Infrastructure as Code: Technology Review and Research Challenges

Claus Pahl[1], Niyazi Gokberk Gunduz[1], Övgüm Can Sezen[1], Ali Ghamgosar[1] and Nabil El Ioini[2]
[1]*Free University of Bozen-Bolzano, 39100 Bolzano, Italy*
[2]*University of Nottingham, 43500 Semenyih, Malaysia*

Keywords: Infrastructure as Code, IaC, DevOps, Defect, Smell, Pattern, Drift, Technology Review, Research Challenges.

Abstract: The quality of software management in infrastructure operations for application software is important as automation in software operations continues to grow. Infrastructure as Code (IaC) refers to a systematic, technology-supported approach to manage deployment infrastructure for software applications. Sample contexts are general software automation, but also cloud and edge and various software-defined networking applications. DevOps (development and operations) practices, which are already applied in the Infrastructure as Code (IaC) context, need to be extended to cover the whole IaC life cycle from code generation to dynamic, automated control. The ultimate objective would range from IaC generation to full self-adaptation of IaC code in an automated setting. We review available IaC technologies based on a comprehensive comparison framework to capture the state-of-the-art. We also introduce an IaC-specific DevOps process. This serves as a basis to identify open research challenges. A discussion of defect categories is at the centre of this process.

1 INTRODUCTION

Software management depends on the quality of the software operation within the chosen infrastructures. These are also central concerns in the DevOps (Development and Operations) approach to software lifecycle automation. The automation of these operations concerns is important for instance in cloud and edge computing or software-defined networking. Infrastructure-as-Code (IaC) is an approach that enables the automation of deployments, configurations, and management tasks. Tool support enable the creation, execution and management of secure, reliable, and ideally self-healed IaC software. The DevOps approach plays a crucial role here. It is widely adopted for managing software. When applied to the context of IaC, DevOps practices must extend toward dynamic, automated control (Pahl et al., 2019). The objective for an IaC lifecycle support in a DevOps framework range from the generation of IaC code to self-adaptation of IaC code at the end of the cycle.

Our contributions in this work are a comprehensive review of the current state-of-the-art in IaC, framed within a DevOps-specific IaC framework. This review covers the following key areas:

- **Technology Review:** Development of a comparison framework and an analysis of key technologies. Central aspects include context, purpose, language, and architecture.
- **DevOps Framework for IaC:** A specific lifecycle tailored for IaC that builds on a general application DevOps cycle, with an emphasis on IaC quality management and defect handling.
- **Research Challenges in the IaC Lifecycle:** An exploration of open concerns across the IaC lifecycle, using the DevOps framework as a basis.

2 RELATED WORK

Infrastructure as Code (IaC) is an active area of research, as a number of recent surveys demonstrate (Quattrocchi and Tamburri, 2023; Alonso et al., 2023; Kumara et al., 2021). More specifically, some technical contributions deal with different aspects of the IaC lifecycle (Borovits et al., 2022; Quattrocchi and Tamburri, 2022; Palma et al., 2022) with defects and quality management at the centre of their investigation. For instance, (Borovits et al., 2022) address the detection of inconsistencies as sources of possible defects using machine learning techniques. (Palma et al., 2022) investigate defect prediction in projects. In (Quattrocchi and Tamburri, 2022), also defect prediction is the focus. Quality management is a specific concern. Here, dedicated investigations into metrics catalogs exist (Palma et al., 2020). The recent European research project PIACERE (Alonso et al., 2023) aimed to cover the full IaC lifecycle from IaC code generation to code analysis to deployment to self-management of IaC solutions. It demonstrates

Pahl, C., Gunduz, N. G., Sezen, Ö. C., Ghamgosar, A. and El Ioini, N.
Infrastructure as Code: Technology Review and Research Challenges.
DOI: 10.5220/0013247700003950

