The blasphemy of YAML and other configuration "languages"

DD2482

John Landeholt

April 2022

Contents

1 Introduction

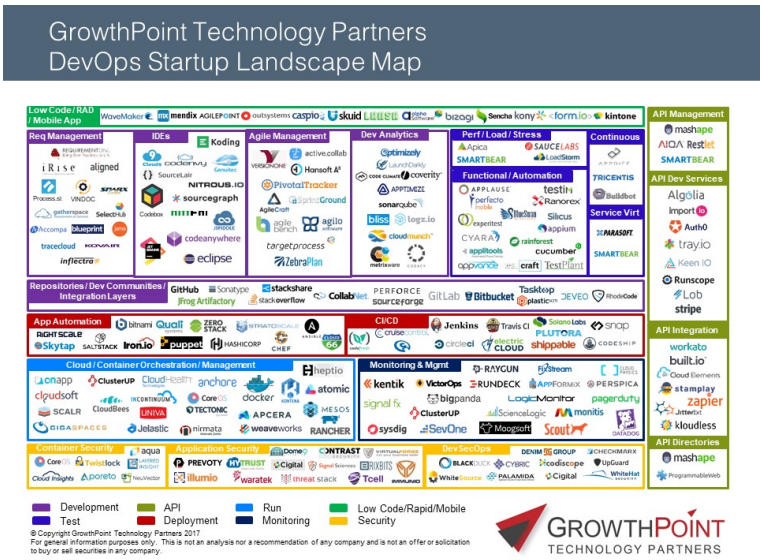

Figure 1: A glimpse of the startup landscape year 2016 and their devops tools. From: Growth Point, technology partners (2016)

A ever-growing problem that new software engineers encounters only first they actually enter the industry is the **lack of standards**. No interface is the other alike. Sure, there exists some flavours that are similar or even super-sets of each other, but the environment is still vastly scattered and divided. We all have probably seen a magnitude of repositories filled to the brim with weird files such as $.*\text{rc}$, $*.\text{yaml}$, $*.\text{json}$, $*.\text{hcl}$, $*.\text{config}$ and Dockerfile.

Why is this? How did this happen? All these file types have their own purpose and are without a doubt useful for their specific use case. The problem is that they all are configuration files that does the same thing in their own novel way.

This essay will research this particular area of development operations because it should be relevant to establish a standard for configuration, preferably as code with a ecosystem built around it by a community. Much like what libraries such as React has done for the web development community.

Configuration files are extremely error prone. Especially as they are usually written manually and does not have any additional syntax capabilities for specific domain configurations such as Github actions, Terraform, Kubernetes, Docker and likes among them that can be seen in figure 1.

Sure, you can constrain an IDE to validate keys in the case of Github actions (YAML), but it doesn't constrain the actual value that is appended to the key, and thusly creates a window of uncertainty.

This level of uncertainty can be intimidating, especially when it is applied to the production configuration. But wait just for one minute Mr. Landeholt! Can't we just build a tool that prevents this from happening? Of course you can! But don't you see the irony in it? It should had been taken care by the configuration orchestrator! But not at all, we deliberately bypasses this by entering the era called tool explosion accordingly to Kersten (2018).

1.1 Research question

DevOps as a field has grown exponentially for quite some while now. Unfortunately, everything hasn't followed pursuit. One of these subjects are configuration. Google SRE (2018) calls it configuration-induced toil, which they explain will eventually grow into "complexity toil", which is a concept that compounds with growth of the organization and-/or project. It stems from the fact that the number of applications, servers and services will over time increase, and that will intrinsically make configuration more complex and verbose when each application and-/or service uses their own configuration file. Meaning that you might have to update multiple configuration files in multiple locations with the same information. Those that are familiar with the DRY (don't repeat yourself) concept, understands that this is a problem. The logic shouldn't be that each configuration has irrelevant duplicated data. It should be shared across each configuration file from one source of truth. I.e it should be reusable. Configuration systems are one of the backbones of large platforms, and if they become miss-configured, it can result in severe downtime. In 2010 this happened to Facebook (Facebook Engineering, 2010), resulting in 2.5 hours of unreachable services. What they learnt from this was that they had to rethink their configuration strategy for the automated systems.

