UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Software Engineering Curriculum

Kaarel Loide

Analysis of Practices for Large Scale Configuration Validation - A Case Study

Master's Thesis (30 ECTS)

Supervisor(s): Pelle Jakovits, PhD
Jevgeni Demidov, BSc

Tartu 2022

Analysis of Practices for Large Scale Configuration Validation - A Case Study

Abstract:
The move towards containerisation and microservices architecture by software companies has dramatically increased the complexity of configuring modern software systems. The increase in complexity also brings a rise in misconfigurations that contribute heavily to system failures. The DevOps movement amplifies this problem by insisting that developers should own the whole software development lifecycle. Developers now have to configure complex systems while usually not understanding the impact of their decisions. A way to reduce misconfigurations is to use automatic configuration validation software. This software can be easily misused and hard to keep up to date, causing frustration for the end-users and maintainers. This thesis looks into what principles should organisations follow when using automatic configuration validation on a large scale. A real case of a software company with about 400 developers is analysed. The author brings out the problems identified from the analysis and offers solutions for each. Based on the offered solutions, they lead an effort to improve the existing tooling and processes around automatic configuration validation. After the project's conclusion, the analysis of the results shows that the efforts successfully reduced the maintenance effort and complaints made about the configuration validation tooling and processes. As a final contribution, the author proposes eight best practices that other organisations could follow to improve their usage of automatic configuration validation.

Keywords:
Configuration validation, fault detection, misconfiguration, continuous integration

CERCS: P170 Computer science, numerical analysis, systems, control

Suremahulise Konfiguratsioonide Valideerimise Tavade Analüüs - Juhtumiuuring

Lühikokkuvõte:
Tarkvaraettevõtete liikumine konteineriseerimise ja mikroteenuste arhitektuuri poole on järsult suurendanud kaasaegsete tarkvarasüsteemide konfigureerimise keerukust. Keerukuse suurenemine toob kaasa ka vigaste konfiguratsioonide tõusu, mis põhjustavad suurel määral süsteemitõrkeid. DevOps liikumine võimendab seda probleemi, nõudes, et arendajad omaksid kogu tarkvaraarenduse elutsükklit. Arendajad peavad nüüd konfigureerima keerukaid süsteeme, kuid reeglina ei mõista nad enda otsuste tagajärgi. Üks viis vale-konfiguratsioonide vähendamiseks on kasutada automaatset konfiguratsiooni valideerimist. Automaatset konfiguratiooni valideerimise tarkvara on aga lihtne väärkasutada ja seda on raske ajakohasena hoida, põhjustades lõppkasutajatele ja tarkvara haldajale frustratsiooni. Käesolevas lõputöös uuritakse, milliseid põhimõtteid peaksid organisatsioonid järgima automaatse konfiguratsiooni valideerimise laiaulatuslikul kasutamisel. Analüüsitakse reaalset juhtumit umbes 400 arendajaga tarkvarafirmast. Autor toob välja analüüsis käigus leitud probleemid ja pakub igaühele lahenduse. Pakutavate lahenduste põhjal täiustatakse olemasolevaid tööriistu ja protsesse automaatse konfiguratsiooni valideerimise ümber. Pärast projekti lõppu näitas tulemuste analüüs, et tehtud töö vähendas edukalt hooldustööde keerukust ning konfiguratsiooni valideerimise tööriistade ja protsesside kohta esitatud kaebusi. Kokkuvõtteks pakub autor välja kaheksa head tava, mida teised organisatsioonid võiksid järgida, et parandada automaatse konfiguratsiooni valideerimise rakendamist.

Võtmesõnad:
Konfiguratsiooni valideerimine, veatuvastus, vale-konfigoratsioon, pidevintegratsioon

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Contents

1 Introduction

Misconfigurations are a significant cause of failures in modern software systems [1, 2]. As more and more companies move towards cloud-based systems and microservice architecture, their application's complexity and the number of moving parts increases. The growing complexity means that the components needed to successfully operate a software system are ever-increasing. Since most companies can not afford to develop and maintain components like caches, message brokers or databases in-house, they need to use various third-party solutions and make them work with each other. Each of these tools like Apache Kafka [3] or Redis [4] requires tuning hundreds of configuration parameters to work effectively [5]. The configuration parameters need to be maintained for each service in a microservice architecture. In the context of larger companies, each of these services is often deployed to multiple regions that all need slightly different configurations. All of this causes the complexity of configuration management to skyrocket. It is not feasible for a single DevOps team to maintain this many configurations, so the responsibility falls into the hands of the developers of the services.

The practice of handing off configuration management to the developers introduces a new problem. Developers know their services, but often, they do not know how to set up the supporting tooling and configure the infrastructure to run them effectively. The use of automatic deployment processes further amplifies the problem. Since developers maintain the configurations, storing them next to their application code also makes sense. Deployment tooling can then pick up the configurations from the service's version control repository and automatically deliver them to different environments. Automated deployments combined with maintaining configurations for unfamiliar tooling becomes dangerous quickly. A simple mistake can be propagated into the live environment in a matter of minutes and compromise the entire application. In addition to the catastrophic scenarios, configuration mistakes can lead to other issues that might be less serious but still problematic in various ways. For example, a misconfiguration can cause the build or deployment pipeline to fail. If the CI/CD (continuous integration and continuous deployment) flow does its job correctly, then this will not cause any harm to the application. However, it will break the developer's workflow. As they send their new changes down the pipeline, they might already start working on a new problem while waiting for the deployment to propagate into different environments. This process might take several hours in some cases, so it is not feasible to wait and do nothing in the meantime. In the case of a failure in one of the pipeline steps, the developer will be required to switch context back to the previous task, often causing them to lose focus. As described by Abad et al. [6], such interruptions can cause a heavy cognitive load for the developers and severely hinder their performance and productivity. An even worse case is when a misconfigured service manages to pass through the pipeline. If an issue is not detected right from the start, it might go unnoticed for a while. A misbehaving service could stealthily cause problems like data loss or decreased performance.

Security issues are also a frequent consequence of a configuration error. For example, it is very straightforward to accidentally expose an internal backend service in Kubernetes [7] to the web and allow anyone to have access to it. An even easier potential security hole in Docker-based systems is the mistake of not configuring the container to use an unprivileged user, making privilege escalation attacks easy [8].

A compromise between having developers or operations handle the configurations is to introduce an automatic configuration validation step into the CI/CD process. Often this is not straightforward as companies have very different setups that require various additional policies enforced in addition to standard best practices. Huang et al. [9] mentioned that configuration validation code is often imperative and riddled with implementation details. In addition, the solutions put in place are ad hoc because company-specific policies usually have no ready-made validators and need to be enforced fast as they are often proactive. Such ad hoc solutions tend to build up and become a maintainability nightmare quickly.

This thesis aims to understand better how to use large-scale automatic configuration validation effectively without hindering the developers' workflow or causing unnecessary load on the DevOps team. The context for the research is one specific large software company struggling with maintaining an ad hoc configuration validation solution. The author will first look into state of the art for configuration validation tooling. They will analyse the problems with the existing tooling by gathering data from the developers historic feedback, studying the rules enforced and having discussions with the maintainers of the tool. They will then propose and lead the implementation of a solution for the company, addressing problems found by the analysis. As a final step, they will compare different vital aspects of the new solution to the old one to understand if the situation improved. The research question (RQ) driving the thesis is the following.

RQ: What guidelines should DevOps teams follow when using large-scale automatic configuration validation?

The RQ will be answered by first analysing state of the art in configuration validation tooling. After which, the author will describe how they created an improvement plan and designed a new solution based on it in a specific company setting. Finally, they will look back on the process and analyse the results to propose best practices aimed at helping future adopters in their work.

The rest of the thesis is structured as follows. Section 2 briefly introduces concepts necessary to understand the rest of the thesis and describes the findings of peers. Next, section 3 describes the company under observation. The author analyses the existing tooling and gathers feedback from developers and DevOps engineers, after which they set objectives for the new solution. Section 4 proposes a solution and gives an overview of how it addresses the set objectives. After the solution proposal, section 5 analyses the results and proposes best practices. Finally section 6 concludes the thesis.

2 Background

In the next section the author will first briefly introduce concepts that are necessary for the reader to understand the later parts of the thesis. Furthermore they will describe the findings of peers in similar areas and analyse their findings in the context of this thesis.

2.1 Continuous Integration

A continuous integration (CI) flow generally starts from a pull requests (PR). A pull request is a way for a developer to propose changes to a larger codebase using a distributed version control system. PRs are very common in modern software development workflows that involve multiple people collaborating on the same project. A PR typically occurs when a developer creates their branch or fork from the base repository and continues the development of a feature without affecting the main codebase. When they finish working on the feature, they make a PR to the owner of the base repository by requesting a review of their work. Other collaborators of the project can then leave comments and requests on the code submitted and, based on the validity of the code, either accept it to be merged, request some changes to be made or reject the PR. This way of working helps avoid frequent merge conflicts and keeps the developer focused on the scope of their problem. Development teams using PR based workflows often have different automation or tools set up to help them more easily validate the changes proposed in a PR. The most common methods for this are using static analysis tools and linters, automatically building the code, and executing testing suites against the newly made changes. Teams usually achieve this automation by creating a Continuous Integration process supported by most modern version control hosting platforms. In the context of this thesis, PRs play an important role as they are the main trigger for configuration validation and also the location to provide feedback from tooling to the developers.

The practice of CI means that developers merge their changes into the base repository as frequently as possible, often even multiple times a day. A CI process aims to avoid a forked code branch diverging too far from the base. A CI workflow typically contains the following steps.

- **Local testing and linting** - When a developer finishes their work, they run unit tests against the new version of the project before pushing to the central server to catch more superficial issues identified with just local unit tests. Developers also often use linters and static analysis tools locally. Static analysis helps quickly identify common mistakes and bad practices. Linters help detect style issues and also enforce a common code style across the whole project so the developer and later the reviewer or the code can both focus on the purpose of the code not style preferences. Executing these simpler validations on the local workstation makes the feedback loop much faster for the developer.

- **Remote builds** - As a next step, when the developer is sure that their code meets all the standards and passes the local testing phase, they send it to the remote version control server, where the automation takes over. A build server makes a clone of the remote repository, compiles the code and builds a final artefact.
- **Remote tests** - In most cases, build servers also execute the same tests and checks the developer can run locally as an extra validation to catch any human errors and run tests in a clean setting to avoid cases where something only works in a local environment. A more extensive testing phase accompanies this by executing additional test suites like integration or performance tests that the developers can not run on their local workstations.

After a successful CI run, the process moves on to a Continuous Deployment (CD) flow that will then deploy the built and tested artefact into the required environments.

In the context of this thesis, the author feels it is necessary also to briefly describe a specific CI/CD platform: Jenkins mentioned in sections 3 and 4. Jenkins is a well established open-source automation server used for various tasks like building, testing, delivering and deploying software. Pipelines in Jenkins can be described in the Groovy programming language, which makes it possible to automate almost any task. As an example, executing the configuration validation tooling described later is also done with Jenkins.

2.2 Infrastructure As Code

Infrastructure As Code (IaC) is a technique that aims to use best practices from software development to improve infrastructure automation [10]. IaC uses automation to provision and configure IT infrastructure based on infrastructure specification and configuration stored as code files. The only thing modifying the actual state is automation. Letting automation manage the actual state based on the desired state stored as code helps mitigate configuration drift and reduces the risk and effort of changing and maintaining infrastructure.

With technologies like Kubernetes, the DevOps team sometimes hands off maintaining the virtual infrastructure to the developers by making them describe the needed manifest files for their services. Unfortunately, the developers are not always interested or knowledgeable enough to understand what consequences a configuration change might have to the building or running of their application. Even if the developer knows how to configure. their application for a simple setup, they now need to also understand how to express this configuration in terms of a complex containerisation technology.

2.3 Containerisation

Containerisation in software development is the packaging of code and the operating system packages required to run the code as a lightweight standalone unit (container) that will consistently run on any infrastructure [11]. In modern times containers themselves are not enough, more complex architectures also require a system to manage and orchestrate the work of the containerised application. These systems are called container orchestration tools, the most notable of which is Kubernetes. Kubernetes manages applications spanning multiple containers by scheduling them on different cluster nodes, scaling them and managing their health by restarting failed services in case of errors. It also manages all of the surrounding parts of the system like load balancing, networking and security. Kubernetes utilises IaC practices by storing application and virtual infrastructure configurations as YAML files called manifests. Kubernetes manifests are usually located next to the application code making them accessible to the developers. Given the complexity of an container orchestration system, it is no surprise that the average developer is not knowledgeable enough to manage these manifests. As a possible aid for the developers, configuration validation can be used to enforce some best practices and provide guidance.

2.4 Microservices Architecture

Microservices architecture is proposed to deal with various problems with monolithic systems. Most notably, microservices architecture aims to reduce the maintenance complexity of a system, help avoid dependency compatibility issues, reduce system downtime and help with scalability. In their paper, Dragoni et al. [12] define a microservice as a cohesive and independent process. They say that when all modules of an application are such processes and communicate with each other via messages, the application uses a microservices architecture. Furthermore, they expand that microservices architecture solves the various problems of monoliths by limiting the number of functionalities each service implements, making it possible to gradually transition the system to a new version and independently scale different parts.

One of the caveats of following a microservices architecture is that all of the services have their independent configuration setups. It quickly becomes impossible for a single DevOps team to manage these configurations within a large organisation with tens of teams and hundreds of different microservices.

