the same node	4	Readability/ Polyglot
Concurren-cy	Race-conditions and circular dependencies in pipelines both in actual and dry-runs; testability for pipelines is still poor	3	Testability
Lack of IDE	A development environment with pre-made code and compatibility with other more naive formats is missing	3	Runtime Automa-tion

lifecycle (from **Q12**to **Q17**), and also a final question (**Q18**) where interviewees could further clarify their previous answers and share their vision on the future of IaC.

For questions **Q12–Q17** we defined a Likert scale with 5 values (ranging from *Not Important* to *Extremely Important*) on the importance of supporting different aspects of IaC: from tool support of critical activities such as testing, to the extension of IaC coverage to heterogeneous infrastructures (e.g., GPUs, Bare metal). Then we defuzzified the answers (i.e., expressed them numerically) using a triangular membership function [11]. The triangular function assigns a numerical value in the normalized range [0,1] for each linguistic response: in our case, from *Not Important* $= 0.2$ to *Extremely Important* $= 1$. Then, one can analyze the responses distribution upon such range, and calculate the average [12].

All in all, not surprisingly the vast majority of the responses pointed out that Testing Frameworks are extremely important and needed on the context of IaC with a value of 0.90 in the normalized scale. This confirms the findings and tendencies of the previous answers discussed along Sections IV-B and IV-C. However, support for all other activities is also conceived as highly important: Testing support was followed by Static analysis tools (0.82), languages standardization (0.80), security and privacy (0.76) support for heterogeneous infrastructures (0.75) and, finally, ad-hoc IDEs (0.74). One can see that all values are high and practitioners are thriving for support of any kind to develop and maintain their IaC scripts.

The final question (**Q18**) captured the concerns regarding the future of IaC. In the first place, practitioners indicated the advent of serverless computing (also known as Functions-as-a-Service - FaaS). Serverless is a fully outsourced cloud computing model in which the cloud provider dynamically manages the allocation of machine resources, while developers only concentrate in writing application code, i.e., functions [13]. In such a model, management of the myriad of functions becomes even more difficult than in typical cloud scenarios. As pointed out by the practitioners:

> "*Serveless will be a mess – we have an upcoming pipeline with dozens if not hundreds of concurrent functions, management will be a mess, perhaps TOSCA policies could help.*".

In summary, there are concerns among practitioners about the overlapping of the increasing number of tools; Quality assurance and testability both in production and local development environments; and finally but perhaps most important, the organizational rewiring that lies beyond IaC, which goes towards embracing DevOps. More in detail:

> "*Not only focus on technology but also the change in work processes and organization that are needed when using IaC to better serve the business.*".

V. Discussion and Implications

The results of our study highlight a number of insights to be further discussed in order to explicitly address our research questions.

A. *RQ*$_1$*: On the adoption of Infrastructure-as-Code*

There does not exist one full-fledged and bullet-proof solution for IaC, rather the tools used by practitioners are very varied and also very common; out of the top 8, the median frequency is 15,5 times a tool is used with no major winner whose frequency greatly surpasses the others. This denotes a proliferation and divergence of tools which could make maintenance and evolution of IaC sensibly more difficult; as highlighted in Table VII there exists a sensible variety on the dimensions reported by practitioners to evaluate used technologies. In essence, to address our **RQ**$_1$, namely, *How do practitioners currently develop infrastructural code?*, practitioners still tend to develop IaC on a best-effort basis, diversifying the tool used based on several still implicit principles, with no best fit-for-purpose tool and little to no best practices for their integration. More specifically, reported best practices reflect more how to internally structure and write the code in the same snippet as opposed to combining multiple tools and formats together. Beyond that, one best practice explicitly aims at *avoiding* to combine multiple formats since this lowers the general quality and maintainability of the blueprint, denoting that practitioners are explicitly working to simplify overly complex blueprints and avoiding interoperation. However, at the same time another best practice proposes the recombination of possibly diverse formats by abstraction (e.g., using the OASIS TOSCA standard for IaC and including multiple formats inside node-type definitions). These rather conflicting best practices indicate a *balance* in the complexity of IaC blueprints which is still implicit and needing further research to be empirically established.