In *Proceedings of the 15th International Conference on Cloud Computing and Services Science (CLOSER 2025)*, pages 151-158
ISBN: 978-989-758-747-4; ISSN: 2184-5042

the feasibility of comprehensive tool support for all stages, but also indicates open challenges. Security is emphasised as the central property in a DevSecOps framework. In a special issue (Quattrocchi and Tamburri, 2023) on the IaC topic, the individual contributions have addressed distributed deployment, lifecycle automation, maturity models and security as current concerns. In (Aviv et al., 2023) Infrastructure from Code (IfC – as opposed to IaC) is investigated. Here, 14 cloud infrastructure procedures are identified in four categories. Links between Infrastructure as Code and software architecture conformance are the topic in (Ozkaya, 2023). Software architecture distinguishes modules as structural design concerns from component and connector concerns that focus on the behavioural side and allocation structures for the required resources. The latter is automated by IaC. Architecture compliance of deployments thus needs to be a continuous process. Both (Karanjai et al., 2023) and (Sokolowski et al., 2023) address the decentralization of Infrastructure as Code, aiming to take into account decentralised deployments across teams. A crucial role play interfaces, which can be declaratively specified to avail of other teams' deployments. Declarativeness allows match-making between the users' wishes and the providers' offers in a continuous process. (Staron et al., 2023) review research into IaC covering the full lifecycle. The importance of the IaC language is noted. In the verification and defect identification, particularly security is singled out, and mechanisms such as security smell detection are identified. Defect prediction is another important research direction.

For large-scale distributed systems that need to adapt to a changing environment, conducting a reconfiguration is a challenging task (Chardet et al., 2021). In particular, efficient reconfigurations require the coordination of multiple tasks with complex dependencies. We present Concerto, a model used to manage the lifecycle of software components and coordinate their reconfiguration operations. Concerto promotes efficiency with a fine-grained representation of dependencies and parallel execution of reconfiguration actions, both within components and between them.

3 TECHNOLOGY REVIEW

We will begin with the introduction of our framework for classification by providing an explanation of the dimensions and several aspects related to them. These dimensions and their aspects will then be used for the categorization of IaC tools on the market. Next, we will present the results acquired after evaluating the tools according to the aforementioned dimensions. We found this categorization necessary due to the fragmentation of IaC tools in terms of functionality. While different tools support different functionalities, those that share similar functionalities often vary in how they implement their services.

3.1 Classification Framework

IaC tools will be categorized according to the following four dimensions:

- **Context:** information regarding the environment of the technology like whether if open-source or not, which cloud service providers it is compatible with, how large is the community and for how long it has been supported.
- **Functionality:** information related to the service provided by the tool like whether if it is mainly for provisioning or configuration and if it treats the infrastructure as mutable or not.
- **Language:** information about the programming language the tool utilizes like whether if it is general purpose or a domain specific language and whether if it is procedural or declarative
- **Architecture:** information about the architecture of the solution provided by the technology like whether if it uses a master node to coordinate action or agents on the nodes or not.

A summary of the dimensions and their aspects can be seen in the Table 1. The dimensions are broken down into several, more concrete aspects and the range of values assignable to each aspect is defined.

Table 1: IaC Tools Classification Framework.

Dimension	Aspect	Range
Context	Accessibility	Open-Source/Closed-Source
	Cloud Compatibility	All/Specific Provider
	Community	Small/Large/Huge
	Maturity	Low/Medium/High
Functionality	Type	Configuration/Provisioning
	Infrastructure	Mutable/Immutable
Language	Paradigm	Procedural/Declarative
	Scope	DSL/GPL
Architecture	Master Server	Required/Not Required
	Agent Client	Required/Not Required

3.1.1 Context

The context dimensions aims to describe the selected tools in terms of the frame of reference in which they are actively used. This frame of reference is analyzed through the following aspects:

- **Accessibility:** The organization of the contribution to the technology.

- **Cloud Compatibility:** The compatibility with the various cloud service providers.
- **Community:** The size of the community that utilizes and/or supports the technology.
- **Maturity:** The duration for how long the technology has been supported.