Facebook is not alone in this problem, as the problem lies in systems having hundreds of configuration options, which gives an infinite set of variants.

SAYAGH et al. (2020) literature review examines the industry and through an exhaustive survey that concluded that configuration engineering will in the future be one of the most important research topics, where the focus for now is how to prevent configuration failures.

Therefore, this essays research question will continue in the footsteps of SAYAGH et al. (2020), by answering the following questions:

- What requirements are necessary for creating a ubiquitous configuration language?
- What prevents configuration failures to occur?

1.2 Problem statement

The current used "languages" are not feature rich enough to accommodate to today's automation and-/or configuration. They are for the most part data serialization languages (Red Hat, 2021), hence the quotation marks around the word language.

But how can we then pass dynamic values to our configurations? Short answer: We cannot. Long answer: lets recall to the previous section: Tool explosion.

There is a tool for infusing that functionality: **template engines**. But the problem still prevails. The configuration orchestrator is not the one doing this, but yet another external tool that your project now has to maintain or rely on.

There is nothing wrong with modern template engines as a tool in general. However, for configuration manipulation, it is an extremely fugitive solution. We need something more robust and ubiquitous. But what could that be?

Before we can answer that question, we do on the other hand also need to state the obvious. Why does this problem matter in the first place?

Take this very real scenario as an example: Management has decided that the company no longer will work with the current underlying infrastructure and wants you, the developer to relocate their infrastructure to another cloud provider. That shouldn't be hard, right?

Well, if you are in luck the two cloud providers shares the same data serialization schema, but that is however very unlikely as many providers are using their own proprietary solutions as Opara-Martins et al. (2016) mentions being one of the usual mistakes organizations makes, which does inhibit the portability between providers. This might not be so problematic for weekend projects, but take a second to imagine it for a more complex system. A complex system will inherently have a underlying complex configuration, meaning that when your organization has hundreds of services (not just one), it will start to matter to having it more ubiquitous than what simple data serialization languages can provide.

So back to the main question; what can be done to configurations in order to make it more robust and ubiquitous to use in many different systems?

The hypothesis is that:

- It should be a language that have a set of directives that build up reusable configurations at a large scale. I.e code generation and validation
- It is a orchestrator that can configure systems cross platform without any external custom logic providers. I.e templating and validation

2 Reasoning

As the industry is moving into a new paradigm: Infrastructure as Data (Adri Villela, 2021). The author stipulates a rhetorical, yet important question; "are we actually writing code?" The answer is a clear no when it comes to provisioning cloud infrastructure. What are we then writing you might ask? We are writing data-driven policies stored in human readable text that represents resources that later on is provisioned through an API.

The author does however not see the full picture. She argues that infrastructure is static, i.e never changing. This is as far from the truth as one can go. Infrastructure is not static. It is incremental. And as such, the underlying medium cannot only be a data serialization language such as YAML, JSON or HCL. It can at the same time not be a programming language such as Python for obvious reasons. It has to be language and-/or platform agnostic.

2.1 Configuration needs a different language model

As the current paradigm states, we no longer code our infrastructure provisions, but instead with a instruction set in the form of key-value pair maps. In order to conform to this we need a language that is based on First-Order Logic (FOL) (Simran, 2006). The reasoning for this is that it must have a set of directives that enables re-usability. Considering that a system is built on-top of multiple domains, it should fit perfectly to view it as a First-Order Logic, where the agent sees thing as true or false in a partial view of the source. Therefore it should be possible to describe data as formulas but at the same time as pure values. What FOL brings to the table is a set of axioms and inference rules that in theory should support first-class support of the following: validation, templating and consequently code generation. This is however highly assumptive by the author, and should be contested.