B. $\mathbf{RQ}_2$: On the support given by Infrastructure-as-Code tools

The data reveals no less than 6 dimensions which allow comparison and trade-off between technologies, but the lack of a specific "winner" in terms of most-frequently adopted solutions indicates that much work still needs to be done to establish and fill the gaps in the current state of practice. Furthermore, the matrix in Tab. VII is rather *sparse*, meaning that technology vendors are, perhaps deliberately, assuming that Operations engineers would intermix series of tools as part of a DevOps *pipeline* which, although true in fact, reveals tool-specific as well as tool-interaction challenges that need further attention from a maintenance and IaC evolution perspective. To address specifically $\mathbf{RQ}_2$, namely *How do currently available tools support practitioners when developing infrastructural code?*, available tools offer very limited quality automation as well as maintenance & evolution support features. The findings recapped in this section can aid the work of practitioners providing them a heads-up concerning the tools and approaches that they exploit as part of their IaC adoption. Furthermore, the findings clearly outline avenues for further research beyond the state of the art.

C. $\mathbf{RQ}_3$: On the Challenges of Infrastructure-as-Code

Tools and automation in infrastructure code reveal several typical issues such as consistency-breaking across different versions of the same technology or level of runtime automation. Perhaps most importantly, the most frequent challenges reflect testability and understandability which have been empirical shown to be deeply interwoven and also majorly affecting production code, as seen in previous work [14]. Overall, it seems the maintenance and evolution research on IaC code features and automation needs to undergo the same tortuous road previously walked by practitioners and academicians for production code; our data highlights that the challenges along this path are considerable. To address $\mathbf{RQ}_3$ namely, *What are the challenges that practitioners face when developing infrastructural code?*, as previously stated the practitioners clearly perceive it very difficult to test infrastructure code, e.g., while maintaining code readable and consistent with multiple formats. Furthermore, version management is still a big issue, considering the multiple technologies involved and their dependencies. In addition, there is an implicit non-trivial relationship between readability and polyglottism in IaC blueprints and their *portability*; these two challenges implies a trade-off which is yet to be fully understood and deserving further study. Finally, perhaps from an overarching perspective, the lack of an IDE specifically designed to support the development, operation, and maintenance of infrastructure code is perceived as a challenge, although not from the majority of practitioners. It should be noted that several practitioners highlight the usage of formats which do in fact provide some basic development environments (e.g., several TOSCA implementations, Terraform and more), there are in general very few supporting IaC development (not only IDEs, but also static analyzers, tools supporting security aspects, etc.), highlighting another direction for research.

D. Observations, Lessons Learned, and Implications

The first key observation evident from our data is that, on the one hand, there is evidence for a proliferation of infrastructure code automation tools, techniques, languages, and approaches but, on the other hand, the key standard in the field, namely OASIS TOSCA, is adopted by a mere 20% of the total practitioners in our sample. This could identify a shortcoming of the standard in terms of its dissemination and exploitation or a divergence of its practicability from the real requirements that practitioners put forth. This study may serve as a lens to understand those requirements perhaps bringing about alignment between the two and thus fostering better exploitation of the OASIS open standard for the benefit of IaC maintenance and evolution.

The second key observation reflects on the striking diversity of *types* of tools we reported in our sample, ranging from orchestration tools (e.g., CloudFormation, Saltstack) to configuration management (e.g., Puppet), to topology management (e.g., Kubernetes), to containerization (e.g., Docker) and more. It is evident from this data that infrastructure code is polyglot by design and therefore practitioners and academicians alike should strive to find the right blends, patterns, and any anti-patterns matching or existing in their practice and theoretical outlook. It is also evident that an ontology of IaC, namely, a representation and definition of the categories, properties, and relations between IaC concepts, data, and entities [15], could aid in reasoning how certain IaC tool/format patterns may be fit for purpose in specific scenarios or whether those tools may adhering specific properties (e.g., consistency, readability). Perhaps an ontology engineering [16] is needed in order to distill such an ontology for the benefit of practitioners and academicians alike.

