The blasphemy of YAML

John Landeholt (johnlan@kth.se)

5 April 2022

1 Introduction

An ever-growing problem that new software engineers encounters only first when they actually enter the industry is the *lack of standards*. No practice is the other alike. This is especially true for large scale software development that is inherently multi-stepped in nature. At a large scale, a system can be divided into smaller systems called micro services. These services can further in line be configured differently, as per a predetermined release strategy. This is where configuration files comes into the practice of DevOps in order to deliver software faster and more reliably. Yet, there isn't a common understanding of what requirements a configuration file, or rather what a configuration language should have. This have resulted in a scattered and divided marketplace for configuration languages, where YAML has taken the leading role because of its human readability. But is human readability the de facto only prerequisite for a configuration language?

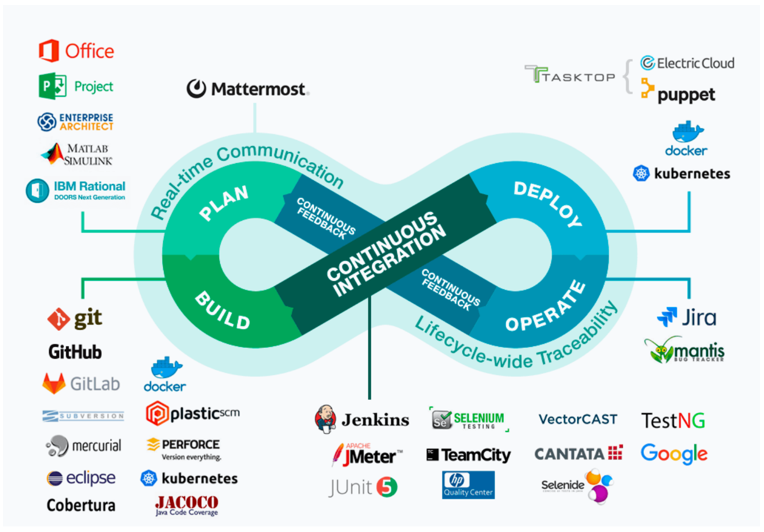

Figure 1: DevOps pipeline cycle, and its tools and platforms [1]

This paper will highlight how the configuration step is becoming a bottleneck for DevOps, and how a new specification could solve this. This matter foremost, as configuration files act as the hidden glue between each step of the DevOps pipeline. And as visualized by Figure 1, it should be apparent that it can become fatiguing as each platform or tool uses its own interface for configuring their software.

2 Infrastructure as Code versus Data

Stroustrup, Bjarne [2] suggested already in 2012 that Infrastructure software needs more stringent correctness, reliability, efficiency and maintainability requirements. His proposal suggests that a configuration language should have a static structure enforced by a type system as well as having first-class tools that supports the very basic necessities of automation.

For quite some time now, configuration languages has only been data. Especially serializable data. This has however its drawbacks. Mainly that it can only enforce one thing, and that is its data structure. Take for instance the prominent data serialization language YAML. Accordingly to the specification of YAML [3], it can only enforce that a block's entries are either a *collection* (map, sequence or sequence of maps, etc) or a *scalar* (string, integer, floats, timestamps, boolean or a tag). Each block is a key-value pair, which value in itself is a block, thus allowing for deeply nested relations. YAML can't enforce its data, only its structure. It requires external tooling such as *python scripts* to enforce strict object parsing by its "TAG" directive [4]. It is hence not compliant with the requirements that Stroustrup, Bjarne [2] suggests.

Further more, data serializing languages does not provide the same reusability that a regular programming language entails [5]. Meaning that YAML essentially breaks the DRY principle, but requiring re-writes and duplication of operations across environment and services. Infrastructure as Code promotes management of multiple subsystems as a single commonly available source of truth [5] by splitting up subsystem and its configure into separate re-usable blueprints.