2.2 The language must be type extensible

In order for a language to be easily statistical analyzed and validated, it should be built around the foundation of type checking without bloating the configuration file with regards of boilerplating.

An example of a strikingly verbose configuration "language" or rather specification is the OpenAPI specification that is built on-top of the JSON data serialization language. It is capable of combining types and values, but it comes with a cost. The cost being a much more verbose language. They call it for a Component Object, but what it in reality is just a regular map with a set of fixed properties (keys). A OpenAPI specification is valid JSON, but not necessarily the other way around, as the OpenAPI specification requires a set

of fixed keys to exist. See appendix A as an example of the OpenAPI way of expressing custom type definitions.

However it (OpenAPI) may be working, it takes a lot of excessive boilerplate syntax to write just to define two types with a total of with two attributes each. A language should be concise enough, such that configuration files are not cluttered with boilerplate that does no good for the actual configuration, but only add complexity for the users.

Types are a necessity, as they protect users from errors when defining said configurations and will also serve as automatic documentation if the syntax is rich enough to be human readable. It get things rolling in regards to setting up constraints for what specific configuration options are allowed to be. This cannot be widely said about YAML and JSON without diffusing the configuration with fluffed syntax, as Appendix A simply reveals.

2.3 The language should depart from the inheritance paradigm

Most configuration languages are rooted in the inheritance paradigm, which by the opinion from the author creates more problems than it actually solves. The reasoning for this assumption is that defining abstraction layers comes with an upfront cost. It also hides the data from the user behind objects. It is no longer data, but some sort of abstraction that is built on-top of something unknown to the user. Rich Hickey (2012), the creator of the programming language Clojure argues that data (or more specifically values) should be semantically transparent. They should be easy to convey and humanly interpretable. This is something that is very valuable for a configuration language to have. It should be human readable and at the same time it should be machine interpretable. The author argues that this is not possible with the current flavours of configuration languages. This is especially true for large scale projects, as inheritance will create complicated layers of abstractions as the application grows.

2.4 Findings

By outlining the specified requirements mentioned in the above sections, and also listed below, the aim of this essay is to evaluate whether there exists a configuration language that ticks the majority of these necessities. The following requirements was detected during the reasoning part:

- the language should follow the current paradigm of data driven infrastructure.
- the language should be agnostic
- the language should First-Order Logic, such that it allows for re-usability

- the language should have first-class validation, templating and code generation
- the language should be type safe
- the language should be compact
- the language should be easily interpretable for both humans and automation.
- the language should be easy to convey by not abstracting away data

With this information in hand, we can start to deduce whether it exists a solution that answers one of the research questions; "What prevents configuration failures to occur?". This was done through a nominal querying for widely adopted configure file formats where the following 8 "languages" were found:

- YAML - Yet another markup Language
- JSON - JavaScript Object Notation
- HCL - HashiCorp Configuration Language
- CUE - Configure Unify Execute
- ~~LUA~~ (LUA is considered being a programming language at first)
- Dockerfile

These where the most adopted configuration flavours that was found by querying the following: "config language", "github config language", "configuration languages", "extensible configuration languages", "most popular configuration languages".

A limit of page rank was set to ignore results after page 5, in order to not find any obscure flavours.

From this established set of languages we perform a basic analysis for whether the language covers the listed requirements and presents it in appendix B.

Surprisingly, there seem to exist a configuration language that ticked all the requirements. It is called CUE and is accordingly to their documentation a data validation language with its roots in logic programming. It does however state that it is not a general-purpose programming language (such as Python). What CUE aims to be is a inference engine that can be used to validate data in code or to be included as a code generation pipeline. CUE is a fork of the dominant configuration language that is being used at Google.