VI. Threats to Validity

A number of threats might have influenced our findings. In this section, we summarize and explain how we mitigate them.

A. Threats to construct validity

To conduct our study and get the practitioner's perspective on the state-of-the-practice of IaC, we have proceeded with semi-structured interviews. Rather than alternative approaches (e.g., surveys), our methodology allowed us to have a direct interaction with practitioners and ask them more details on the three aspects treated with our research questions. Of course, we are aware that our empirical investigation is limited to the analysis of the developer's perception and, indeed, we plan to complement our study with a large-scale mining software repository investigation of how infrastructure code is actually treated by developers. Another discussion point in this category is related to the set of practitioners surveyed. We explicitly took into account developers having experience with infrastructure code: to enable it, we designed a specific recruitment process that allowed us to get in touch with IaC experts. Nevertheless, our study may miss the perspective of novice IaC practitioners: we are aware of this potential limitation, however our study was explicitly focused on gathering opinions from experts of the field. Replications targeting novice developers is part of our future research agenda.

B. Threats to conclusion validity

To address our **RQ**s, we collected and transcribed the semi-structured interviews performed. Afterwards, we applied an iterative content analysis method [10] to assign the specific themes emerged from the practitioner's point of view. Afterwards, to ensure the validity of the assigned codes, we opened an internal discussion aimed at addressing possible conflicting names or misunderstanding. As explained in the previous section, our analysis is qualitative in nature: complementing it with quantitative methods is part of our future work.

C. Threats to external validity

Threats in this category concern with the generalisability of the results. We interviewed 44 practitioners from as many companies: this made our analysis quite broad and able to give us a good coverage of the practitioner's view on the state of the practice of infrastructure code development. Of course, an even broader analysis would corroborate our findings further.

VII. Related Work

IaC has recently received increasing attention in the research community, mainly due to the paradigm shift it brings in software design and development. Various previous works relate to our study by empirically investigating the adoption, defects, or challenges of IaC.

Schermann et al. [17] proposed a structured database of information about over 100,000 Docker files retrieved from over 15,000 GitHub repositories. The database is used to answer questions about typical images, programming languages, quality defects, and files evolution. The database was made available open source for future research and, although limited to the scope of Docker, could prove useful to validate our results from a mining software repository perspective.

In the field of IaC defects and smells, various works have appeared in very recent years. Jiang and Adamns [18] analyzed the co-evolution between infrastructure and production code, finding that the former is tightly coupled with test files, leading testers to often change infrastructure specifications when modifying tests. Sharma et al. [19] looked for code smells in the source code of configuration management tools (e.g., Puppet, Chef). As a result, they proposed a catalog of 13 *implementation* and 11 *design* configuration smells. The catalog was then benchmarked against 4,621 Puppet open source repositories. Interestingly, design smells showed higher average co-occurrence with respect to the implementation smells. That is, one wrong or non-optimal design decision introduces many quality issues in the future.

Rahman et al. [20] investigated the challenges in developing IaC, specifically in the context of configuration management tools. They looked for the questions that were more asked by programmers on Stack Overflow with the goal of helping IaC developers. Also in this case, the focus was on Puppet-related questions. By applying qualitative analysis they identified the three most common question categories as being (i) syntax errors, (ii) provisioning instances, and (iii) assessing Puppet's feasibility to accomplish certain tasks. The three categories of questions that yielded to the most unsatisfactory answers were installation, security and data separation. Then, the authors classified IaC defects according to standard non-IaC defects categories by doing qualitative analysis of commit messages and issue report descriptions in open source projects, limited to Puppet code. Syntax and configuration-related defects turned out to be the most common categories. Moreover, these two categories of defects were much more prevailing in IaC than in non-IaC software. Later on, Van der Bent et al. [21] defined a measurement model to assess the quality of Puppet code.