The **Contribution Style** aspect can be assigned *Open-Source* or *Closed-Source* values. The **Cloud Compatibility** aspect can take the value *All* if the technology can be used with every cloud provider or a list of specific cloud providers if it is only compatible with a specific set. The **Community** aspect can take the values *Small*, *Medium* or *Large* depending on the amount of interaction Github or software related forums such as StackOverflow or social media sites like X. The **Maturity** aspect can take the values *Low*, *Medium* or *High* depending on the date the project has been made available.

3.1.2 Functionality

The functionality dimension describes a tool in terms of features it provides to the users. The features are analyzed through the following aspects:

- **Type:** The category of the functionality provided by the technology.
- **Infrastructure:** The selected infrastructure paradigm of the technology.

The **Type** aspect can be assigned *Configuration* or *Provisioning*. If the technology supports both, the most commonly used type will be selected. The **Infrastructure** aspect can be assigned *Mutable* or *Immutable* values.

Configuration Management and Provisioning: Configuration management and provisioning tools differ in the stage of the DevOps lifecycle they aim for. Configuration management tools are primarily focused on automatically monitoring and managing the state of the infrastructure that hosts the application, so they are related to the monitor/self-heal stage. Provisioning tools are primarily focused on initializing the infrastructure the application will be deployed on, so they are related to the deploy stage.

Mutable and Immutable Infrastructures: Mutable and immutable infrastructures refer to the way the tools consider the infrastructure in terms of the method of managing, replacing and updating them. A tool that considers infrastructure as mutable applies it's changes directly without the need for a rebuild. A tool that considers the infrastructure to be immutable do not change the existing configuration but instead creates new deployments to replace the existing one.

The relationship between configuration management and mutable infrastructures can be seen as the aim of configuration management is to edit and update the existing infrastructure according to the results of monitoring, which can be done only if the existing infrastructure is considered mutable. A similar relationship can also be observed between provisioning and immutable infrastructure as the aim of provisioning is to set up a new infrastructure for deployment, which signals for an infrastructure that is immutable.

3.1.3 Language

The language dimension aims to describe the selected tools in terms of the programming language they utilize. The programming languages are analyzed through the following aspects:

- **Paradigm:** The method of describing the infrastructure used by the programming language of the technology.
- **Scope:** The scope of the application domain of the programming language used by the technology.

The **Paradigm** aspect can be assigned *Procedural* or *Declarative* values. The **Scope** aspect can be assigned either *Domain-Specific Language (DSL)* or *General-Purpose Language (GPL)* values.

Procedural and Declarative Languages: Procedural and declarative languages differ in terms of the method of defining the infrastructure. Procedural languages explicitly defined the actions to be taken to reach the required state. Declarative languages define the properties of the desired final state of the infrastructure to be, leaving the instructions to create it to the underlying technology.

A relationship between declarative languages and configuration management tools can be formed, as the defined end state of the infrastructure can be used by the configuration management tool as a standard to manage. A similar relationship between procedural languages and provisioning tools can also be formed as the defined execution steps to reach the desired infrastructure can be run on each deployment.

General Purpose and Domain Specific Languages: General purpose languages (GPL) and domain specific languages (DSL) differ in terms of the scope of domains they are designed to be used for. A general purpose language (Java) is designed to be used for a variety of domains(ML, web development, etc.). A domain-specific language (Terraform HCL, etc.) is designed specifically to be used in a single domain (cloud infrastructure provisioning).

3.1.4 Architecture

The architecture dimension aims to describe the technology according to the architectural style of the solution provided by the technology. The architectural style is analyzed through the following aspects:

- **Master Server:** The requirement of a master server to utilize the technology.
- **Agent Client:** The requirement of an agent client to be installed on the configured infrastructure for the technology to be used.

Both the **Master Server** and the **Agent Client** aspects can be assigned *Required* or *Not Required* values.

Master Server and Agent Client: Some tools may require a server to be set up to manage the infrastructure. These servers are called "master" since the updates on the infrastructures are decided on these servers. Similarly, some tools may require an agent to be installed on the server that needs to be configured. These agents are called "client" as they are usually in a server-client relationship with master servers. Agent clients are responsible for applying the necessary updates on the existing infrastructure.