3 The Configuration Complexity Clock

The Configuration Complexity Clock is a story first mentioned by Mike Hadlow [6]. Its story surrounds a developers project life cycle, where one at the beginning starts of with hard coding values directly into the project for its simplicity. But as the developer sobers up from its sleepy mind, it realises that it might be better of using *configurable values*. The developers

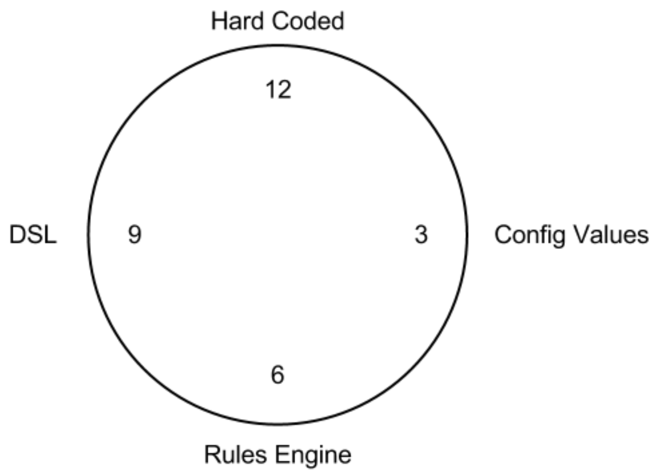

Figure 2: The Configuration Complexity Clock [6]

change of heart comes from understanding that it isn't feasible to rebuilding and redeploying it just to change a few hard coded numbers. Its obviously wrong. However, as time passes the developer receives new requirements for the project and realises that the *configurable values* aren't enough. The project needs rules and structure! Enter schematics such as XML, JSON and YAML. All being data serializing languages, i.e data. As the project grows, its requirements grows. Consequently so does its configuration. Even worse is that the developer is the only one that fully understands the configuration and there is now a considerable learning curve before a new contributor can successfully carry out a deployment. This is obviously wrong. The project consolidates and its decided that it needs a *rules engine* in order to combat the now uninterpretable configuration. But during development, the development team realises that some of the business rules can't be configured using the new *rules engine* and has a crisis meeting about what to do next. One of the developers mentions that a Domain Specific Language (DSL) surely allows the team to write arbitrary complex rules and solve all the projects problems. So the team agrees and stops work for several months to implement the DSL. This is obviously wrong. Finally, it works and there has been several month gone by without any changes being need in the core application. But the project now owns and maintains a language, with barely any tooling. Furthermore, the code is difficult to read and to debug. Where have we seen this before? The project has grown in mass, but they are back to hard coding values, except in a much worse language.

The moral of the story is that a project needs to maintain its sanity. And by implementing custom tooling, or as in the story a custom language will inherently with its own complexity of maintenance. A software developer is far of better with going with something off the shelf rather than developing it in-house. With the slight drawback that it might not be exactly compatible with the their business requirements.

4 Open source has changed the market

Remind figure 1 and cross check it to figure 3. Open source has without a doubt made a vast contribution to software as a whole. Something that Xie, Chen, and Neamtiu [7] describes in depth through their study of open source projects. But as systems are becoming globally accessible and ever more demanding in up-time and responsiveness, DevOps has recited to automating whole infrastructures, where a arbitrary set of tools and software from figure 1 and figure 3 are being used in unison.

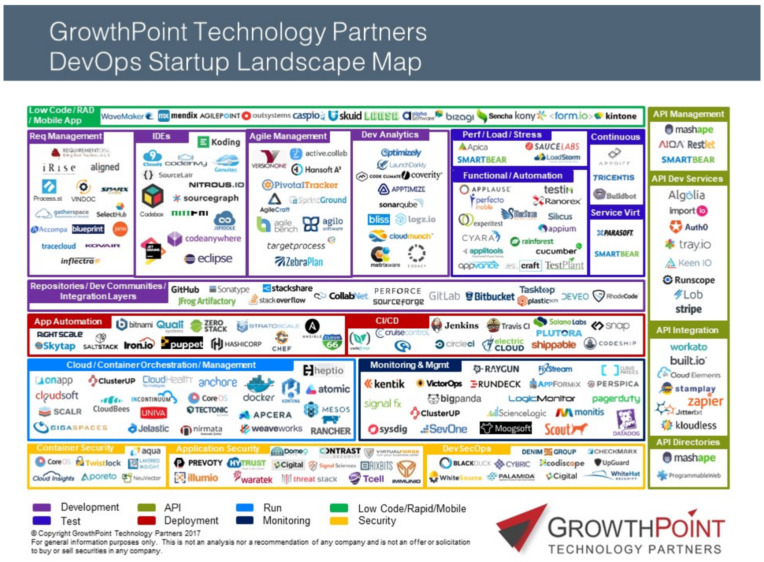

Figure 3: A glimpse of the startup landscape year 2016 and their DevOps tools. [8]

This comes however with its own set of problems, and that is how to glue these components together. They are built for different systems, with different programming languages and uses different design paradigms. The only thing that is guaranteed to be standard, is that there exist no standard.