In the scope of software security, Shu et al. [22] studied vulnerabilities in Docker Hub. They proposed a framework for Docker Images Vulnerability Analysis (DIVA) able to automatically discover and analyze images from Docker HUB. DIVA found more than 180 vulnerabilities on average when considering all versions of images. Moreover, many images were not updated for hundreds of days and that vulnerabilities easily propagate from parent images to child images. The authors advocated for more automated methods for applying security updates to Docker images.

Also Rahman et al. [23] presented a catalog of seven security smells in IaC. These were extracted from qualitative analysis of Puppet scripts in open source repositories. The identified smells comprise: (1) granting admin privileges by default, (2) empty passwords, (3) hard-coded secrets, (4) invalid IP address binding, (5) suspicious comments (such as 'TODO' or 'FIXME'), (6) use of HTTP without TLS, and (7) use of weak cryptography algorithms. However, this is again limited to Puppet scripts and not all smells are generalizable to other languages or tools. Finally, Rahman et al. [24] investigated the research challenges in IaC through a Systematic Literature Review (SLR). The main goal was to identify the various research areas surrounding the field of IaC. The four main topics that have been identified are (i) framework/tool for IaC; (ii) use of IaC; (iii) empirical studies related to IaC; and (iv) testing in IaC. They concluded that, while several studies exist on framework and tools, research in the context of IaC defects and security flaws is still at its early stages. The results of the SLR are perfectly in line with the research proposed in this paper: indeed, the state of research and practice in IaC is still immature, which calls for more empirical and industry-focused studies, as the one carried out throughout this paper.

VIII. Conclusion

DevOps is a family of tactics that aim at accelerating deployment and delivery of large-scale applications. Essentially, all the DevOps automations are driven by infrastructure code, that is, the series of *blueprints* laying out the application infrastructure, its dependencies, and involved middleware across a DevOps pipeline.

In this paper we investigate infrastructure code tools, practices, and challenges from a practitioner perspective, with 44 semi-structured practitioner interviews. Our findings reveal critical insights into the available tools, their complexities, the challenges thereto, as well as best practices adopted by practitioners to address (some of) those challenges. Overall, however, the most direct conclusion stemming from our evidence is that the field of software maintenance and evolution of IaC is in its infancy and deserves further attention.

On the one hand, several best practices exist but they are mostly concentrated on limiting the complexities inherent within IaC. On the other hand, many challenges exist, from conflicting best-practices, to lack of testability, readability issues, and more.

Practitioners can benefit from our results by: (1) using the content analysis and comparisons provided in Sec. IV to identify the tools and practices best fitting with their domain and have an evidence-based view over what's available currently in the state of practice; (2) understand and be prepared for the challenges inherent with adoption of IaC; (3) control for the levels of complexity and plan ahead for major maintenance and evolution phases in their IaC strategy.

In the future we plan to extend this work with tools that address the identified challenges. First, we plan to refine pattern detection tools and metrics able to appraise the level of complexity of a IaC blueprint and identify any recurrences which may amount to any anti-pattern revealed in our findings. Second, we plan to refine and test prototypical support for the automated measurement of IaC readability, understanding empirically how that relates to higher or lower IaC quality and maintainability. Finally, we plan to improve the testability of IaC starting from state-of-the-art testing approaches from production code and empirically establishing the extent to which those approaches generalise to IaC as well.

References

[1] L. J. Bass, I. M. Weber, and L. Zhu, *DevOps - A Software Architect's Perspective.*, ser. SEI series in software engineering. Addison-Wesley, 2015.

[2] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero, and D. A. Tamburri, "Devops: introducing infrastructure-as-code," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 497–498.

[3] K. Morris, *Infrastructure As Code: Managing Servers in the Cloud*. Oreilly & Associates Incorporated, 2016. [Online]. Available: https://books.google.si/books?id=kOnurQEACAAJ

[4] M. Hüttermann, "Infrastructure as code," in *DevOps for Developers*. Springer, 2012, pp. 135–156.