3.2 Technology Classification

This section displays and explains the results acquired after applying the aforementioned classification framework to the following selected IaC technologies in the market: **Chef**, **Puppet**, **Ansible**, **Pulumi**, **Heat**, **DOML**, **Terraform**, **TOSCA** and **CloudFormation**. The results of the classification can be seen in Table 2. The rows in the aspect column of Table 2 refer to the resource the information is acquired from. The links to these resource can be found in the list at footnote[1].

[1]
- Chef: https://docs.chef.io/ - github.com/chef/chef
- Pulumi: https://www.pulumi.com/docs/iac/ - github.com/pulumi/pulumi
- Terraform: https://developer.hashicorp.com/terraform/docs - github.com/hashicorp/terraform
- Puppet: https://www.puppet.com/docs/index.html - github.com/puppetlabs/puppet
- Heat: https://docs.openstack.org/heat/latest/ - github.com/openstack/heat
- TOSCA: https://docs.oasis-open.org/tosca/TOSCA/v2.0/csd07/TOSCA-v2.0-csd07.pdf - github.com/orgs/OpenTOSCA/repositories
- Ansible: https://docs.ansible.com/ - github.com/ansible/ansible
- CloudFormation: https://docs.aws.amazon.com/cloudformation/ - github.com/aws-cloudformation

4 DevOps FOR IaC

DevOps (Development and Operations) is a framework that captures the need for continuously used automation across the whole life cycle of software. We present an IaC-specific DevOps model here. This serves as a conceptual framework to align challenges and discuss defect management as a central problem.

4.1 DevOps Principles

It is important to distinguish DevOps for general application software and DevOps for IaC as a specific type of software. *Application DevOps* covers the full cycle for the application management automation (Pahl et al., 2020). *IaC DevOps* covers the IaC deployment part as part of the Ops automation. Some concrete challenges for IaC stem from the technology ecosystem (Rahman et al., 2019; Kumara et al., 2021). Market/technology fragmentation is the key problem as the discussion of language, approach and infrastructure dimensions above with different combinations has shown. A consequence, where multi-layer deployments are present, are requirements of a wide range of IaC skills.

4.2 An IaC DevOps Chain

In Figure 1, we present the IaC DevOps framework, associating IaC-specific activities to the four main phases Creation, Verification, Deployment and Management. We divide this into four phases:

- Phase 1: Coding - plan/create/code/build
- Phase 2: Quality assurance - test/verify
- Phase 3: Deployment - please/configure/deploy
- Phase 4: Management - operate/monitor/self-heal

5 PRINCIPAL CHALLENGES AND DEFECT HANDLING

Based on the IaC-specific DevOps framework from the previous section, we now identify and organise research challenges along the different phases. The wider challenges are then mapped to more concrete defects that the core of those challenges. Defects are a central concept as they will allow to associate a concrete remdial action to the problem encountered.

- DOML: https://www.piacere-doml.deib.polimi.it/specifications/DOML_Specification_v2.1.pdf - git.code.tecnalia.com/piacere/public/the-platform/doml

Table 2: Classification Table for the IaC Technologies.

Dimension	Aspect	Chef	Puppet	Ansible	Pulumi	CloudFormation	Heat	Terraform	TOSCA	DOML
Context	Accessibility [2]	Open-Source	Open-Source	Open-Source	Open-Source	Closed-Source	Open-Source	Open-Source	Open-Source	Open-Source
	Cloud Compatibility [1]	All	All	All	All	AWS	All	All	All	All
	Community [2]	Large	Large	Huge	Small	Small	Small	Huge	Large	Small
	Maturity [2]	High	High	Medium	Medium	Low	Medium	Medium	Medium	Low
Functionality	Type [1]	Configuration	Configuration	Configuration	Provisioning	Provisioning	Provisioning	Provisioning	Configuration	Provisioning
	Infrastructure [1]	Mutable	Mutable	Mutable	Immutable	Immutable	Immutable	Immutable	Immutable	Immutable
Language	Paradigm [1]	Procedural	Declarative	Declarative	Declarative	Declarative	Declarative	Declarative	Declarative	Declarative
	Scope [1]	GPL	DSL	DSL	GPL	DSL	DSL	DSL	GPL	DSL
Architecture	Master Server [1]	Required	Required	Not Required	Not Required	Not Required	Not Required	Not Required	Not Required	Not Required
	Agent Client [1]	Required	Required	Not Required	Not Required	Not Required	Not Required	Not Required	Not Required	Not Required