Furthermore CUE does not distinguish between values and types. Which is extremely valuable, as it should then allow mixing between types and values. This is something that the other languages fail to do in an elegant manner. As previously mentioned, it is not a general programming language, and but a data validation language, meaning that it should be conformed to Infrastructure as Code.

What CUE has against its competition is that it has first-class support for all the above mentioned directives! It's crazy to think that this isn't widespread already!

3 Conclusion

As a conclusion for this every limited empirical analysis is that most of the configuration languages that exist today does tick a majority of the requirements detected. However most of them comes with a drawback and that is that they become extremely verbose in the process. A peculiar find was that it actually exists a configuration language that managed to tick all the requirements and that it has been developed by Google Engineers. The language is called CUE and aims accordingly to their community to gain market shares in the configuration space by achieving ubiquity. Meaning that it should be widely used and integrated in a vast set of different domains. One of the ways CUE differs from its competition is its ecosystem. It is built in such a way that a configuration can be defined as a package for a specific domain and then reused across multiple workflows.

Another feat that CUE has going for it is their typing structure, that accordingly to their discord community will in the future be developed into a Language Server Protocol, such that IDE's like Visual Studio Code will be able to parse and analyze A CUE structure, and apply intellisense capabilities. This in itself will be a major breakthrough for configuration languages, as it would actually make the language typed, just like typescript introduced types to JavaScript, which made it more acceptable in the mainstream.

A OpenAPI Specification

```
{
    "components": {
        "schemas": {
            "Person": {
                "type": "object",
                "required": ["name"],
                "properties": {
                    "name": {
                    "type": "string"
                    },
                    "address": {
                        "\$ref": "#/compontents/schemas/Address"
                    }
                }
            },
            "Address": {
                "type": "object",
                "properties": {
                    "streetName": {
                        "type": "string"
                    },
                    "zipCode": {
                        "type": "integer",
                        "format": "int32"
                    }
                }
            }
        }
    }
}
```

Listing 1: Example of a verbose configuration language that enables custom type definitions (Swagger, 2022).

B Result matrix

Lang	Data	Agnostic	Tooling	Type safety	Compact	Easily interpetable	non-inheritant
YAML	X	X	X		X	X	
JSON	X	X	X				
HCL	X	X	X		X	X	
CUE	X	X	X	X	X	X	X
Dockerfile		X					

References

Adri Villela (2021). Shifting from infrastructure as code to infrastructure as data. Access from: https://medium.com/dzerolabs/shifting-from-infrastructure-as-code-to-infrastructure-as-data-bdb1ae1840e3.

Facebook Engineering (2010). More details on today's outage. Access from: https://www.facebook.com/notes/10158791436142200/.

Google SRE (2018). Site reliability engineering: Workbook. Access from: https://sre.google/workbook/configuration-specifics/.

Growth Point, technology partners (2016). Devops startup landscape map. Access from: https://growthpoint.com/growthpoint/announcing-growthpoints-devops-startup-landscape-map/, url = https://growthpoint.com/growthpoint/announcing-growthpoints-devops-startup-landscape-map/.

Kersten, M. (2018). A cambrian explosion of devops tools. IEEE Software, 35(2):14–17.

Opara-Martins, J., Sahandi, R., and Tian, F. (2016). Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective. J Cloud Comp, 5(1).

Red Hat (2021). What is yaml? Access from: https://www.redhat.com/en/topics/automation/what-is-yaml.

Rich Hickey (2012). Value of values. Access from: https://github.com/matthiasn/talk-transcripts/blob/master/Hickey_Rich/ValueOfValues.md.

SAYAGH, M., Kerzazi, N., Adams, B., and Petrillo, F. (2020). Software configuration engineering in practice interviews, survey, and systematic literature review. IEEE Transactions on Software Engineering, 46(6):646–673.

Simran, M. (2006). Topics in ai - logic programming. Access from: http://www.doc.ic.ac.uk/~cclw05/topics1/first.html.

Swagger (2022). Openapi specification. Access from: https://swagger.io/specification/.