[6] D. Soldani, B. Barani, R. Tafazolli, A. Manzalini, and C.-L. I, "Software defined 5g networks for anything as a service [guest editorial]." *IEEE Communications Magazine*, vol. 53, no. 9, pp. 72–73, 2015. [Online]. Available: http://dblp.uni-trier.de/db/journals/cm/cm53.html#SoldaniBTMI15

[5] M. Jarschel, "Network function virtualization: Towards the commoditization of middle boxes," DATEV Trendscout, Nurnberg, Germany, 11 2013.

[7] P. Lipton, D. Palma, M. Rutkowski, and D. A. Tamburri, "Tosca solves big problems in the cloud and beyond!" *IEEE Cloud Computing*, vol. 5, no. 2, pp. 37–47, 2018. [Online]. Available: http://dblp.uni-trier.de/db/journals/cloudcomp/cloudcomp5.html#LiptonPRT18

[8] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, 2018.

[9] R. S. Weiss, *Learning from strangers: The art and method of qualitative interview studies*. Simon and Schuster, 1995.

[10] B. Hanington and B. Martin, *Universal methods of design: 100 ways to research complex problems, develop innovative ideas, and design effective solutions*. Rockport Publishers, 2012.

[11] W. Pedrycz, "Why triangular membership functions?" *Fuzzy sets and Systems*, vol. 64, no. 1, pp. 21–30, 1994.

[12] A. Christoforou, M. Garriga, A. S. Andreou, and L. Baresi, "Supporting the decision of migrating to microservices through multi-layer fuzzy cognitive maps," in *International Conference on Service-Oriented Computing*. Springer, 2017, pp. 471–480.

[13] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski *et al.*, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20.

[14] A. Trockman, K. Cates, M. Mozina, T. Nguyen, C. Kastner, and B. Vasilescu, ""automatically assessing code understandability" reanalyzed: combined metrics matter." in *MSR*, A. Zaidman, Y. Kamei, and E. Hill, Eds. ACM, 2018, pp. 314–318. [Online]. Available: http://dblp.uni-trier.de/db/conf/msr/msr2018.html#TrockmanCMNKV18

[15] N. Guarino, "Formal ontology and information systems," 1998.

[16] Y. Sure, S. Staab, and R. Studer, "Ontology engineering methodology," 2009.

[17] G. Schermann, S. Zumberi, and J. Cito, "Structured information on state and evolution of dockerfiles on github," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: ACM, 2018, pp. 26–29. [Online]. Available: http://doi.acm.org/10.1145/3196398.3196456

[18] Y. Jiang and B. Adams, "Co-evolution of infrastructure and source code: An empirical study," in *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 2015, pp. 45–55.

[19] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?" in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 189–200. [Online]. Available: http://doi.acm.org/10.1145/2901739.2901761

[20] A. Rahman, A. Partho, P. Morrison, and L. Williams, "What questions do programmers ask about configuration as code?" in *Proceedings of the 4th International Workshop on Rapid Continuous Software Engineering*, ser. RCoSE '18. New York, NY, USA: ACM, 2018, pp. 16–22. [Online]. Available: http://doi.acm.org/10.1145/3194760.3194769

[21] E. Van der Bent, J. Hage, J. Visser, and G. Gousios, "How good is your puppet? an empirically defined and validated quality model for puppet," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 164–174.

[22] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on docker hub," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, ser. CODASPY '17. New York, NY, USA: ACM, 2017, pp. 269–280. [Online]. Available: http://doi.acm.org/10.1145/3029806.3029832

[23] A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *Proceedings of the 41st International Conference on Software Engineering*, 2019, in Press.

[24] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, "Where are the gaps? A systematic mapping study of infrastructure as code research," *CoRR*, vol. abs/1807.04872, 2018. [Online]. Available: http://arxiv.org/abs/1807.04872