There are various ways to reduce this complexity. One of the ways is to use a package managment and templating tool like Helm [13] to provide templates for different reusable components. While reducing some management complexity from the DevOps team, it also introduces new configuration files that the developers need to handle. When the management responsibility falls into the hands of the individual service owners, usually misconfiguration issues start needing the help of an outside member to resolve. This

constant need for assistance from the DevOps teams, in turn, reduces the time and energy they have for advancing other tooling and processes that would benefit the development organisation in the long run. A way to mitigate some manual issue resolution is to introduce automatic configuration validation that helps the developer understand and resolve some misconfigurations before the code reaches any build or deployment process. Providing automatic assistance is also essential for fully allowing the developers to own the software development lifecycle.

2.5 Related Work

The question of best practices for configuration validation seems not to be widely investigated by many authors as there seems to be a lack of more recent work in this area. However, nowadays, developers are sometimes responsible for describing IaC like Kubernetes manifests or Helm charts and templates, so the author figured that looking at work done about best practices and code smell detection in IaC might be helpful for this study.

The most relevant work in the configuration validation area is an article by Huang et al. [9]. In their paper, the authors from Microsoft describe a configuration validation framework called ConfValley, which tries to address some of the concerns about why the previous approach for configuration validation was not working out. Unfortunately, as the paper predates some more recent tooling options, the solution proposal made by the authors might not be the most optimal anymore. Some more recent tooling, like Open Policy Agent (OPA) [14], solves similar problems while being more advanced and open source.

In the article the authors create another domain-specific language (DSL) that must be maintained and extended to new validation requirements. The authors themselves also note that while the systematic approach proposed is indeed helpful, the current state of the tool is relatively limited. Nevertheless, their concerns are still valid and very similar to those described in sections 3, 4 and 5. Some key points mentioned are that configuration validation code is often ad hoc and bulky, which in turn causes it to be hard to maintain. The authors also note that using a different language for the rules helps keep implementation details out of rule descriptions. A DSL makes rules easier to read and helps the implementer focus on the validation details instead of worrying about the technical details of how to retrieve some values from a file for example. Another valid point they raise is that configuration validation cannot reject all invalid configurations in most cases and thus should not be the primary gatekeeping mechanism. Instead, it should be used with other validation methods like deployment testing.

A more recent article by Tianyin Xu and Owolabi Legunsen looks at configuration testing more thoroughly [15]. They propose that configuration validation is too disconnected from code and can not catch most errors. Instead, the authors believe that configuration should be tested together with the application code. The authors of

ConfValley raise a contradicting point to deployment testing [9]. They mention that testing every configuration change becomes costly at the cloud scale.

Furthermore, the previously mentioned point raised by Tianyin Xu and Owolabi Legunsen saying that configuration validation can not detect most issues is also somewhat dismissed by ConfValley authors. They describe that dormant issues could pass deployment testing as well and appear later in the production environments. This problem is amplified because large-scale companies usually have multiple regions, requiring slightly different configuration values, and trickier issues might not surface in all contexts.

A valid concern Tianyin Xu and Owolabi Legunsen raise is that while academia is fascinated with learning-based methods, they are no magic solutions to every problem. In the case of deployment testing or configuration validation, practitioners rarely use learning-based approaches in production environments. The problems brought out by the authors are that outlier detection is not a suitable solution as misconfigurations are not always outliers and vice versa. Furthermore, there is a lack of large configuration datasets, making training difficult.

To see if there are any lessons to be learned from a somewhat related field with more available completed research, the author also looked at works related to best practice enforcement and code smell detection in IaC. To get a better understanding of the best practices used, Guerriero et al. interviewed practitioners from the industry, and their data revealed that 8-10 tools are about equal in their usage in the IaC space [16]. The findings point out that there is no best tool established yet. Users have said that each tool has its strengths and weaknesses, meaning that the right choice of tooling is a matter of determining the most suitable solution for the problem at hand. Their research also discovered that although several best practices emerged from the interviews, they differ in the applicability based on the technology.

Even though the work of Guerriero et al. shows that code smells and best practices are still not that well developed, there also exists contradictory evidence. Multiple works point to the fact that the general approach of code smells can in fact be applied to IaC templates as well [17, 18, 19, 20, 21]. The findings of Schwarz et al. indicate the fact that IaC smells are, in most cases, technology agnostic and can be defined independently of the underlying tool [20].

As for detecting faults and code smells, the maturity problem arises once again. The field is still in the early stages of development, and there is a large number of different tools being used with no clear all-around winner.

Two central problems seem to contribute to this lack of tools capable of detecting faults in IaC. First is the diversity of languages used by the tools and the fact that often multiple languages are interwoven in a single tool. Secondly, there is a lack of standard best practices established as the area is still relatively young, making the development of tools complicated as there is no joint knowledge base for what issues to look for. There have been some attempts at creating such tools, but they have not become well adopted or are still in the prototype phase.

For example, Ting Dai et al. [17] from IBM created SecureCode in Python. They try to address the issue of arbitrary script invocations in IaC by combining existing script analysers like ShellCheck and PSScriptAnalyzer into a framework that detects and prioritises faults found in scripts embedded into Ansible files. They also have plans to extend SecureCode to cover many IaC tools by adapting it for Terraform as a next step. The created framework also provides a way to plug in different analysers for the injected scripting languages. If some IaC tool provides a way to inject Python code, for example, this could be solved by plugging in a Python analyser to compliment ShellCheck and PSScriptAnalyser. As a case study, this tool was also introduced into the DevOps pipelines at IBM to test 45 of their community service repositories.

Borovits et al. also created a tool for Ansible scripts called DeepIaC. They attempted to use a deep learning-based approach to detect linguistic anti-patterns in Ansible scripts and achieved an accuracy of between 0.785 and 0.915. Currently, DeepIaC helps to debug inconsistencies in the names and bodies of Ansible code. However, there is also a plan to extend DeepIaC to detect a broader range of bugs and work with other IaC languages [18].

Kumara et al. tried taking a semantic approach to the problem of smell detection in IaC. They created a prototype implementation for detecting faults in TOSCA templates. The created tool uses GraphDB as a knowledge base and SPARQL queries to identify smells in the TOSCA files uploaded by the user. The authors validated the results through three industrial case studies of the SODALITE project. Existing TOSCA files were modified to contain smells presented in the paper and validated with the defect predictor to verify that each smell can be detected [19].

A joint takeaway from both investigated research areas is a lack of well established best practices and tooling. While there are advancements and experimentation done to develop new tooling, most of it seems to be in relatively early stages of development and not ready to be adopted by the industry yet. There are still valuable lessons and ideas in the examined research that the author found useful and will keep in mind while performing the case study.

Two points resonated most with the author. The first is that configuration validation code is often bulky and ad hoc, similar to what they observed in section 3. The second is that configuration validation code should be separate from implementation logic to help avoid the first point.

3 The Case

The following section will give an overview of the company whose needs are the primary motivation for this thesis. The author will describe the identification process of existing problems by analysing the existing configuration validation tooling looking into the rules, and gathering the opinion of the developers and the DevOps team of the company. Finally, the section will conclude by making some speculations about why some problems might have emerged and setting objectives for the new solution to address the identified problems.

3.1 The Company

The company that is the primary beneficiary for the solutions developed with the help of this thesis is an international cloud-based software as a service (SaaS) provider that offers a customer relationship management (CRM) tool for small to medium-sized businesses. The company's development team has around 400 members and is growing fast.

The company uses microservice architecture for its product and each microservice has a separate Git repository under the company's private organisation in GitHub. Given that the company has been active for over ten years and technologies and best practices have developed a lot over that time, there is a wide range of different setups and frameworks used in about 1400 repositories. Even though the setups might differ, there are a few common denominators across all the repositories and teams.

First, the service must automatically end up as a containerised application running in a Kubernetes cluster. The build and deployment should happen using Dockerfiles and Helm templates that the automation, built by the DevOps tooling team, can use to transform the source code into a working service in the production Kubernetes cluster.

Second, each team follows the GitHub flow [22]. Following the GitHub flow means that when work on a new feature begins, the developer creates a separate branch from the repository's main branch. The developer works on the changes in the created branch until they are ready to be incorporated into the main branch again. When the developer completes a feature, they create a PR, requesting the team to review code changes before pulling their work into the main codebase. The creation of a PR is also where the CI process starts. A Jenkins job and a GitHub Actions workflow trigger, and the new code is built, validated and tested. Two separate automation tools are needed as GitHub Actions are relatively new for the company, and all of the services have not migrated. In addition, some features developed by the DevOps tooling team are not possible to achieve with GitHub Actions alone and still require Jenkins. One of these features is configuration validation which is the main focus of this thesis.

3.2 Previous Tooling

The tool previously used for configuration validation was an in-house solution called "Deployment or repository analyzer" (Dora) [23]. Dora is a CLI-based tool that initially had the purpose of analysing Docker and docker-compose files and validating the use of best practices and the companies internal policies. Over time Dora has drifted from its original purpose as it became a catch-all place to introduce any rule that the developers must follow. In addition to being a tool for validation, it also became a gatekeeper in the deployment pipeline, where each execution was context-dependent and time-sensitive, causing inconsistent results. Before, a typical deployment triggered Dora in three different contexts and up to ten times. The results were not guaranteed to be the same for each execution, even though the set of rules enforced was static because the rules themselves were not.

The first execution happened on a push to the GitHub repository. This execution aimed to help developers fix issues before their code reached the build and deployment process. PR time validation also blocked faulty code from reaching the main branch of the repository.

The subsequent execution happened in the build pipeline. If developers bypassed PR time validation Dora blocked misconfigured code from being built. The build time validations included things like requiring branch protection on the main branch and having at least one approving review on a pull request.

The final executions happened in the deployment pipeline. This final execution was done for deployments to each region as a final safeguard. Checking deployments to each region resulted in the same configuration being validated multiple times in various context, as the company uses multi-region architecture to provide the best experience for customers across the globe.

3.3 Initial Analysis

This subsection will describe how the DevOps team understood Dora as a problematic tool. It will then detail the process the author went through to identify concrete problems with the existing solution. The analysis will be done both from the developer's view, who needs to deal with the tool and its outputs daily, and the DevOps engineer's view who needs to maintain and utilise Dora to educate developers and block faulty configurations from reaching production.

3.3.1 The Developers View

In recent times, Dora had been a significant subject of complaints in the company's public DevOps Slack channel. The team has noticed this trend of complaints based on plain observation of the chat logs.

To better understand the problems and trends, the author created a script to scrape the history of the previously mentioned Slack channel. They then used the script to extract the channel's history spanning two months before developing the new solution.

As a result, they gathered 592 messages and filtered them based on whether they contained the name of the tool "Dora" or not. After filtering, 65 messages about Dora remained. The author then analysed and categorised these messages to better understand the major pain points and complaints. In Figure 1 we can see the exact number of messages in each category the author managed to recognize from the data. Categories marked with red are actual issues while blue are false positives in the context of this analysis. After excluding the messages marked in blue we get a total of 53 complaints. The author will now elaborate on each of the categories to give a better understanding about each.

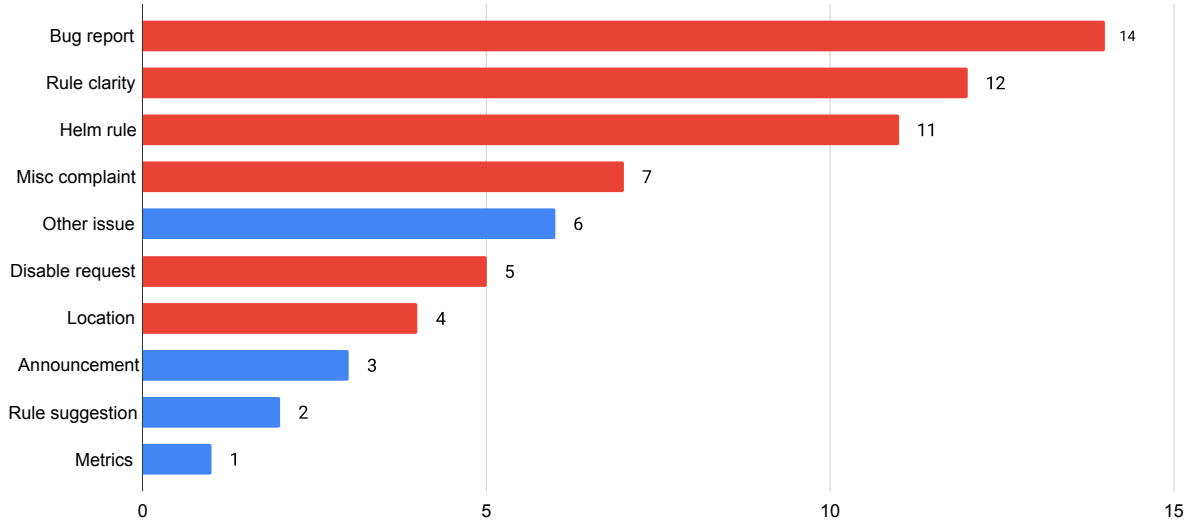

Figure 1. Types of Slack messages about Dora

Bug report (14)
Messages about a bug in Dora. Bugs in the CLI or in some rule implementation are included in this category. Based on the limited context of the Slack message and the implementation of Dora, it is impossible to tell where the bug might be in the internals of the CLI or the rule code.

Rule clarity (12)
Developers complaining or asking questions about rules because they did not understand what Dora was trying to tell them to do.

Helm rule (11)
Complaints about a specific rule causing great frustration to many developers. The rule validated if memory and CPU resource requests and limits were specified between the optimal range for each deployment region in the Helm manifest file. The optimal value was calculated in real-time based on Prometheus [24] metrics and changed frequently. This value could have even changed between the multiple validations executed against a single deployment as deployments to secondary datacenter regions happened multiple hours later than the first validation done at PR time. The complaints about this rule were both about the frequently changing nature of the correct value and the fact that the developer had to change something with their PR that was unrelated to the actual thing they were implementing.