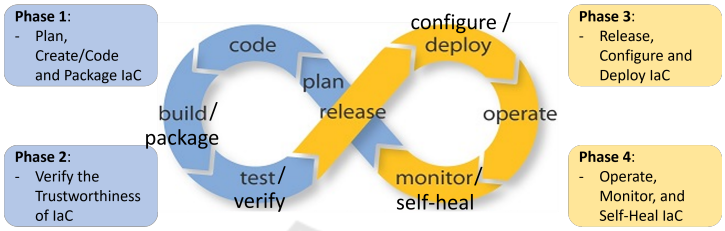

Figure 1: IaC DevOps Framework with IaC Creation, Verification, Deployment and Management Stages.

5.1 IaC DevOps Challenges

Specific challenges shall now be discussed to systematically cover the stages of the IaC DevOps lifecycle with its processing activities, starting with coding, followed by testing and quality assurance and finally maintenance and change management. We highlight in this subsection the key concerns before discussing them in more detail in the subsequent sections.

Stage 1: Coding has, apart from the languages aspects discussed above, aspects in software design that apply: the definition of well-known IaC *code patterns* and *anti-patterns*.

Stage 2: Quality assurance looks at a software verification and validation perspective. Generally, a difficulty in *replicating errors* is noted, as the state issue linked to the mutability aspect shows. IaC *languages differences and tools heterogeneity* also cause problems and *security and trustworthiness* are a specific concerns beyond a functionality view.

Stages 3 and 4: In particular maintenance and change management result in problems: *configuration drift* between the specification and its changing server state can be difficult to detect and deal with, and *changing infrastructure requirements* beyond basic maintenance cause another set of difficulties. We will make these challenges more concrete by identifying and categorising defects as the underlying concrete problems behind the challenges. To give an example of a concrete defect illustrating security challenges, Algorithm 1 shows a security-related defect where a password is exposed in logs. The defect

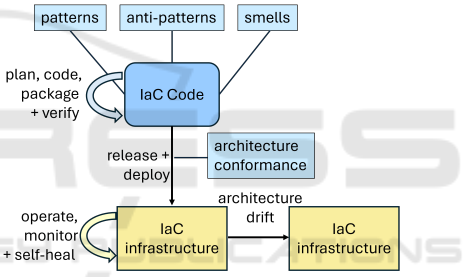

Figure 2: Conceptual Framework.

is mitigated by adding '$secret => true$', which can be done during the DEV-stages through code analysis. Patterns are examples of preventive measures to avoid problems. We also investigate what can go wrong across the lifecycle stages in more detail and also how to fix these defects. The main concepts are summarised in Figure 2.

5.2 Defect Handling

In general, defects in IaC scripts can be categorised according to when they occur in the different DevOps stages and whether they can be fixed statically or dynamically or by a combination of both. Defects can be organised in eight general categories (Rahman et al., 2020): syntax, conditional, documentation, dependency, service, itempotency, configuration data, security. We adopt their empirically determined categories, but also align them with the DevOps stages below in order to clarify possible remedial strategies.

For the defect handling, we need to consider a

Algorithm 1: IaC Script Defect – visible password, to be hidden.

```
glance_cache_config {
   ' DEFAULT / auth_url ' : value = > $auth_url ;
   ' DEFAULT / admin_tenant_name ' : value = > $keystone_tenant ;
   ' DEFAULT / admin_user ' : value = > $keystone_user ;
-  ' DEFAULT / admin_password ' : value = > $keystone_password ;                    [to be removed]
+  ' DEFAULT / admin_password ' : value = > $keystone_password , secret = > true ;  [to be added]
```

range of activities across different DevOps stages: (i) defect avoidance at early stages through smell and anti-pattern detection and pattern usage, (ii) defect identification across all stages, applying from code-level analysis during coding to log-based analysis of running systems, and (iii) defect remediation, including self-healing based on identified causes through root cause analysis (RCA) and other techniques.