With that being said, there have been a few market leaders that have amassed a cult following for best practices, and as Xie, Chen, and Neamtiu [7] mentions, Open Source has been driven by personal growth. Something that has been accelerated by having code bases publicly available, and hence allowing newcomers to adopt giants practices. Somewhere within this impression, YAML as a configuration language arose. Shimon Tolts (CTO and co-founder of Datree) once said to The Register [9] "Everything is becoming YAML-fied", when referencing to code developing and deployment being automated. Shimon Tolts based this statement from his company findings after analyzing several million open-source GitHub Projects and finding over 0.5 million repositories with *docker-compose* YAML files for Docker service configuration.

It cannot however be declared that Docker was the pioneer for implementing their configuration through YAML, but they sure have been a vital player for the widespread developer adoption. Especially as the Docker service in itself contributed to one of the previous mentioned cornerstones of the problem between figure 1 and 3. Which is that it consolidated software to be able to be run in a containerized system with pre-built environments. Thusly removing the problem of having tooling that runs on different operating systems.

Another advantage that Open Source enabled is free knowledge. Anyone can view and modify their favorite tooling, and can therefore contribute to newer features without having to be apart of the core project team. This also allowed for third-party providers to implement interfaces as if they were first-class only by knowing how the core of the tooling works.

In spite of all of this, there has yet to exist a marketplace or a repository that follows the philosophy of Unix. The Unix philosophy emphasizes building simple, short, clear, modular, and extensible code that can be easily maintained and re purposed by developers other than its creators [10]. With that being said, there do exist solutions that tries to solve the problem of creating the glue between popular tooling, but the one of the problems that comes with these solutions are that they are not simple nor clear in how data is being piped between the different layers. One of these solutions are Github Marketplace, which aims to becoming a marketplace for application and action steps in their workflow scheme. Their approach is built on-top of human readable directives written in YAML that describes what the workflow scheme should do with said action, as well as the underlying business logic that will manipulate or analyze the provided data.

5 The obscure workflow

GitHub's workflow system is vast. And with its mass, there will intrinsically exist entanglement between development and maintenance. Furthermore, as its workflow is extremely intertwined with the underlying architecture of how GitHub works, it goes without saying that debugging and developing of actions are nothing but laborious. The reasoning for this is two fold. Firstly, its workflow cannot be utilized on a local machine without any third party software. Secondly, its directive data structure (written in YAML) is extremely prone to errors. These problems in conjunction results in a development experience (DX) less than desirable, and far of from the Unix philosophy.

Recall to chapter 2, where the paper discussed about re-usability. YAML isn't built with re-usability in mind as it is only data, and therefore cannot be dynamically assigned. GitHub's spin off to solve this is to combine JavaScript's template strings onto YAML's string scalar. This allows GitHub's workflow system to interpret the directives as a template and replace the keywords with data that comes from the workflow.

Fundamentally, this deepens the complexity of the system, as it first off all isn't transparent of what kind of keywords are allowed to be expressed. This is nothing that YAML or a traditional IDE can spot. It will only be unraveled through testing, which cannot be done locally, as earlier discussed. Furthermore, a traditional IDE won't be able to spot if there is a faulty key in the directive, or if it is missing something. It doesn't know, mainly because of it being just YAML, a data serialization "language". Here again, the developer has to push up changes to GitHub in order to understand what he or she has done. This is far fetched from a first-class development experience.

6 Configure, Unify & Execute (CUE)

Luckily, the author of this paper isn't the first that is displeased with current configuration languages. Queue CUE. It is a new configuration language, built around Directed Acyclic Graph theory (DAG). It focuses on the today's core issues of configuration languages, which accordingly to them are [11]:

- Tooling
- Type Checking
- Abstraction versus Direct Access

Much of the core of CUE has its roots in Google configuration language (GCL), as it was originally designed to configure their underlying cloud provision tooling. But instead they decided to take CUE Open-Source, in order to nourish a community around it.