Misc complaint (7)
Different complaints or issues that did not fit into any other larger category but could be identified as issues related to Dora.

Other issue (6)
Messages mentioning Dora but where the actual complaint was not about the tool or any of the rules. These complaints were about some other tooling or service that Dora was either executed by or that Dora relied on to perform some validation or reporting. As an example developers were requesting help configuring Sonarqube because they got notified by Dora or reporting that something might be wrong with Jenkins as the pipeline used for Dora was logging network issues.

Disable request (5)
Requests to disable Dora validation for some specific case. For example, the requests to disable Dora were cases where some developers wanted to deploy a hot-fix, but a failing rule stopped them or where the validation did not make sense to them, for example, on draft PRs.

Location (4)
Messages somewhat related to the Helm rule cases. These were complaints about the location of Dora execution. More specifically about it being executed before deployments to every region.

Announcement (3)
Different announcements made by the DevOps team about changes to Dora. Changes were things like introducing a new rule or alterations to some existing ones.

Rule suggestion (2)
Suggestions people had for new rules that could be included in Dora.

Metrics (1)
A single message about a person requesting metrics about how many repositories are not deployable due to violations in some rules.

3.3.2 The DevOps Teams View

In addition to the developers' feedback, the worries and issues the DevOps team had with the tool were gathered in an open forum discussion. Every team member could express concerns and observations they have gathered over time while working with Dora.
From the discussion, four main issues stood out to the author.

- **Testing:** The source code of the tool itself lacked test coverage at 52.9% and was, in general, not easily testable. The rules themselves were also not testable as individual units. Functional tests were testing all the rules together but there was no way to validate the behaviour of each rule individually.
- **Validation of new rules:** There was no way to understand the impact of a rule before including it into the suite of existing rules. Before introducing a new rule, the creator had no way to see how many repositories would start getting failing results on their PRs immediately as they were already violating the new rule.
- **Code complexity:** The tool's source code was relatively hard to grasp. A multitude of different factors caused the complexity. The internals and the rules were both developed using Javascript. Using the same dynamic language for both allowed the maintainers of Dora to mix the logic of rules with the internal logic for parsing inputs and reporting the results. The mixing of concerns caused the rules to be tough to understand and made introducing bugs extremely easy.
- **Lack of metrics and observability:** Dora had minimal metrics. The only way to observe Dora was to either analyse build logs with a short retention period or look at a dedicated Slack channel where the tool reported all failures. Additional feedback from the team was that it was challenging to understand where and how automation was executing Dora. The only reasonable way to identify problems was through developers' feedback, which is already too late in most cases. It made the developers break their workflow to understand why something was faulty in their PR even though the fault was actually within Dora implementation in most cases.

3.3.3 Rules

At first, most of the rules enforced Docker best practices and tried to prevent security issues as this was the original purpose of Dora. Over time, general best-practice enforcement rules became less relevant as developers fixed the existing issues and company-wide templates became widely adopted for new services. The adoption of a template meant developers did not have to deal with writing Docker configurations themselves as the default configuration suited most use cases. The company also moved from Docker Swarm to Kubernetes, meaning that all rules related to docker-compose files became obsolete. During this shift, the purpose of Dora also slowly evolved to be more of a way to enforce company-specific rules instead of general best practices. The new rules that emerged over time mostly fell into three categories.

Company-specific configuration enforcement: Rules that make sure the repository configuration is suitable for other internal tooling. For example, each repository needs a valid owner, a correct type and what regions it can be deployed to. This configuration must be done by the developer and each configuration value has a set of rules it needs to adheer to.

After the fact preventative validation: Such rules are created after some incident that was caused by a misconfiguration. If the cause is something the developer has to deal with and is likely to happen again, then a rule was written to prevent future cases. One example would be a rule validating that each service that uses a Javascript dependency called Webpack has a specific configuration value: "jsonpFunction" set. Without this value there were multiple cases where some parts of the application failed to load.

New practice enforcement: As the company evolves, the developers need to start using new tooling. It is impossible to have every single developer start using a new tool out of their free will, especially if it is something that makes their life more difficult in the beginning. Even though developers are usually not keen on such change, it is necessary. Some examples of this type of change would be the adoption of Helm instead of docker-compose for migrating to Kubernetes or validating if repositories have Sonarqube analysis set up correctly.

The biggest problem with some of the new rules was that they were trying to do too much. Dora was the place to introduce any new requirement that the developers needed to follow, even if it did not make much sense to notify them about it on each PR or deployment step. One example of this type of rule would be the one described in the Helm rule paragraph in section 3.3.1.

3.3.4 Conclusion

Dora was used for validating configurations and blocking unvalidated builds and deployments. A tool initially meant for configuration validation became a gatekeeper in the deployment flow; instead, the CI/CD pipeline should handle these cases as it is more aware of the context it is operating in. Automation executed Dora in multiple contexts and at different times for each PR, build and deployment. Dora would often run successfully on a PR but later fail in the building or deployment phase because some dynamic value from an outside source would change when the new version of code reached the next step in the pipeline. The inconsistency of results caused significant frustration for the developers. They thought everything was okay with their changes to find out a few hours later that deployment to the final region failed because some change in the region's resource usage caused a rule to start failing.

To address the complaints of the developers and comply with the the initial analysis and requirements set by other company parties, the following objectives were set for the new implementation.

- Remove or rework problematic rules.
- Review and reconsider location of execution for Dora.
- Design the new solution to allow for unit testing of rules.
- Expose easily consumable metrics for the tool.
- Create a way to bulk execute the tool for every repository in the organisation.
- Provide an easy to follow output format for the developers.

As a first step in satisfying the objectives, the author mapped out the exact requirements and looked into existing tooling for configuration validation to evaluate if a suitable solution is already available on the market.

4 Solution Proposal

This section will detail the author's proposal for a technical solution to the company's problem. The section consists of five parts, section 4.1 will describe the requirements set by the previous tooling and improvement ideas. Section 4.2 will describe existing solutions and how they meet the requirements. After which section 4.3 will reason about the choice of technology and describe how the proposed stack helps meet some of the requirements. Section 4.4 will describe the architecture of the new CLI and detail how each requirement for the tool was satisfied. And finally sections 4.5 will go into detail about all the supporting tooling required to meet the rest of the objectives set in section 3.3.4.

4.1 Requirements

The first significant set of requirements that quickly became apparent from technical discussions with the team was that if we create a new tool, it must support all the existing capabilities of Dora.

To understand the requirements the author analysed all of the 70 rules implemented in Dora. The set of requirements that the author extracted by this analysis was the following.

Parsers
The new tool needs to support parsers for JSON, YAML, Ignore Files, Dockerfiles, plain text and properties files as these were the types of files Dora was already scanning.

HTTP requests capabilities
Some existing policies made HTTP requests to fetch dynamic information from external APIs during policy evaluation. As an example these policies were validating the configuration of a GitHub repository or requesting some additional input from an API to make sure a value in a configuration file is correct.

Safe secret injection
A sub-requirement for HTTP requests is the safe injection of secrets like API tokens into the tool to call any API that requires authentication.

Comparison of multiple input files
It is often the case where a specific value in one configuration determines the required value in another. This interdependency means that the tooling must load multiple input files into the same evaluation context.

Repository based rule overrides
As the company has about 1400 repositories with a wide range of specific configuration setups, it is sometimes the case that a particular rule does not apply to a repository. To support this case, Dora had the functionality to override some properties of the defined policies. Dora supported this by looking for a configuration file called ".dora.json" in the root of each repository. In this file, the repository owner could configure some predefined values of a policy to be overridden or disable a policy altogether. Since this file was present in over 300 repositories, the new tooling needed to be backwards compatible to support it.

Handling of missing files
A missing file can mean a misconfiguration or a non-issue based on the context of the repository; this was done case by case for each policy in Dora and based on the analysis of developer feedback was a frequent cause of inconsistent results or bugs in the execution.

Configuration merging
Some tooling like Helm, for example, allows the user to define multiple configuration files that it will merge into one. As Helm configuration validation makes up a significant part of the policies defined by the company and most of this validation is done over combined value files, it is a crucial functionality needed in the new tool.

The second set of requirements were formed from the various improvement ideas gathered from different parties.

Separation of concerns
Similarly to the problems described by Huang et al. [9]. A significant problem with the maintainability of Dora was the ad hoc nature of code and rule logic tightly coupled with implementation details. The new tool needed to have a clear separation between the validation logic and the implementation details of the CLI to address the concerns of maintainability.

Internal testability
The tightly coupled nature of Dora code also caused its internals to be hard to test. The overall test coverage of Dora was 52.9%. The new solution was set to have code coverage requirement for the CLI as 80% for each new PR made to keep up code quality and force a testable architecture.

Rule testing
The previous implementation had a suite of functional tests to verify rules. This suite required the developer to use a single mock repository to describe a failing set of

configurations and a single mock repository to define a set of passing configurations. Dora was then executed against these mock repositories to verify the validity of each rule by checking that the failing repository had a failure for each rule and the passing repository had no failures reported. While this provided some validation, it quickly became hard to follow as mock data for different configurations is clumped into the same repository. It also makes testing different failure cases for a single rule impossible as there is only a single input for each execution. As long as the test suite finds one failure case, the tests will pass. Furthermore, this test suite provided no coverage information, so it was difficult to understand the big picture and enforce any quality standards on the rules. Each rule needed to be testable in isolation with mockable input and precise output validation for the new implementation. In addition, it was required for the tests to provide coverage information.

Output
Previously Dora provided two main output formats. First was a human-readable, text-based output for each failure detected and second was a custom HTML report sent to GitHub to provide feedback to the developers on PRs.

For the new solution a JSON output was required to help integration with other tooling. This new output format also needs to support more extensive details about each execution, including successful and skipped rules, to ease integration with metrics gathering and observability services. In addition, the developers often complained that the GitHub report was too extensive and hard to follow. A third party from the company created a re-designed GitHub report that the new CLI needed to implement.

4.2 Existing Solutions

In the following section, the author will describe different existing solutions. They will bring out some of the more notable open-source alternatives mentioned in grey literature for detecting faults and misconfigurations in IaC and various configuration files. The author will list the key features and shortcomings for each tool and summarise if any of them seem useful in the context of the thesis.

4.2.1 Checkov

Checkov is a Python-based static code analysis tool for IaC developed by BridgeCrew [25]. Checkov scans files for misconfigurations and security issues based on predefined policies. The tool has more than 750 predefined policies and supports creating custom ones using either YAML or Python. The maintainers also accept custom policies as contributions to the official policy base.

The tool creates a cloud resource connection graph internally to achieve a more profound analysis. Checkov then uses the created graph to find misconfigurations across resource relationships.

Currently, Checkov offers support for seven different types of IaC tools and frameworks: Terraform (for AWS, GCP, Azure and OCI), CloudFormation (including AWS SAM), Azure Resource Manager (ARM), Serverless framework, Helm, Kubernetes and Docker. The tool supports suppressing individual checks with special comments in the configuration files. At the time of writing the tool is in active development and has usable documentation.

While providing some essential features it is lacking in others. Things like rule testing, HTTP requests, dynamic output and file based input are not supported out of the box. Furthermore the current implementation seems too rigid to allow for sufficient modification in order to satisfy all requirements.

4.2.2 config-lint

Config-lint is a CLI tool written in Go that validates configuration files based on rules written in YAML [26]. It supports scanning configurations for Terraform JSON and YAML. The tool offers built-it rules for Terraform files and also supports custom rules. The YAML syntax supports a list of operators for different comparisons and assertions. In addition, there is a function to request information from HTTP endpoints. This feature seems quite limited. There is no way to specify the HTTP method, headers or body for the request, meaning more complex requests would not be possible. Not providing support for authentication also limits the usage quite a lot, as querying data from protected endpoints would be impossible. Looking at the GitHub history of the tool, no one is actively maintaining it. The last commit at the time of writing was on the 26th of June 2020, almost two years ago.

It seems as the tool is more of a prototype then a well made product. The quality of the tool together with it missing many key requirements makes it not the best candidate for adoption.

4.2.3 Copper

Copper is a tool built by Cloud66 to validate Kubernetes configurations [27]. The documentation also mentions that they support other configuration files without specifying precisely what and how. It is a mix of Go and Javascript with the CLI written in Go and rules together with helper functions written in Javascript. The documentation is minimal, consisting of a single page with an elementary example of a Kubernetes manifest validation.

As with config-lint, this project also seems to be an abandoned prototype, with the latest release being from the 5th of December 2019. The same reasons also apply for why it is not a suitable tool for solving the problems in the context of this thesis.

4.2.4 Open Policy Agent

Open Policy Agent (OPA) in contrast is a more general purpose tool for enforcing policies across the entire software stack [14]. OPA is not specifically designed for detecting faults in configuration files or IaC but it can certainly be utilised for this use case. It is actively maintained and used by multiple different tools described later.

OPA offers a high-level declarative language called Rego that is used for defining policies. To illustrate the language, an example of a Rego policy that denies the creation of Kubernetes Deployment objects where containers are run as root can be seen in Figure 2.

```
package main

import data.kubernetes

name = input.metadata.name

deny[msg] {
    kubernetes.is_deployment
    not input.spec.template.spec.securityContext.runAsNonRoot

    msg = sprintf("Containers must not run as root in Deployment %s", [name])
}
```

Figure 2. Policy described in Rego language [28]

Unlike Python or Javascript, Rego is a declarative language meaning that the decision making for each rule can be separated easily from the software using OPA underneath.