A framework emerges that links static, dynamic and mixed defects with (i) preventive and early-stage detection (smell detection, anti-pattern detection to avoid defects), (ii) preventive use of patterns to avoid defects - although patterns address only two of the defect types (e.g., security and service as defect categories are address by caches, load balancers and circuit breakers as suggested patterns as we will explain in more detail below), (iii) defect detection needs to happen at all stages, and (iv) drift is property between the code and infrastructure level - drift can obscure smell and defects at code level due to the lack of architecture conformance. We discuss in detail some stage-specific challenges and then defect management.

5.3 Development Stages 1 and 2 (DEV)

Stage 1: Creation – Plan, Create, and Package the IaC: Coding has, apart from the languages aspects discussed above, aspects in software design that apply, specifically the definition of well-known IaC code patterns and anti-patterns. As a specific challenge, we focus here on patterns that in the form of design and architecture patterns are important mechanisms to design for quality in normal software development. The definition of IaC code patterns needs to be improved.

Defect avoidance can be based on patterns. Patterns are linked to quality objectives. Some common **code patterns** have emerged for IaC: caches for performance or load balancers for scaling. Another pattern category are **deployment patterns** that encode the availability of certain deployment types: deployment purpose such as canary releases or NFP objective such as green deployment These are well-known, but not yet properly coded as IaC patterns in catalog-format as they exist for general application software.

Another direction that needs more attention is **static defects and anti-patterns as defect indicators**, which we will look into below. The notion of **smells** can also be applied to identify potential defects. A program code smell is a characteristic in the source code of a program such as duplicate code or unsuitable naming, which could result in a potential problem, at least for future problem remediation. For IaC code, this is less well understood than for more established programming languages. These concepts are well-explored for recent application architectures (Cerny et al., 2023), but require better analysis and reflection for IaC (Opdebeeck et al., 2023).

Often, a variety of infrastructures are intended as targets for a single application. Depending on the stage (development, testing, integration, pre-production, or production), the infrastructure to be provisioned varies, which makes a manual management process work-intensive and error-prone.

Stage 2: Verification – Verify the Trustworthiness of IaC: Quality assurance looks at a software verification and validation perspective: generally, a difficulty in replicating errors is noted, as the state issue linked to the mutability aspect shows, IaC languages differences and tools heterogeneity also cause problems, and security and trustworthiness are a specific concerns beyond a functionality view,

The relevant quality concerns are latency, security and trustworthiness. However, the management of these qualities is made difficult by the following aspects. Systems created with IaC workflows are often large and complex to maintain. It is difficult to manually keep Minor configuration changes in IaC code can spread out to different parts of the system. An example is a Web site where security and network parameters can affect different parts of the system.

Defect Management - DEV Stages: Static defects cover coding and verification, with the following defect categories:

- Syntax: script syntax errors can occur, but are detectable statically.
- Conditional: erroneous logic or conditional values are more difficult to identify.

- Documentation: includes incorrect maintenance notes, readme files and other documentation.

Smells are another type of code assessment to indicate potential problems.

Prediction: Defect prediction is also of increasing importance: Collected metrics can be used to predict defect based on past system behaviour. This can be applied to performance as well as security concerns.

5.4 Operations Stages 3 and 4 (OPS)

Stage 3: Deployment – Release, Configure, and Deploy IaC: Distributed deployment is an open challenge as complexity rises. Here the heterogeneity already discussed might aggravate the problem by requiring specifications in different formats and not providing equally formatted or equally reliable feedback. Maintenance and change management also introduce additional problems, including: (i) configuration drift, where discrepancies arise between the specification and the changing server state, making detection and resolution difficult, and (ii) evolving infrastructure requirements that extend beyond basic maintenance, adding further complexity.