6.1 Tooling

As automation has become the norm and use-case of configuration languages, the CUE language, APIs and tooling have been signed to allow for machine manipulation. CUE has first-class tooling for data validation, schema definition, code generation and scripting. As well as conversion between popular configuration representations, like JSON and YAML. This allows CUE to consume YAML and produce YAML to those services that still demands that structure, but as the intermediate stage is CUE, it can constrain the configuration by providing an open and declarative scripting layer on top of the configuration layer.

6.2 Type Checking

As the code base grows, the complexity and lack of documentation will pursuit. The same is true for configuration. Yet, there is no rigorous system to validating said configuration data available to the masses without complicating the configuration file.

CUE has taken a different approach, by unifying types and values as one. This gives the CUE language a very expressive, yet intuitive and compact typing capability [11]. As it is built around a DAG, it will push environment & intermediate variables to their specific source. Whenever said variable isn't available in the system, CUE will promptly exit its process and invalidate it, as well as explaining what it misses.

This isn't available as a first-class solution from any other configuration language at the level that CUE serves.

6.3 Abstraction vs Direct Access

A by-product of having a fine-grained typing system is that it allows for layering detailed constraints on top of the native CUE APIs without the need of an abstraction layer. Mitigating from an abstraction layer will speed up the maintenance of the configuration during the design phase as APIs deprecates and new gets added. An API is inevitably prone to drift in the early stages, and therefore it isn't viable to have an ever-changing abstraction layer just for protecting the user against misuse, when the type system is more than capable of handling this.

7 Conclusion

To conclude this essay, the future looks promising. As Julien Bisconti once said during a guest lecture at KTH, CUE will become the successor of YAML. This is because of the simple fact that configuration is highly dependant on network effects. Recall to docker-compose. Furthermore CUE is inherently built to be extensible to feed structured data to other tooling. What this mean is that CUE views configuration more like a contract between users and tool developers, such that the development experience matures as the protocol between applications and-/or tools continues to expand. Fin.

References

[1] Intland Software, *Accelerate your devops pipeline*, https://intland.com/wp-content/uploads/2019/07/devops-infinity-1-1.png, url = https://intland.com/wp-content/uploads/2019/07/devops-infinity-1-1.png, 2021.

[2] Stroustrup, Bjarne, "Software development for infrastructure," *Computer*, vol. 45, no. 1, pp. 47–58, 2012. DOI: 10.1109/MC.2011.353.

[3] O. Ben-Kiki, C. Evans, and B. Ingerson, "Yaml ain't markup language (yaml™) version 1.1," *Working Draft 2008-05*, vol. 11, 2009.

[4] Matthew Burruss, *A powerful python trick: Custom yaml tags pyyaml*, https://matthewpburruss.com/post/yaml/, 2021. [Online]. Available: https://matthewpburruss.com/post/yaml/.

[5] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero, and D. A. Tamburri, "Devops: Introducing infrastructure-as-code," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 497–498. DOI: 10.1109/ICSE-C.2017.162.

[6] Mike Hadlow, *The configuration complexity clock*, http://mikehadlow.blogspot.com/2012/05/configuration-complexity-clock.html, 2012. [Online]. Available: http://mikehadlow.blogspot.com/2012/05/configuration-complexity-clock.html.

[7] G. Xie, J. Chen, and I. Neamtiu, "Towards a better understanding of software evolution: An empirical study on open source software," in *2009 IEEE International Conference on Software Maintenance*, 2009, pp. 51–60. DOI: 10.1109/ICSM.2009.5306356.

[8] Growth Point, technology partners, *Devops startup landscape map*, Access from: https://growthpoint.com/growthpoint/announcing-growthpoints-devops-startup-landscape-map/, 2016. [Online]. Available: https://growthpoint.com/growthpoint/announcing-growthpoints-devops-startup-landscape-map/.

[9] The Register, *Behold, the world's most popular programming language – and it is...wait, er, yaml?!?* https://www.theregister.com/2018/11/19/popular_programming_language_yaml/, 2018. [Online]. Available: https://www.theregister.com/2018/11/19/popular_programming_language_yaml/.

[10] Danny Sheridan, *October 8: Unix philosophy*, Access from: `https://www.factoftheday1.com/p/october-8-unix-philosophy?s=r`, 2021. [Online]. Available: `https://www.factoftheday1.com/p/october-8-unix-philosophy?s=r`.

[11] CUE Lang, *Configuration*, Access from: `https://cuelang.org/docs/usecases/configuration/`, 2022. [Online]. Available: `https://cuelang.org/docs/usecases/configuration/`.