The language is an extension of Datalog [29], extending it to support structured documents like JSON. Both OPA and Rego have community support and extensive documentation. OPA provides built-in helper functions for use in Rego policies covering a wide variety of common operations needed for configuration validation and object manipulation.

For enforcing the defined policies, OPA offers multiple ways to integrate the tool into your existing stack. For example, you could deploy OPA as an admission controller in your Kubernetes cluster to deny the creation of misconfigured resources [30]. OPA also offers a CLI to enable validating configuration both in CI as well as locally on the machine of the developer.

Another valuable feature of OPA is the possibility of unit testing each rule by writing a special policy in Rego. The developer can mock all the input data for their rule and evaluate it with the OPA CLI. The CLI also provides coverage information for the unit tests allowing for out-of-the-box quality assurance.

The following tools are all implementations of the OPA engine with some custom functionality built around it to deal with different inputs and expose the results in various outputs, with some also providing built-in rules.

4.2.5 Conftest

Conftest is an extension built on top of OPA and is developed under the same parent project [31]. Similarly to OPA, it is also in active development. Conftest allows the user to write tests against structured configuration data and run these tests using the Conftest CLI. The tool relies on the same Rego language for defining policies. The main difference form OPA itself is that Conftest is an additional layer on top of the OPA CLI. It offers a way to manage policies by pulling them from different online repositories based on Git or OCI, making sharing and managing policies easy across a whole organisation. It also has many built in parsers for most common languages used in IaC tools or common configuration paradigms.

At the time of writing Conftest has support for the following file types: CUE, Dockerfile, EDN, HCL, HCL2, HOCON, Ignore Files, INI, JSON, Jsonnet, TOML, VCL, XML and YAML, providing almost all of the required parsers for the case of this thesis. Even if the tool is lacking in some required functionality the underlying Go library seems like a possible solution candidate that needs some extra functionality built around it regarding input and output flexibility.

4.2.6 Trivy

Trivy is a tool in active development for vulnerability scanning and validating container images, file systems and Git repositories [32]. Even though it is a vulnerability detector first, it can also detect misconfigurations. Out of the box, it provides rules for Kubernetes, Docker and Terraform. Trivy allows the user to specify either the name of a Docker image or a path to some configuration files and execute a suite of checks against them.

Aside from the built-in vulnerability detection, Trivy is similar to Conftest with some additional features and drawbacks. The most notable feature of Trivy is the support for input selectors that allow the writer to define what type of input a rule should validate. Other great additions to the default OPA features are flexible ways of combining input files and rule-based metadata definition support. However, there is one significant drawback to Trivy. As it implements a lot of custom functionality on top of base OPA, the default way of unit testing rules is not possible, and there seems to be no alternative implemented at the time of writing. Another drawback of Trivy is that currently, it supports a minimal set of input files, Dockerfile, HCL, HCL2, JSON, TOML and YAML.

4.2.7 Regula

Regula is another actively maintained OPA based tool [33] Like Trivy, Regula also has a set of built-in rules and the possibility for custom rule definition. In contrast to Trivy, Regula mainly focuses on validating cloud resources. It supports CloudFormation JSON/YAML templates, Terraform HCL code, Terraform JSON plans, Kubernetes YAML manifests and Azure Resource Manage JSON templates.

As a great bonus, Regula looks to be the most well developed out of the three OPA-based tools to support the rule developer. It offers functionality like automatic generation of test inputs and a fully functional repl that supports the features of Regula in addition to base OPA. With Regula, the developer can also define metadata for rules. It also allows for unit testing of the rules while supporting Regula specific features in the tests. While Regula seems like a good candidate overall, it is limited in the supported input types and is fairly opinionated in how it modifies the base Rego language.

4.2.8 Summary

After examining the existing tooling, the author noticed a trend in people adopting OPA and Rego as the basis for different validation needs. Based on the observed trend and experimentation with the tools, OPA and Conftest seemed like promising solution candidates. Table 1 shows the summary of all the compared tooling and how they match the previously described requirements. Table 1 also includes other criteria that were important in the opinion of the author, like the quality of the documentation and if the project is actively maintained or not.

Even though some of the existing OPA implementations seemed promising at first, on further inspection, they lacked some essential features or had a slightly different purpose than what was required. Most notable problems were related to the input and output of the tooling. Based on the examples and documentation, none of the tools had flexible enough input resolution and parsing. The only tool providing a critical functionality, dynamic output, was Trivy, but like most other solutions, it was too limiting in the input flexibility.

Table 1. Existing tool comparison summary

	Checkov	config-lint	Copper	Conftest	Trivy	Regula
Actively maintained	x	-	-	x***		
Well documented	x	-	-	x***		
Required parsers	-	-	-	x***		
HTTP requests	-*	x	-*	x***		
Safe secret injection	-*	-	-*	x***		
Handling missing files	-	-	-	-***		
Configuration merging	-*	-	-*	-***		
Separation of concerns	x	x	x	x***		
Rule testing	-*	-**	-*	x***		
Custom rule support	x	x	x	x***		
Multiple input comparison	-	-	-	-	x	-
Rule overrides	x	-	-	-	-	x
Internal testing	x	x	-	-	x	-
Dynamic output	-	-	-	-	x	-
File based input	-	-	-	-	x	x

* Not included in base functionality but a common programming language is used for policies so the missing functionality could be added.
** Includes some tests for built-in Terraform rules but no documentation about if or how similar tests could be added for custom rules.
*** Common for all three OPA based tools as the functionality is provided by OPA and Rego.

4.3 Choice of Technology

As previously observed, many problems arise in configuration validation because of ad hoc code and the mixing of concerns. These points were kept in mind right from the beginning for the new tooling and were a significant focus when designing the new solution. Considering the requirements, the author proposed that refactoring the old code base of Dora to satisfy all needs would result in the same amount of effort as starting from scratch.

The benefit of starting from scratch was to decouple policy code from CLI implementation details. Due to a large amount of ad hoc code, this would require significant additional effort for the old code base. Instead it would likely be a lot more straightforward if kept in mind right from the design phase.

Based on the analysis of previous tooling the author figured that using Conftest as a Go library in a custom solution would be a good start as Conftest seemed like a good middle ground for the new implementation. It provided solutions for many of the requirements set: a wide variety of parsers, combining multiple inputs, testing of policies,

and separating rules from the implementation by using Rego language to describe policies and Go for everything surrounding it. The author decided that it would be best not to use the CLI of Conftest but instead extend the functionality it offers as a Go package as some key functionality was missing from the CLI but a lot of the internals seemed useful.

After some initial prototyping, it quickly became apparent that only using OPA, the engine behind Conftest, was the appropriate path instead. As seen from the tooling analysis, OPA is already adopted by a large group of developers which shows its reliability and usefulness. In addition to the evaluation engine, OPA offered out-of-the-box tooling for policy development in the form of its own CLI. The primary value of the OPA CLI was the unit-testing framework that also provided coverage reporting. A bonus was the built-in formatter that allowed for the enforcement of a unified style in all policies. The adoption of OPA also solved some of the requirements set during the analysis phase without any significant additional effort.

The programming language of choice for the new tool was Go. The reasoning behind choosing Go was that OPA, the engine for policy evaluation, was also written in Go and provided its internals as Go packages. Go is also well suited for CLI development as it offers easy to use packages like Cobra [34] for creating and managing different commands and flags.

For rule definitions, Rego was the choice as this was the language supported by OPA. Even though the project is relatively young, it already has many adopters, and the project itself is active with multiple new releases each month. The documentation is also easy to use and the tool has active community support.

As Rego is a declarative logic-programming language, it might be daunting for most developers at first because they have not worked with something like this before. Despite the initial scare, Rego is easy to read and write as it offers powerful and simple to use support for referencing deeply nested documents. The declarative nature combined with powerful query tools allows the policy creator and the reader focus on what the policy should evaluate instead of worrying about implementation details like how to fetch some data in a document.

The new CLI project had a code coverage requirement right from the start. The minimum coverage required for any new code written in Go was at 80%, and for Rego, the requirement was at 95%. These requirements aimed to ensure the new solution addresses both testing-related requirements, Internal testability and Policy testing.

As mentioned before, OPA and Rego solved some of the requirements on their own. The requirements solved were:

- **HTTP request capabilities:** Rego provides built-in support for making HTTP requests.
- **Safe secret injection:** OPA exposes environment variables under the data object in Rego.

- **Comparison of multiple input files:** OPA and Rego work with the concept of input and data objects where input is the main file evaluated while data can be any additional files needed.
- **Configuration merging:** another built-in function of Rego is the ability to recursively merge objects by simply passing them to a function called "union_n".
- **Separation of concerns:** by using Rego as the policy language and Go for the engine and CLI, the developer cannot mix implementation details with policy code.

OPA and Rego did not directly address the other requirements set during analysis. However, they were flexible enough that the author could design a wrapper around them that would provide all the missing functionality. The next section will detail how the rest of the requirements are solved by the wrapper.

4.4 Extensions to OPA

Now the author will describe the high-level design of the CLI. After giving the high-level overview, they will also mention how they addressed the requirements not solved by OPA. The architecture of the CLI was designed by the author while the implementation involved two more developers. The author's role in the development phase was to take part in the implementation of all parts of the system and to lead others in order to make sure their work satisfied all of the requirements described in section 4.1. The tool itself or the low-level technical details are not the focus of this study. Rather, the following description provides examples of how they adopted an existing solution like OPA for the configuration validation use case.

The high-level overview of the CLI is in Figure 3. An important detail is that rule definitions are stored in the same Git repository as the CLI itself and embedded directly into the binary on build time. Embedding the rules means that the CLI always has the same rules and does not need to do any additional fetching on evaluation time. The high-level overview uses the OPA logo to illustrate what components it provides. The colour blue indicates the internal components of the CLI. Yellow marks the additional components that needed to be created around OPA, and grey indicates the various external files that are not necessarily part of the CLI, rather they provide the input for the validation engine. The arrows indicate how the evaluation flow starts from the rule parser component and moves through the various parts of the system ending in one of three output formats.

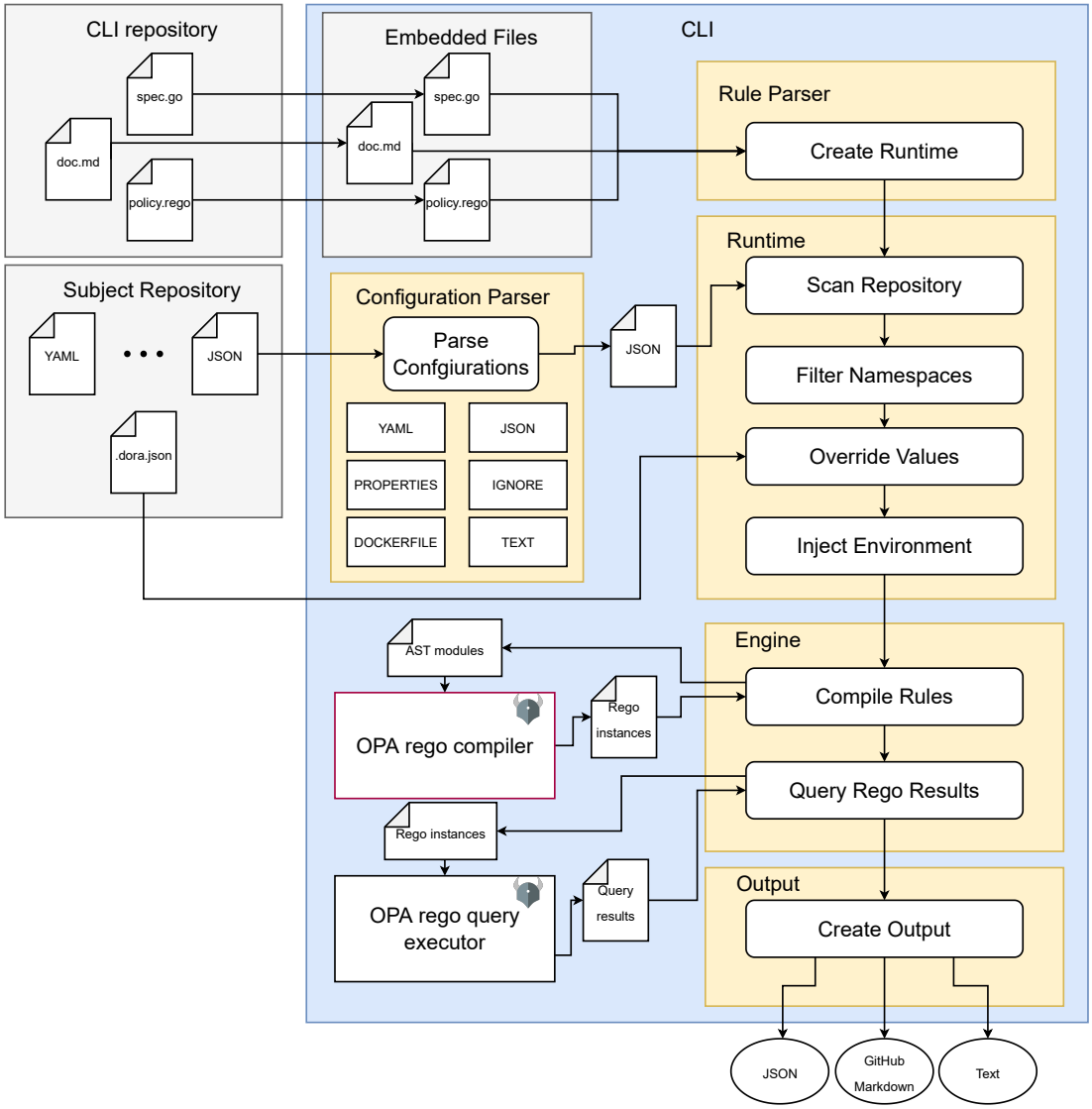

Figure 3. Components of the CLI and the execution flow

A key feature of the wrapper is how each rule is defined. The approach to rule definition is one of the biggest things that sets the solution apart for other existing solutions. A rule definition consists of three files and an additional unit-test file that is not embedded into the CLI. The rule definition starts with declaring the "spec.go" file. In "spec.go", the developer specifies details of the rule. These details include the following:

- **Id:** rule name in the Rego policy file, also used for referencing the rule for external configuration overrides.