Configuration drift occurs when production or primary hardware and software infrastructure configurations deviate from their original state due to manual interventions. Configuration drift is the central challenge at this stage: Once a system is created via an IaC workflow, a manual modification of its configuration often leads to a misalignment (which is the configuration drift phenomenon) between the actual system and its initial infrastructural code. This occurs with more major changes, but also even smaller security problems need to be patched. Configuration drift is a natural phenomenon caused usually by large numbers of ongoing hardware and software changes. Configuration drift is said to accounts for 99% of reasons why recovery and high availability systems fail. Additionally, unidentified configuration drift poses significant risks, including data loss and outages.

As with other concepts, such as patterns, software engineering is aiming to deal with similar problems, e.g., architecture drift in a more general setting.

Stage 4: Management – Monitor, Self-Heal, and Replan: Changes at this stage cover changing quality for a given application configuration, but also changing infrastructure requirements. The infrastructure requirements may change over time, for instance, it might be necessary move an application from private on-premises to cloud/edge. This also includes the maintainability of the IaC and its consistency to the changes and needs of the infrastructure.

Ideally, the remediation of possible problems can be automated and self-healing of problems and self-adaptation to new requirements is feasible.

Defect Management - OPS Stages: Dynamic defects cover deployment and management stages, where the following defect categories occur:

- Dependency: such as missing artifacts, i.e, this is usually about availability as the quality concern.
- Service: e.g., insufficient provisioning, resulting in performance and availability quality concerns.
- Idempotency: is a reference to the problem that the repeatability of effects is often not given due to unobservable state differences.

Static and Dynamic is a defect category referring to a mix of categories of two individual detection stages:

- Configuration Data: includes pathname errors that generally lead to availability problems, which in some cases can be statically detected. Also other configuration option can be statically checked.
- Security: this covers the wider CIA (confidentiality, integrity, availability) concerns, some of which can be syntactically detected (such as the password leak above) and others such as attack-related actions only be dealt with dynamically.
- Architecture Conformance: The ongoing conformance to a specified system architecture needs to be continuously verified (Ozkaya, 2023). Otherwise, the drift phenomenon occurs that hinders comprehensive analysis and maintenance. For instance, root cause analyses benefit strongly from knowledge about the intended behaviour to allow an exact determination of defect root causes.

5.5 Summary

Table 3: Challenges Summary.

Stage	Challenge	Defect
Stage 1	Pattern + Anti-Pattern	Syntax, Conditional, Documentation
Stage 2	Replication, Heterogeneity Security	
Stages 3 + 4	Architecture drift, Change	Dependency, Service Quality, Idempotency, Configuration Data, Security, Architecture Conformance

Using a number of empirically determined challenges as the basis, we linked these to wider challenges that are aligned with our DevOps model, see Table 3.

6 CONCLUSIONS

We compared important open IaC technologies based on a structured catalogue of criteria to capture the state-of-the-art. Then, we identified challenges for IaC that would direct future research in the area. We aligned the challenges with an IaC-specific DevOps framework and mapped challenges to defect in order to indicate defect handling strategies within the DevOps phases. We have build on a variety of sources here to incorporate empirical research and thematic issues on the topic, but have extended and combined these into a coherent review of technologies and challenges. The review revealed a variety of different solution, which due to their heterogeneity create a diverse technology market with many challenges resulting from this. Variety allows for innovative directions to be taken, but also reflects that currently no consensus on successful directions and that, in many technology aspects, not enough best practice knowledge exist, adding to the list of open challenges.

What emerges is the need for more automation. In particular the management of defects requires more intelligent approaches for all stages. Here, solutions from general software engineering with is defect prevention, detection and remediation techniques using patterns, smell and drift identification can be borrowed to transfer ideas from general code to IaC as a specific code format. This requires the role of AI (Pahl, 2023; Tatineni and Chakilam, 2024) to be explored for the challenges in general and how in particular the defects across the DevOps stages can be addressed.

REFERENCES

Alonso, J., Piliszek, R., and Cankar, M. (2023). Embracing iac through the devsecops philosophy: Concepts, challenges, and a reference framework. *IEEE Software*, 40(1):56–62.