- **Level:** Critical or Warning, where Critical failures will cause the CLI to exit with code one and Warning will not, allowing CI to block Critical mistakes.
- **Name:** descriptive name for the rule.
- **Documentation:** reference to the markdown file that describes the exact details and mitigation steps for a rule.
- **Status**: Active or Inactive to allow for disabling of a rule globally.
- **Input:** Glob pattern to specify what files this policy should evaluate. A single rule can have as many input patterns specified as needed. The engine evaluates inputs one by one. During a policy evaluation, the engine does not have access to any other inputs for the same rule.
- **Data:** Additional data files made available for all of the inputs to compare and combine multiple files.
- **Variables:** Custom variables passed to the OPA runtime must be overrideable by external configuration.

After the specification is defined, the developer writes the policy definition in the "policy.rego" file. Here is where the Rego query for evaluating if the repository matches some criteria is located.

The final part of the rule definition is the "doc.md" file. In "doc.md" is where the in-depth documentation for each rule is. This file gets included in a web page published for the developers the allow for a more straightforward resolution of configuration issues by providing detailed descriptions of the rules with examples on how to solve them.

The execution flow of the CLI starts by creating a runtime object that contains all the abstract syntax tree (AST) representations of Rego policies with attached input, data and variables that get resolved to the actual values in the evaluated repository. After the runtime is populated, it gets passed on to the engine, which compiles the Rego policies by passing the AST objects on to the OPA Rego compiler. The CLI engine then passes these compiled Rego policies to the OPA Rego query executor with all the specified input, data and custom variables attached. After the evaluation, the CLI creates an output based on the specified command-line flag resulting in a terminal friendly text-based output or a JSON output meant to be consumed by other tooling. Separate command-line flags toggle additional reporting to a PR in Github and to the metrics server.

To provide a better understanding of what exactly they added to OPA the author mapped out all of the requirements not solved by OPA. The next paragraphs will describe how each missing requirement was solved in the new tooling.

Parsers
The CLI has a common Go interface for implementing parsers for any file type needed. The parser should take a file as input and transform it into a JSON representation for OPA. Currently implemented parsers are for JSON, YAML, dockerfile, ignore file, properties files and plain text.

Safe secret injection
Even though OPA makes environment variables available, there was still a need to dynamically resolve secrets on rule evaluation time for cases where setting secrets to environment variables is not feasible. An example would be the case of executing the CLI on a developers local workstation. Configuring each secret manually would be cumbersome and developers do not usually have access to all the required values. The wrapper solves this by attempting to fetch missing secrets before evaluating rules from Consul [35], a key/value store used by the company.

Repository based rule overrides
The CLI has a parser for extracting overridable variables from a file called ".dora.json" to provide backwards compatibility with the old tool. This file allows people to turn off specific rules for their repositories by overriding the "active" variable of a rule and also enabling them to override some predefined variables in certain rules.

Handling missing files
The engine skips evaluating rules that do not match any input files as default behaviour. This is what other OPA based tooling also usualy does. Unfortunately this behaviour poses a problem when we need to enforce the existence of a particular file. The new tool solves this by allowing no input to be defined in the "spec.go" file. An empty input means that the engine should evaluate the policy every time. In addition to this, the required files should be passed as data. By providing the required file paths as data, the policy creator can then assert their existence in the Rego policy because the paths defined as data are always available in the runtime but they are evaluated to "null" if they do not resolve to a file in the repository.

Output
The CLI provides an interface to implement different output formats based on the results given by rule evaluation. Currently, there are implementations for JSON and text outputs. As a separate feature, there is also a GitHub reporter that enables reporting evaluation results in a easy to understand format to GitHub PRs based on a Go template.

4.5 Surrounding tooling

This subsection will describe all the surrounding tooling needed to meet the objectives set in section 3.3.4. The author will explain how the new CLI runs in the CI/CD flow compared to the old one. Furthermore, they will describe the bulk execution feature used to validate the new solution and detail the new metrics solution implemented.

4.5.1 CI/CD

As mentioned in section 3.2, the CI/CD flow used Dora in three different locations. PR time, build time and deploy time, the last of which was the most problematic. The fact that a new change is deployed incrementally to multiple regions at an interval of many hours in most cases caused the context of each validation run to be vastly different. The change in context often caused failures that the developer could not prevent as the optimal value for some configuration variable had changed when their change reached a new region. The deployment time validation was removed to mitigate such issues. The CI/CD tooling was adjusted to validate that a deployment would only happen if the changes were from the main branch of a GitHub repository. Only allowing changes from the main branch combined with the restrictions set for each repository meant that the code deployed must have passed validation at some earlier point in time.

4.5.2 Bulk Execution

Another big concern of the maintainers was the lack of validation when introducing a new rule. Before the migration to the new solution, there was no way to execute configuration validation across all organisation's GitHub repositories making it hard to understand the impact a new rule would have. The bulk feature gives the DevOps engineer a preview of how many repositories would be blocked by a new critical level rule for example.

A secondary concern was the rule migration process. Introduction of a new tool was the final push needed to implement such a solution as there was a dire need to validate if all the migrated rules behave as expected across all of the repositories. The comparison opportunity helped quickly identify differences in the two solutions and detect faults in both. The author will describe the findings of these comparisons in section 5.2 of the thesis.

4.5.3 Metrics Gatherer

The only data available from Dora was the messages sent to a public Slack channel. Dora sent a message for any execution that detected some rule violation. These messages were not in a friendly format for more profound analysis or real-time metrics as they were designed for human consumption. To gain some initial insight into Dora, the author analysed the data available in this channel, but this was quite limited, mainly

because transforming the data into a usable format required significant effort. For the new implementation, the author implemented a data gathering solution that would allow for real-time metrics, dashboarding and historical analysis. The following section will describe the architecture of the metrics gatherer and explain the reasons for each design decision.

The solution for gathering metrics consists of three components. There is a separate reporter in the CLI, a message receiver service, and a message handler service. After validating a repository, the CLI will transform the validation results into a suitable format for the metrics service. The CLI sends this data with an HTTP request to a central message receiver that forwards the message into a RabbitMQ queue. The metrics handler service will then pick up the message from the queue and save it into a MySQL database. The high level overview of the metrics gathering solution can be seen in Figure 4.

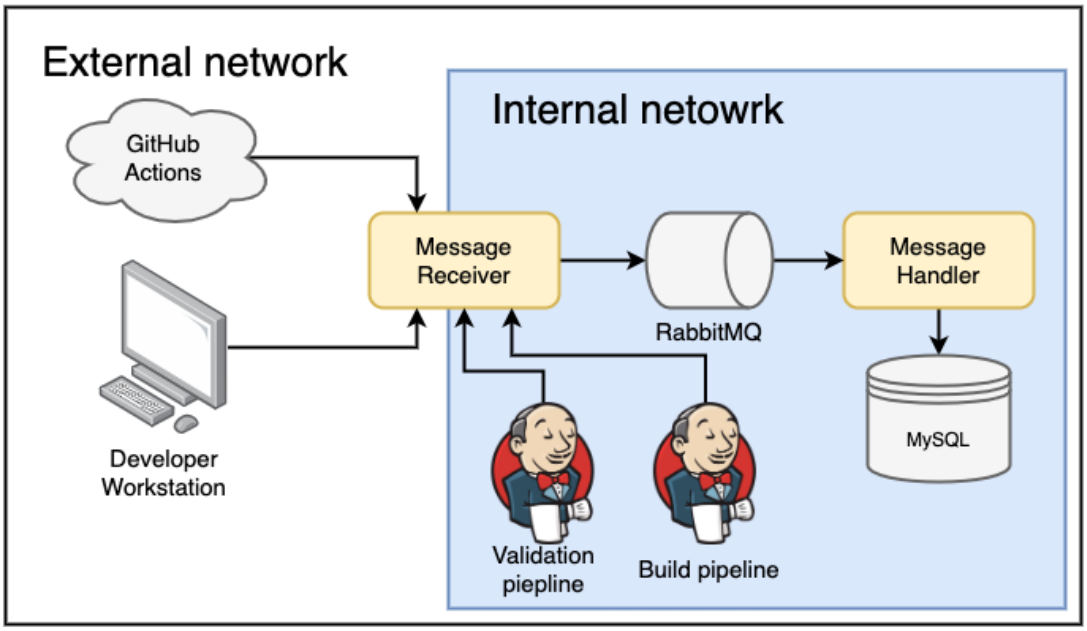

Figure 4. Metrics gatherer overview

Reporting from the CLI

Because the validation occurs in multiple locations, like GitHub actions, Jenkins pipelines or external CI/CD tooling, it is reasonable for the CLI to be a reporter of metrics. Otherwise, each tool that uses the CLI will need to implement separate reporting logic. The user can toggle this reporting with a separate command-line flag that tells the CLI to send a metrics event to the central message receiver.

Message Reciever
A central message processor receives all the metrics events and forwards them into a RabbitMQ queue. The author decided that this middleman is needed for two main reasons. First, the metrics handler is located in an internal network for security reasons. However, the CLI might run in environments that do not have access to internal API endpoints in some cases. An example of such an environment would be GitHub actions. The company already has a central public-facing message receiver that forwards messages to internal services to solve this issue. Furthermore, the already existing message receiver can save messages in a RabbitMQ queue, decreasing the overall complexity of the metrics handler by eliminating the need to implement API endpoints. The metrics handler will only need to process messages from a queue and forward them as necessary. Using a queue will also increase the system's reliability and mitigate potential performance issues as reporting volume done by the CLI will be quite significant.

Message Handler
The message handler is a Node.js service that collects events from a RabbitMQ queue and stores them in a MySQL database. The volume of the data is considerably large. One execution against each repository of the company produces around 34000 new rows. This number increases significantly because the CLI validates multiple branches for each repository, and the validation happens multiple times for each branch. The data gathered should be used in creating different dashboards to provide an overview of the current state of each rule and repository. As dashboards frequently query the database, the queries would become extremely slow if all of the historical data were in a single database table.

To mitigate this problem, the author decided that the best approach would be to keep the current state required for fast real-time metrics in the MySQL database with a periodic clean-up process that would delete entries as they become too old.

The message produced by the CLI is a bulk summary of the validation containing the following metadata:

- **repository** - Name of the git repository that was validated.
- **branch** - Name of the git branch that was validated.
- **execution_location** The context where the validation was done. One of PR, BULK or BUILD.
- **run_id** - Unique ID for the validation.
- **commit** - Git commit hash that points to the repository state that was validated..
- **execution_time** - Timestamp of when the validation was finished by the CLI.

And for rules, the message will contain an array of items where each item will describe the results of evaluating the rule. Each item has the following fields:

- **rule_name** - Name of the rule. Same name that is used to identify the rule in specifications and repository specific override configurations.
- **level** - Severity level of the rule, either WARNING or CRITICAL.
- **status** - Status of the rule for the repository, one of PASSING, FAILING or SKIPPED.
- **report** - The message provided by the CLI that describes the rule with expected and actual values.

The metrics service will create a row for each rule item and add common meta information of each run to create a new data row that will be saved into the MySQL database. The combined row will have all of the top level metadata for the execution combined with all the data from each rule evaluation. In addition one more value will be calculated for each row. This value is the last failure time of the rule in the same context.

For insertions, the service will first validate if there already exists a row for the same context and override it in the case it does. Duplicate rows are determined by looking for a row with the same combination of repository name, git branch execution location and rule name. If the service finds a row with the same combination of values, it overwrites it. If the previous validation result was a failure and this one is not, then the previous execution time is saved as the last failure time for the row. Overwriting previous rows allows us to only keep the most recent state for each validation context and reduce the amount of data stored. There is also a periodic worker in the metrics service that cleans the database of rows that originate from a non-main branch and are older than a week to solve the issue of where old branches will start polluting the data after they get merged into the main branch.

5 Results

In this section, the author will describe how the new solution was validated. Firstly, section 5.1 compares the two tools based on various code quality metrics. Section 5.2 looks into how different rules improved after the initial validation period. Next, section 5.3 takes a look at Slack messages of the developers, after which section 5.4 will go into some subjective discussion about the new solution. Finally, in section 5.5 the author will propose eight best practices for using automatic configuration validation.

5.1 Code Quality Comparison

First, we will compare the code quality of the projects as this was one of the most significant pain points for the team maintaining the old tool. For this comparison, the author used a static analysis tool called Sonarqube. As Sonarqube does not analyse Rego code, not all metrics were directly comparable. For example, previously, the rules were described in Javascript and mixed with CLI implementation, so Sonarqube also scanned them, but for the new tool, OPA CLI measures the Rego rules, and Sonarqube analyses Go code.

First, let us look at the quality metrics that changed after the move from one project to another and that are directly comparable based on the two projects Sonarqube reports: maintainability, duplication and complexity.

For maintainability, Sonarqube offers three metrics first of which is the number of code smells together with a severity rating. The total number of code smells Sonarqube reports for the old project is 152, while for the new, it is four. Figure 5 displays the exact severity categorisation of the smells for both projects. As we can see, the number of smells in each severity category has reduced significantly, making the new project's overall maintainability much better.

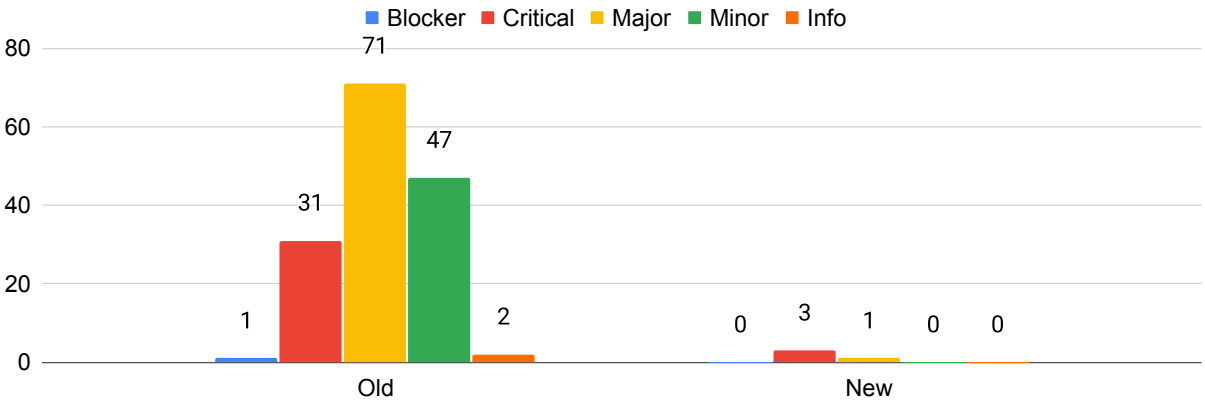

Figure 5. Comparison of code smells based on severity between the new and old tool

The next maintainability metric to look at is the minutes required to solve technical debt in the project. Each issue type in Sonarqube has a function to calculate the time to solve it. As we can assume from the significant reduction of code smells, this metric has also decreased in the new tool. In Figure 6, we can see that the exact difference is a 5724-minute reduction, which translates into about four full days or about 12 working days less for the new project.

Figure 6. Comparison of the minutes required to solve technical debt between the new and old tool

As a final maintainability measure, Sonarqube offers the ratio between the cost to develop the software and the cost to fix it - R. From Sonarqubes documentation [36] the formula is:

$$R = C_r/(C_d * LOC)$$

Where C_r is the remediation cost - total time to solve technical debt in days; C_d is the development cost - value of the cost to develop a line of code, constant 0.06 days as defined by Sonarqube and LOC - number of lines of code.

The improvement in this metric is not as significant as in the previous two as the initial value was already quite low. In Figure 7 we can see that it decreased from 3.7% to 0.1%.

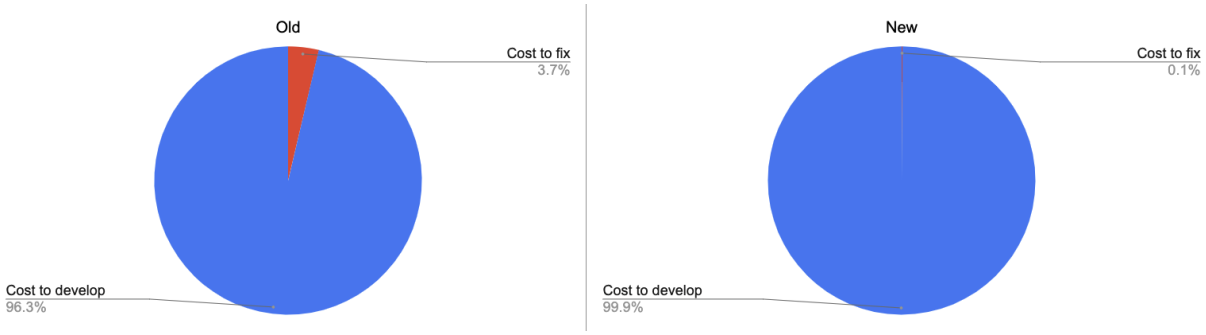

Figure 7. Comparison of the technical debt ration between the new and old tool

For measuring test coverage, we will take a slightly different approach. First, because the old tool had rule code mixed with implementation and thus we can not separate the two. Furthermore, Sonarqube does not analyse Rego code, but OPA CLI offers us test coverage information. We can combine Sonarqube and OPA CLI measurements for the new tool and compare them to the test coverage measurements from Sonarqube for the old tool to get a more accurate comparison.

The old tool had 2262 lines of code to cover with 952 lines of uncovered code. Meaning the total code coverage of the tool was 52.9%. For the new CLI, the total number of lines to cover measured by Sonarqube was 1053. Uncovered of those lines was 109, resulting in 89.6% coverage. As for the Rego code, the total number of lines to cover was 1298, and the coverage measured by OPA was 97.6%. By combining the two metrics for the new CLI, we can see that the total number or lines to cover was 2351, and the coverage percentage when combining the Go and Rego code was 95,3%. Form Figure 8 we can see that while the total number of lines to cover is slightly higher for the new solution the overall test coverage increase makes up for it.

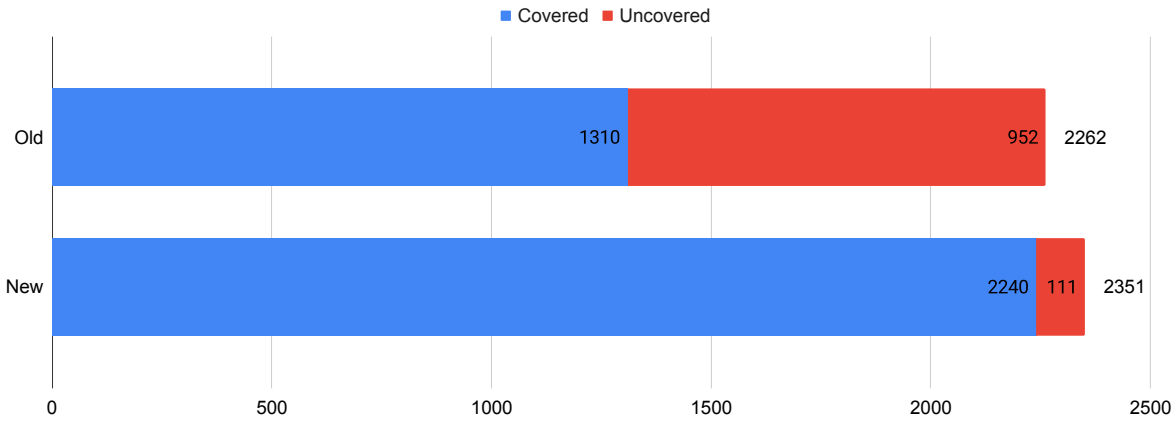

Figure 8. Comparison of test coverage in the new and old tool

5.2 Rules

Next, we will look at how the rule statuses changed after the release of the new solution. To make this comparison, we will observe the number of failures for each rule just before the new solution was released and compare it to the number of failures one month after the release. The rule-by-rule comparison is in Figure 9. By looking at each rule, we can see that 18 rules had a positive change after one month, meaning fewer failures than before. Nine rules remained the same, and six had a negative change.

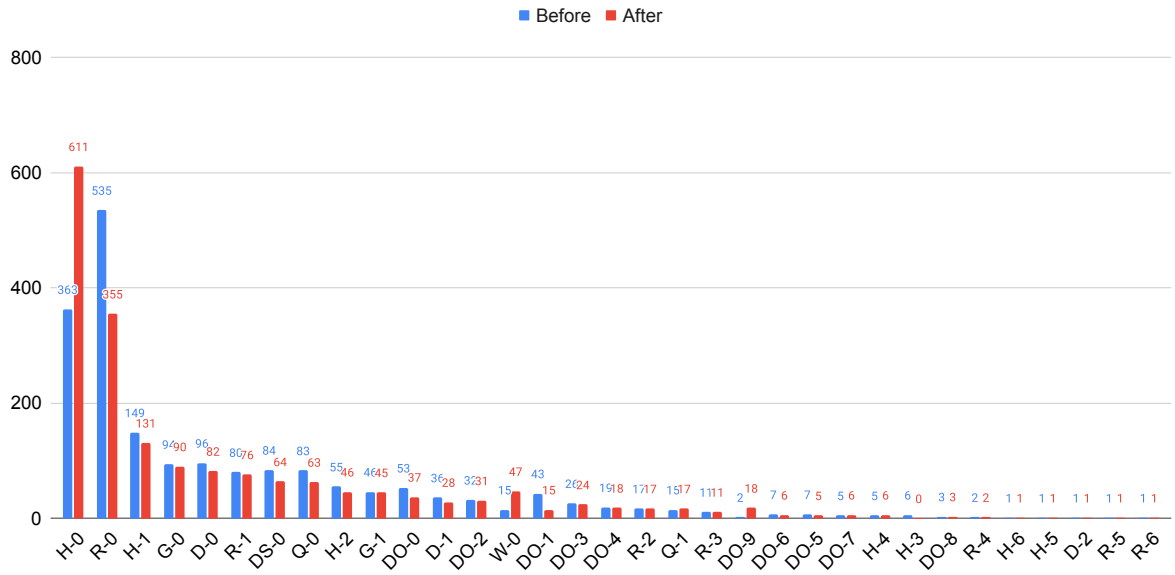

Figure 9. Comparison of rule failures before and after the 1 month evaluation period

The author will discuss each category separately to understand better what these numbers mean. We will look at the simplest case, rules with a positive fix ratio. These rules can be considered adequate as they make the developer fix some configuration mistakes. The author acknowledges that most of these improvements are not that significant, but it is not currently possible to observe the trend at a more meaningful scale due to time constraints. Another contributing factor to the improvements not being that significant in most rule statuses is that a majority of the rules are only warnings and will not block PR merges or builds, meaning that developers can ignore them.

When looking at Figure 9, a question might come up about what is different for rule R-0 as the change is quite significant compared to others. The rule labelled R-0 shows how the tool can enforce some practice when necessary. The rule in question changed considerably during the rule review process causing 535 repositories to fail the validation at first. We can observe that if a significant change is introduced, it will get fixed for many repositories. Another contributing factor for this specific case might be that this rule was introduced as critical, meaning that it blocked merging PRs and creating builds.

Next, we will discuss the rules that had no change. The author believes that these are, in most cases, caused by two key factors that are also the same as why most of the positive change is relatively small.

The first factor, as previously described. A lot of the rules are just warnings. The developers can ignore them if they want to, and since these rules have been around for a while just enforced by a different tool, the cases where the warnings get ignored will not be fixed now just because the warning message is in a different format.

The second factor is that these metrics come from a bulk execution of the tool against all repositories. However, developers get feedback only on PR creation or pushes to existing ones. Combined with the fact that some repositories rarely get any updates, some rules will stay failing for a while. The same reasoning can also apply to the previously described case where the positive trend was minor for some rules.

Lastly, the most interesting case the rules that had more failures after a month. Three of these cases with a slight uptick: Q-1, DO-7 and H-4, can be attributed to the developers creating new repositories frequently and causing new warnings. When creating a new service, fixing these warnings is usually not prioritised.

As for the remaining three, we should look at each separately as they are all caused by different factors. Let us start from the rule H-0. The rule is about bumping the version of the central Helm chart used by each service in the organisation to the latest released version. A significant increase in failures happened because the template version was updated on Friday before the author did the analysis. No one had started upgrading to the new version before the data extraction. A more accurate comparison would be to see what was the state one day prior to the new template version. The number of failures was 492 one day earlier. It is still an increase from the initial 363 but significantly less than 611.

In the author's opinion, this rule shows a troubling trend in its status changes by constantly moving up and down in failure count based on the version releases of the Helm chart. Such a trend is troubling because it indicates that a rule failure is out of the developer's control as it gets reintroduced by something external. As of now, the author is unfortunately unaware of a better approach for enforcing the version upgrades so the rule remains in action.

The next rule with a significant uptick was W-0. For this case, the reason was simple. A fundamental mistake in the old tool got reintroduced while adding the rule in the new tool. This mistake was realised during the first month after the release and fixed. The fixed version managed to find many previously undetected faulty repositories.

And lastly, the rule DO-9. A DevOps engineer changed something that required developers to break the previous convention enforced by the rule without changing the rule definition used for repository validation.

In conclusion, even with the rule H-0 behaving like it does and causing a significant increase in failures. The overall failures decreased from 1894 to 1859, and if we exclude this rule, the change is from 1531 to 1248. As the time frame for validation was relatively small, the author considers this a successful result with the takeaway that PR time validation is not a way to solve all misconfigurations at a rapid pace but rather a way to inform developers and prevent them from introducing new mistakes.

5.3 Developer Feedback

For the final point of comparison, we will now look at the same Slack channel history that the author used for the initial analysis in section 3.3.1. A direct and fair comparison is not possible as the new tool has been in use for only a month, meaning the amount of data available is limited. Nevertheless, we can still look at the general trends in the month following the release to get some idea of how the tool is performing.

As illustrated in Figure 10. The total number of messages about the tool posted to the Slack channel was 28. Considering that these messages are from a period of one month, if we count only the working days within the two periods, the frequency of messages increased from about one message per day with the old solution to about 1.3 a day with the new one. An important thing to consider here is that if we look at Figure 11 we can see that 13 of the 28 messages originated from the two days following the release. It is expected that the frequency of questions increases during the initial transition period. After the initial release we can see that the maximum number of messages in a day is two and a lot of days have zero messages about the tool especially towards the end of the evaluation period.