Aviv, I., Gafni, R., Sherman, S., Aviv, B., Sterkin, A., and Bega, E. (2023). Infrastructure from code: The next generation of cloud lifecycle automation. *IEEE Softw.*, 40(1):42–49.

Borovits, N., Kumara, I., Nucci, D. D., Krishnan, P., Palma, S. D., Palomba, F., Tamburri, D. A., and van den Heuvel, W. (2022). Findici: Using machine learning to detect linguistic inconsistencies between code and natural language descriptions in infrastructure-as-code. *Empir. Softw. Eng.*, 27(7):178.

Cerny, T., Abdelfattah, A. S., Maruf, A. A., Janes, A., and Taibi, D. (2023). Catalog and detection techniques of microservice anti-patterns and bad smells: A tertiary study. *Journal of Systems and Software*, 206:111829.

Chardet, M., Coullon, H., and Robillard, S. (2021). Toward safe and efficient reconfiguration with concerto. *Science of Computer Programming*, 203:102582.

Karanjai, R., Kasichainula, K., Xu, L., Diallo, N., Chen, L., and Shi, W. (2023). Diac: Re-imagining decentralized infrastructure as code using blockchain. *IEEE Transactions on Network and Service Management*.

Kumara, I., Garriga, M., Romeu, A. U., Di Nucci, D., Palomba, F., Tamburri, D. A., and van den Heuvel, W.-J. (2021). The do's and don'ts of infrastructure code: A systematic gray literature review. *Information and Software Technology*, 137:106593.

Opdebeeck, R., Zerouali, A., and De Roover, C. (2023). Behaviour-aware security smell detection for infrastructure as code. In *BE-NL Software Evolution*.

Ozkaya, I. (2023). Infrastructure as code and software architecture conformance checking. *IEEE Softw.*, 40(1):4–8.

Pahl, C. (2023). Research challenges for machine learning-constructed software. *Service Oriented Computing and Applications*, 17(1):1–4.

Pahl, C., Fronza, I., El Ioini, N., and Barzegar, H. R. (2019). A review of architectural principles and patterns for distributed mobile information systems. In *Intl Conf on Web Information Systems and Technologies*.

Pahl, C., Jamshidi, P., and Zimmermann, O. (2020). Microservices and containers. *Software Engineering Conference. Gesellschaft für Informatik eV*.

Palma, S. D., Nucci, D. D., Palomba, F., and Tamburri, D. A. (2020). Toward a catalog of software quality metrics for infrastructure code. *J. Syst. Softw.*, 170:110726.

Palma, S. D., Nucci, D. D., Palomba, F., and Tamburri, D. A. (2022). Within-project defect prediction of infrastructure-as-code using product and process metrics. *IEEE Trans. Software Eng.*, 48(6):2086–2104.

Quattrocchi, G. and Tamburri, D. A. (2022). Predictive maintenance of infrastructure code using "fluid" datasets: An exploratory study on ansible defect proneness. *J. Softw. Evol. Process.*, 34(11).

Quattrocchi, G. and Tamburri, D. A. (2023). Infrastructure as code. *IEEE Softw.*, 40(1):37–40.

Rahman, A., Farhana, E., Parnin, C., and Williams, L. (2020). Gang of eight: A defect taxonomy for infrastructure as code scripts. In *ICSE*.

Rahman, A., Mahdavi-Hezaveh, R., and Williams, L. (2019). A systematic mapping study of infrastructure as code research. *Information and Software Technology*, 108:65–77.

Sokolowski, D., Weisenburger, P., and Salvaneschi, G. (2023). Decentralizing infrastructure as code. *IEEE Softw.*, 40(1):50–55.

Staron, M., Abrahão, S., Penzenstadler, B., and Hochstein, L. (2023). Recent research into infrastructure as code. *IEEE Softw.*, 40(1):86–88.

Tatineni, S. and Chakilam, N. V. (2024). Integrating artificial intelligence with devops for intelligent infrastructure management: Optimizing resource allocation and performance in cloud-native applications. *Jrnl of Bioinformatics and AI*, 4(1):109–142.