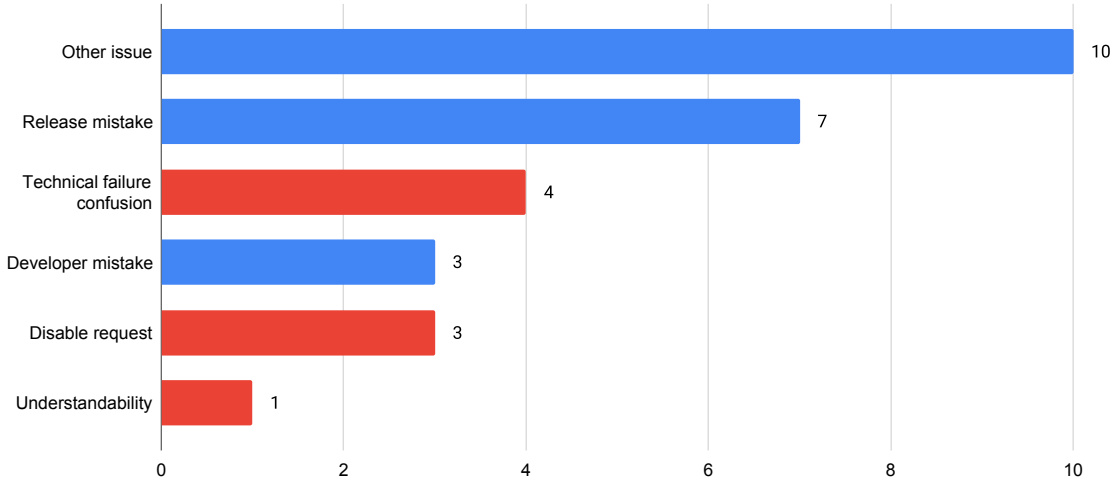

Figure 10. Categories of Slack messages about the new solution after 1 month

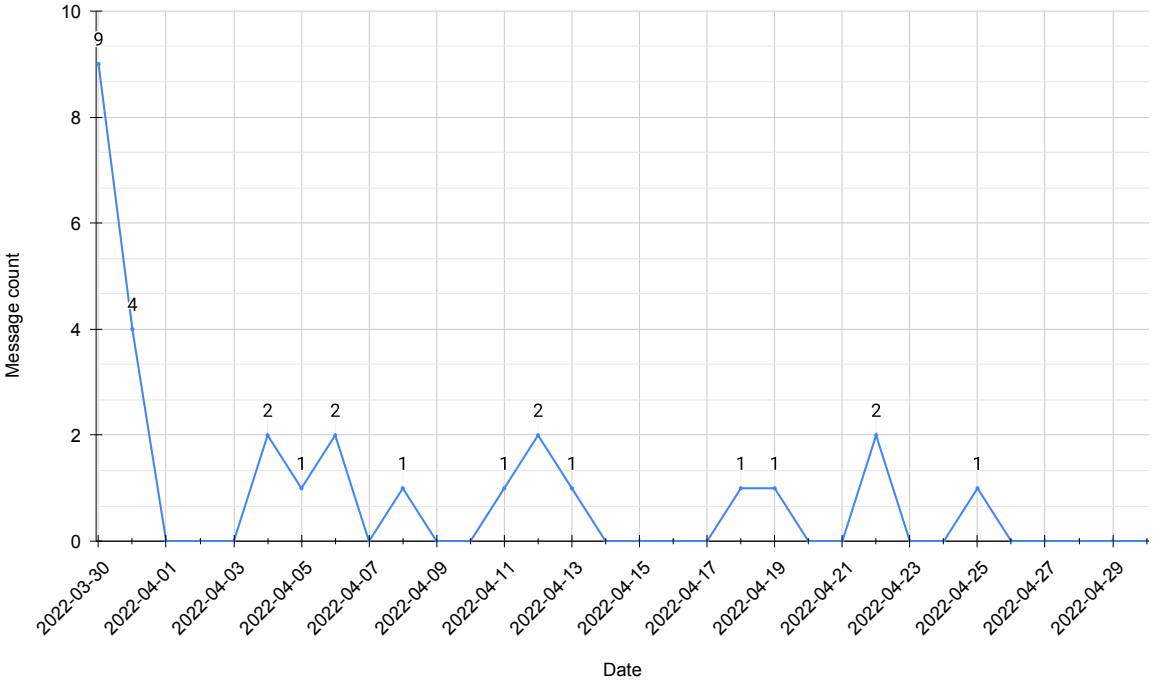

Figure 11. Number of Slack messages about the new solution over one month

Now looking back at Figure 10, the categories considered non-issues are again blue, and issues are red. We have eight relevant complaints about the new solution and 20 messages about something else that mentioned the new tool in the message. Like we did during the initial analysis in section 3.3.1, let us now look at the meaning behind each of the categories more thoroughly.

Other issue (10)
This category holds all the messages where the developer reported a real problem, but it was not with the new tool. The message posted in Slack just contained the tool's name while talking about some other issue.

Release mistake (7)
All seven of the messages in this category were complaining about a single thing. This could be considered a mistake in the release process. We did not consider that some people had set the status check posted by the old tool as a requirement in their PR settings in GitHub. When we turned off the old tool, they started getting a status check from the new tool but could not merge the PR branch because GitHub settings required the check

from the old tool. These complaints are not a problem with the tool itself but more of an oversight of the DevOps engineers.

Technical failure confusion (4)
Messages in this category are about cases where the new tool reported a failure for some rule but not for failing the validation rather for some other technical reason—for example, not being able to reach some remote resource over HTTP. Sometimes the developers did not understand these messages, which caused confusion and complaints.

Developer mistake (3)
These messages were complaints where the failure was valid, but the developer did not find the mistake they made. As the number of complaints in this category was low and the messages were recent, it was possible to validate that all three of these complaints were not due to the lack of documentation about the rule. Instead, the developer failed to find the mistake they made. Thus, the author considered them as not issues with the new solution.

Disable requests (3)
Two messages where a developer requests the tool or some rules to be disabled are under this category. The author considered this the tool's fault during the initial analysis, and they will do the same now.

Understandability (1)
And finally, a single message where the developer did not understand what the tool was trying to tell them. Such a case is a fault of the new solution, more precisely, a missing clarification in the documentation of a rule.

In conclusion, the total number of messages that are actual complaints about the new solution was 8 out of 28. If we compare this to the previous 53 out of 65, we can see that the previous solution got about 0.8 complaints a day while the new one, even with the increased attention caused by the release, got 0.3 complaints a day. In the author's opinion, this is a considerable improvement.

5.4 Discussion

Now we will take a more subjective look at the new solution. The author will describe the new rule development experience and what people think of it so far. Furthermore, they will discuss the lessons learned from the release process of the new tool.

5.4.1 Rule Development

As a first point that was also brought out by Huang et al. [9] let us look at the separation of concerns aspect, to be more specific, how well are rules separated from the implementation details of the CLI. By looking at Figure 12, we can see a simplified example of the project structure. The CLI implementation lies in the internal directory, with the Go code separated into different modules based on functionality. As for the rules, we can see that they are in a separate package called rules. Rules are separated by namespace, and each rule has its sub-directory that contains all of the files described in section 4.4. The interaction between the two packages is one-way. The internals load and invoke the rules, but the rules themselves have no access to the Go code. This separation restricts the developer from mixing any implementation details in the rules. Since rules definitions are in Rego, a declarative query language, it is difficult for the developer to mix any complicated logic into the rule definition.

At first, it might seem like this is a downside as some functionality might not be implementable. From the author's experience, this is not the case so far, after implementing about 40 rules. Every rule the company previously had is possible to implement in the new tool as well. Indeed some adjustments needed to be made to a few rules. However, it is arguable that these changes simplified and improved the clarity of the rules while maintaining their original intent.

Adding new parsers or output formats is as simple as creating new implementations for the standard parser or outputter interfaces located in the internal package. Keeping parsing and output out of the rule definitions helps declutter the validation logic and lets the developer of the rule focus on all the parts separately.

As for the rule-writing experience, developing rules in Rego is overall a positive experience from the limited feedback gathered so far. For most, there is an initial hurdle in getting acquainted with Rego as it is a new language in a paradigm that is unfamiliar to many. Nevertheless, the developers that have used it so far have said that the experience is a definite improvement over the old way. A somewhat biased opinion of the author about the experience is that it is sometimes frustrating because of some Rego concepts. Other than that, they would describe it as logical and go so far as to say that it is very rewarding and even satisfying to use at times.

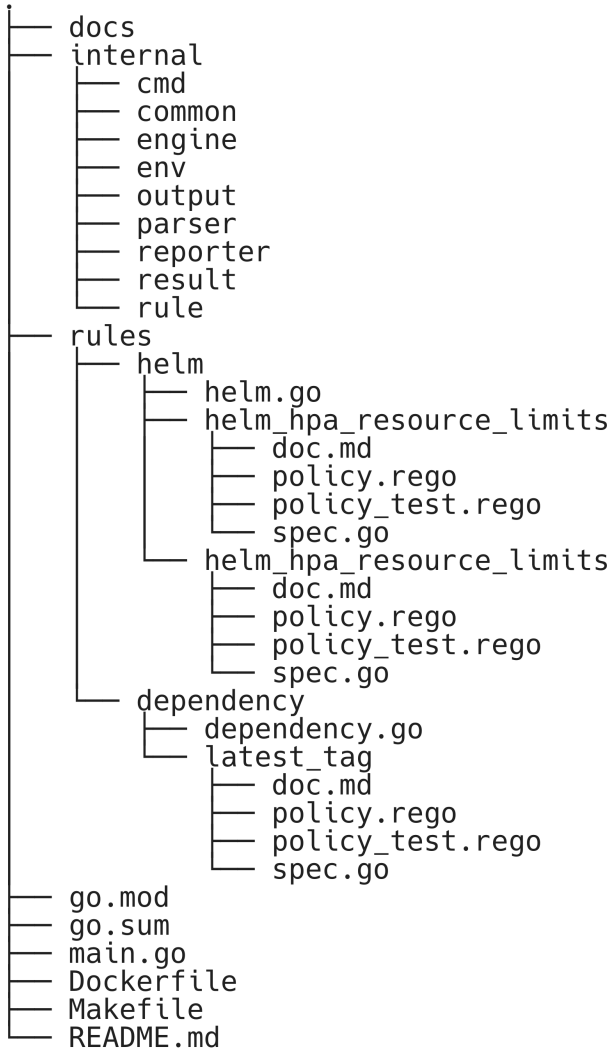

Figure 12. Simplified project structure

Another bonus not directly related to the new CLI implementation is that the combination of bulk execution and rule migration discovered some faulty rules. Bugs in the code of the old CLI lead to some rules that were never actually triggering, meaning some misconfigurations have gone unnoticed for years. These faults were all related to bugs in the implementation details like incorrect handling of HTTP requests or failing to parse a value out of a configuration file.

5.4.2 Lessons Learned

Additional key takeaways from the case study are the lessons learned from the release of the new CLI.

The first lesson was more a confirmation of a previously known problem. **Developers do not like to read detailed documentation or problem descriptions.** Although the author rewrote all rule descriptions and provided quick fix examples for every rule applicable, some developers still had trouble understanding what to do and why some rules exist. Some of these misunderstandings are the author's fault, as developers are sometimes interested in the reasoning behind why we are enforcing a particular rule. The author omitted these details in a few cases because they seemed apparent or unimportant enough not to include. In other cases, the developers did not read any additional details in the failure messages. They only saw that something was wrong and did not bother to investigate further, leading to questions that already had answers in the linked documentation.

Another somewhat related issue is that **making a rule fail for a technical reason is not always good.** For example, suppose some validation relies on an HTTP call, and this call fails for some reason. Reporting this to the developer might leave them confused, even if the output shows them that the reason was a failed API call.

An isolated instance of the two previously described problems was where a developer got a failure message about using a docker image that was not allowed. This failure happened because the docker hub API failed to respond, not because of a bad container image definition. Without reading the documentation and understanding the purpose of the rule, the developer proceeded to replace all of the base images in the services they manage. The migration process had a cascading effect. They had multiple compatibility issues between their configuration and the new replacement docker images that needed help from the DevOps team to solve.

The last big takeaway was that **the developers will sometimes try every possible way to get the validation passed before even attempting to solve the reported issue**, even if a rule is valid and enforcing something like a critical security practice. The problem exposed itself in multiple ways. Some developers will ask for ways to disable a rule even if fixing the issue would be more straightforward. Others will argue over the validity of specific rules. The best examples were in cases where a rule was previously never validated because of a bug. After the bug got fixed, suddenly developers were getting warnings about problems they had had in their projects for years. Instead of reading the provided reasoning and implementing the suggested fix, some resorted to using arguments like, "It was okay before. Why is it not okay now."

5.5 Best Practices

The author will now answer the **RQ: What guidelines should DevOps teams follow when using large-scale automatic configuration validation?** To answer the RQ, they will propose best practices for configuration validation based on the information gathered from related works and experience gained from the case study in order to help others having similar issues described in section 3 while using configuration validation at a large scale.

Configuration validation should be complementary to deployment testing. This first practice primarily originates from the work of Huang et al. [9] In their article, they describe that while deployment testing is an effective and proactive solution for preventing misconfigurations from reaching production, it might not be possible to do in all scenarios. The primary constraint is that deployment testing does not scale well at cloud scale and will get very costly. The problem is especially relevant in more recent times, where the DevOps movement has helped companies increase their number of daily deployments into hundreds. Complementing deployment testing with a preliminary configuration validation step helps catch some of the mistakes early in the CI/CD process, thus reducing the resources required for deployment testing.

Configuration validation should not be the only gatekeeper. Throughout the case study, the author realised that some of the problems in the existing approach stemmed from the fact that the configuration validation tooling seemed to be the first and final place for any validation that the CI/CD process needed to apply. This mindset introduced some validations that had no place in configuration validation. They were either too complex to be solved at this level or were too context-dependent, complicating the rule descriptions unnecessarily. Huang et al. also mention that similar to other validation approaches, configuration validation cannot reject all invalid configurations because a configuration that passes all specifications is not necessarily correct [9].

Validate only actionable items. If the end-user for the configuration validation tooling is a developer, then the things validated should be things that the developer has control over. There is little point in providing developers feedback about things that they can not change or that are unrelated to their changes. One example of this problem is the rule H-0 described in section 5.2. Feedback like this might cause unnecessary friction and noise. It might make the tool lose credibility for the developers and ignore the validations completely when they constantly have to deal with unfixable issues.

Avoid context dependant validation. Configuration validation should not be the place to do context-specific validation. Especially in the case where it is also used as a blocking

mechanism for CI/CD. An example of validations to avoid are rules that only apply for deployments and are dependant on the deployment region. Based on the analysis done in this study, such rules often have different results based on the deployment's region. When the context is different for each deployment region, the validation might succeed for some regions but not all. Failures like this can cause frustration in the developers as deployments might take several hours to propagate to all regions. When an issue occurs at some later region, they have to switch their attention away from what they were doing and solve the problem.

Do not mix validation code with implementation details. Mixing validation logic with things like parsing logic or output formation causes the rules to become hard to understand and maintain. The author believes it is good to have different languages for validation and the surrounding implementation. Separating concerns allows the rule writer to focus on validation logic without thinking about how to fetch some value from a configuration file or what the output will look like.

Each rule must be individually testable. One of the problems solved in the case study was that the validation logic was hard to test. More precisely, it was difficult to validate rules individually or even test multiple scenarios for a single rule. The rules must be testable, and the test cases must be straightforward to implement. Furthermore, it is also essential to have success and failure scenarios for each test to keep up the maintainability and provide examples for future readers of the rules. The author proposes that as a rule of thumb, each rule should have a success case and a failure case for every failure scenario. An excellent way to enforce this is to measure test coverage and set a minimum requirement for each new rule.

Validation code should have an emphasis on readability. Generally, configuration validation requirements change more often than regular application code as best practices evolve and requirements enforced by a company change. The more frequent change also means that the code must be read and changed more often. Making rule code easy to understand might also give developers additional insight about the rule if they decide to check the validation code in addition to the documentation. Thus, it needs to be easily understandable over anything else. The writer should refrain from trying to write clever code or complex one-liners. Sometimes it might even be better to duplicate logic in order to avoid hard to understand abstraction. Some tools like OPA, for example, even have built-in optimisation so the author can solely focus on the correctness and readability of their code [37]. Another case favouring Rego and OPA is to consider using a high-level declarative language to define rules over an imperative one.

Technical failures should be separated from rule failures. The scenario described in section 5.4.2 is an excellent example of why reporting a technical failure, like a failed HTTP request, as a rule failure might be problematic. An further confirmation is that more than half of the valid complaints made about the new solution were caused by technical failures. The author proposes that when such a failure is unavoidable, the writer of the rule should consider making the failure reason very explicit or even not a failing case for the end user. When the issue is not critical, letting it pass might cause less harm in the long run than showing a rule violation to the developer. Even if the reported message indicates a technical failure, some might not read it thoroughly and make assumptions based on only the rule name. When making such consideration it is also important to keep in mind that even if technical problems are not reported to the end user, they are still important for the maintainers of the rule engine to detect misbehaving parts of the system.

6 Conclusion

The thesis analysed how to use large-scale automatic configuration validation effectively. The analysis was done based on a case where an existing configuration validation solution performed poorly at a specific company. The author gathered input from the DevOps team that maintains the tool and from the history of developer's complaints about the tool. In addition, they analysed related literature to understand if peers have run into similar problems and what could be the possible solutions. They then used the collected information to map out issues with the previous tooling and formulate requirements for a new solution.

After forming the requirements, the author investigated possible existing solutions from the open-source community and evaluated their fit for the case. They decided that Open Policy Agent is a suitable technology to use as a basis for a new configuration validation solution. As a next step, they designed and lead the implementation of a CLI tool around Open Policy Agent based on the previously created requirements and evaluated the solution. Finally, the author proposed eight best practices based on the experience gathered from the study in order to try and help others in adopting automatic configuration validation at scale. In short the proposed practices were the following.

- **Configuration validation should be complementary to deployment testing.** Complementing deployment testing with a preliminary configuration validation step helps catch some of the mistakes early in the CI/CD process thus reducing the resources required for deployment testing.
- **Configuration validation should not be the only gatekeeper.** Configuration validation cannot reject all invalid configurations because a configuration that passes all specifications is not necessarily correct
- **Validate only actionable items.** Feedback about unactionable items might cause frustration and noise which makes the end user lose credibility in the feedback.
- **Avoid context dependant validation.** Providing different results for the same state of configurations because of some external factor might cause confusion and frustration in the end user.
- **Do not mix validation code with implementation details.** Mixing validation logic with things like parsing logic or output formation causes the rules to become hard to understand and maintain.
- **Each rule must be individually testable.** The author proposes that as a rule of thumb, each rule should have a success case and a failure case for every failure scenario. Furthermore, making rules testable greatly improves the maintainability of the tooling as well as helps future rule readers understand the purpose of the rules.

- **Validation code should have an emphasis on readability.** Generally, configuration validation requirements change more often than regular application code as best practices evolve and requirements enforced by a company change. The more frequent change also means that the code must be read and changed more often. Thus, it needs to be easily understandable over anything else.
- **Technical failures should be separated from rule failures.** Reporting technical failures might cause more confusion in the end user then it will do good as the failure is not related to their changes.

The results of the limited evaluation period showed that the solution is effective at detecting misconfigurations and helping developers solve them. By excluding an outlier, the number of issues reported by the new solution decreased by 18.4%. The analysis of Slack logs also showed that while there was an initial spike in messages about the tool, this slowed down quickly after the release; furthermore, the number of complaint messages reduced from 0.8 to 0.3 a day. The software quality metrics of the tooling also improved significantly. Most notably, the number of code smells, according to Sonarqube, reduced from 152 to four, and the code coverage increased from 52.9% to 94.5%. From the limited feedback so far, we can also say that the experience of writing new configuration validation rules in Rego using the designed framework is more pleasant than it was previously using Javascript. The main concern to the validity of the results is that the evaluation period was relatively short, and some problems did not have enough time to emerge.

Even though the solution is performing well, there is always room for improvement. So far, the author has two directions for improvement. The first would be to take a deeper look into what makes a good failure message, as there are still some problems with the understandability of the issues reported by the tool. And the second would be to automate the functional testing of the CLI. The rule developer can currently use the created bulk execution feature, but it is relatively manual and time-consuming.

References

[1] Baset S, Suneja S, Bila N, Tuncer O, Isci C. Usable declarative configuration specification and validation for applications, systems, and cloud. Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference on Industrial Track - Middleware '17. 2017.

[2] Sun X, Cheng R, Chen J, Ang E, Legunsen O, Xu T. Testing configuration changes in context to prevent production failures. In: Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020. Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020. USENIX Association; 2020. p. 735-51.

[3] Apache kafka [Internet]. The Apache Software Foundation; 2022 [cited 2022 May 11]. Available from: `https://kafka.apache.org/`.

[4] Redis [Internet]. Redis; 2022 [cited 2022 May 11]. Available from: `https://redis.io/`.

[5] Tuncer O, Bila N, Duri S, Isci C, Coskun AK. ConfEx: Towards Automating Software Configuration Analytics in the Cloud. In: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W); 2018. p. 30-3.

[6] Abad ZS, Karras O, Schneider K, Barker K, Bauer M. Task interruption in software development projects. Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018. 2018.

[7] Production-grade container orchestration [Internet]. Kubernetes; 2022 [cited 2022 May 8]. Available from: `https://kubernetes.io/`.

[8] Isolate containers with a user namespace [Internet]; 2022. Available from: `https://docs.docker.com/engine/security/userns-remap/`.

[9] Huang P, Bolosky WJ, Singh A, Zhou Y. Confvalley. Proceedings of the Tenth European Conference on Computer Systems. 2015.

[10] Morris K. In: Chapter 1: What is Infrastructure as Code? O'Reilly Media, Inc.; 2016. p. 4.

[11] Containerization [Internet]. IBM; 2022 [cited 2022 May 11]. Available from: `https://www.ibm.com/cloud/learn/containerization`.

[12] Dragoni N, Giallorenzo S, Lafuente AL, Mazzara M, Montesi F, Mustafin R, et al. Microservices: Yesterday, Today, and Tomorrow. Present and Ulterior Software Engineering. 2017:195–216.

[13] Harness. What is helm? introduction to helm: Charts, deployments, & more [Internet]; 2022 [cited 2022 May 11]. Available from: https://harness.io/blog/continuous-delivery/what-is-helm/.

[14] Open Policy Agent: Introduction [Internet]. Open Policy Agent; 2022 [cited 2022 May 11]. Available from: https://www.openpolicyagent.org/docs/latest/.

[15] Xu T, Legunsen O. Configuration Testing: Testing Configuration Values as Code and with Code. arXiv: Software Engineering. 2019.

[16] Guerriero M, Garriga M, Tamburri DA, Palomba F. Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry; 2019. p. 580-9.

[17] Dai T, Karve A, Koper G, Zeng S. Automatically detecting risky scripts in infrastructure code; 2020. p. 358-71.

[18] Borovits N, Kumara I, Krishnan P, Palma SD, Di Nucci D, Palomba F, et al. DeepIaC: Deep learning-based linguistic anti-pattern detection in IaC; 2020. p. 7-12.

[19] Kumara I, Vasileiou Z, Meditskos G, Tamburri DA, Van Den Heuvel WJ, Karakostas A, et al. Towards Semantic Detection of Smells in Cloud Infrastructure Code. vol. Part F162565; 2020. p. 63-7.

[20] Schwarz J, Steffens A, Lichter H. Code smells in infrastructure as code; 2018. p. 220-8.

[21] Rahman A, Farhana E, Parnin C, Williams L. Gang of eight: A defect taxonomy for infrastructure as code scripts; 2020. p. 752-64.

[22] GitHub flow [Internet]. GitHub; 2022 [cited 2022 May 11]. Available from: https://docs.github.com/en/get-started/quickstart/github-flow.

[23] Paljasma T. Validating Docker Image and Container Security Using Best Practices and Company Policies; 2019. Available from: https://digikogu.taltech.ee/et/Item/b9367be6-0646-4c2f-b32b-56ee8a024f0d.

[24] Prometheus - Monitoring System & Time Series Database [Internet]. Prometheus; 2022 [cited 2022 May 13]. Available from: https://prometheus.io/.

[25] Checkov [Internet]. Bridgecrew; 2022 [cited 2022 May 11]. Available from: https://www.checkov.io.

[26] config-lint [Internet]. Stelligent; 2022 [cited 2022 May 11]. Available from: https://stelligent.github.io/config-lint.

[27] What is copper [Internet]. cloud66; 2022 [cited 2022 May 11]. Available from: https://help.cloud66.com/copper/index.html.

[28] GitHub: conftest/deny.rego at master · open-policy-agent/conftest [Internet]. Open Policy Agent; 2022 [cited 2022 May 11]. Available from: https://github.com/open-policy-agent/conftest/blob/master/examples/kubernetes/policy/deny.rego.

[29] Ceri S, Gottlob G, Tanca L. What you always wanted to know about datalog (and never dared to ask). IEEE Transactions on Knowledge and Data Engineering. 1989;1(1):146–166.

[30] Kubernetes Admission Control [Internet]. Open Policy Agent; 2022 [cited 2022 May 11]. Available from: https://www.openpolicyagent.org/docs/v0.12.2/kubernetes-admission-control/.

[31] open-policy-agent/conftest: Write tests against structured configuration data using the Open Policy Agent Rego query language [Internet]. Open Policy Agent; 2022 [cited 2022 May 11]. Available from: https://github.com/open-policy-agent/conftest.

[32] Trivy documentation [Internet]. Aqua Security; 2022 [cited 2022 May 11]. Available from: https://aquasecurity.github.io/trivy/v0.27.1/.

[33] Welcome to the Regula Docs [Internet]. Fugue; 2022 [cited 2022 May 11]. Available from: https://regula.dev/.

[34] Cobra.dev [Internet]. Cobra; 2022 [cited 2022 May 11]. Available from: https://cobra.dev/.

[35] Consul [Internet]. HashiCorp; 2022 [cited 2022 May 13]. Available from: https://www.consul.io/.

[36] Metric definitions [Internet]. SonarQube; 2022 [cited 2022 May 11]. Available from: https://docs.sonarqube.org/latest/user-guide/metric-definitions/.

[37] Hinrichs T. Rego design principle #3: Optimize performance automatically [Internet]. Open Policy Agent; 2020. Available from: https://blog.openpolicyagent.org/rego-design-principle-3-optimize-performance-automatically-2d29ad3ce96d.

I. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Kaarel Loide**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Analysis of Practices for Large Scale Configuration Validation - A Case Study, supervised by Pelle Jakovits and Jevgeni Demidov.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Kaarel Loide
17/05/2022