Kubernetes Resource Validation

Abstract

Linters and static code analysis in some domains have proven to help developers catch bugs and speed up the development process. Kubernetes is growing in popularity, but the tooling for validating configuration files is limited. Most tools that exist only look at plain local YAML-files. This can is a problem because you typically only have a subset of the full infrastructure configuration on your local machine, so many of the object references/dependencies will naturally be broken locally, and thus the tools will often give incorrect/inadequate valuation. Furthermore, developers may structure their YAML files using *kustomize*, which only generates Kubernetes objects at runtime and thus never stores the plain YAML file locally. This results in most tools not checking check Kubernetes objects generated by *kustomize*. This paper presents a prototype of a Visual Studio Code Extension published in the VS Code Marketplace. The tool highlights object references in YAML-files based on plain local YAML-files, objects that exist in running clusters, and objects generated output by *Kustomize*. The extension is tested on X people for two weeks to a month. X people filled out a survey with feedback. The main response was that reference-highlighting is useful and can prevent errors in certain situations. The tool is not perfect and is not find all references it is still useful. It proves that even simple highlighting can limit bugs and thus increase productivity. Further research and development are needed to cover

The source code can be found on *GitHub*[1], and it can be installed from the *VS Code Marketplace*[2]).

Keywords

- Open-Source Software, IDE Extension, Code Linting, Static code analysis, Code Validation, Kubernetes

Introduction

Background

Container Orchestration and Kubernetes are continuously getting more popular during the last couple of years. Around 25-30% of professional developers use Kubernetes (25.45% [5], 30% [6]), and around one-fourth of the remaining wants to work with Kubernetes in the future [7]. So, we can expect that more and more developers will be deploying and configuring infrastructure on Kubernetes through YAML. However, Kubernetes is known for being remarkably complex, and hence the learning curve can be quite steep. In a survey done by StackOverflow it is shown that 74,75% of the developers who use Kubernetes "loves it" and the remaining 25,25% "dreads it" [5] Why the ~25% of the participants *dread* Kubernetes is not specified in the survey, but anything that would lower that percentage is worth striving for.

The ecosystems around Kubernetes are constantly evolving, and it can be difficult to keep up with the newest tools and features. When building workloads aimed at running on Kubernetes, most configuration is done in YAML and revolves around Kubernetes Object Definitions. This means we, as a community, need to ensure that we have the proper tooling for validating this infrastructure. Both to speed up the development process but also to reduce the number of errors that goes into production. The sooner the bug is found in the software process, the cheaper it is to correct [8].

Problem

It is very easy to make mistakes when deploying to Kubernetes. There are strict rules of what fields you can add in your Kubernetes Object Definition, but there are no checks that the values provided are valid (besides the fact that it needs to be the correct type) - e.g., your *Pod* can request environment variables from a *Secret* that does not exist, or your *Services* can point to *Pods* that do not exist.

Terraform does static validation checks, so you will be informed about your broken references before you apply your code. By design, Kubernetes does not do that and cannot do that. Kubernetes objects are not expected to be created at the same time or in a given order, so we cannot talk about a "compile" time. Kubernetes is managed with Control Planes, and there is no order to when which resource is created/scheduled. In Kubernetes, your references will often not exist on creation time but will only be created at a later point. This explains why there is no static validation by default when creating Kubernetes resources.

Kubernetes object definitions are written in YAML. YAML is a data serialization language that only knows the concepts of primitive data types like arrays, strings, numbers, etc., and therefore does not know the concept of references. All references declared in your Kubernetes objects definition is written as a string, with no type checking or validation. This means there is no built-in reference validation in YAML.

Since most IDEs, Kubernetes plugins, or tools do not verify any of the magic strings in your YAML-files, it can be quite cumbersome to debug the code. It typically involves a human manually reading through all magic strings until you discover that there is a typo or each of your resources exists in two different namespaces and, therefore, can't communicate. No public and free IDE feature or extension exist that checks this.

Furthermore, developers often use tools like *Helm* and *Kustomize* for templating their YAML-files. This is done so multiple configurations can inherit/share common code across different configurations. This means that an object referenced in a YAML file may not exist in any of the plain files but may only be generated on "runtime" when one of the templating tools is used. This makes it much harder to give valuable information to the developer because Kubernetes objects' names are dynamically generated by the tool. I have not found any free extension that tries to tackle this challenge, so currently, developers don't get any assistance validating their YAML-files live while coding when using templating tools like *Helm* and Kustomize.

Currently, plenty of open-source validation tools exist for Kubernetes Object Definition, but most of them are Command Line tools and do not give continuous feedback live while the developer is coding. Furthermore, the tools only validate the definitions based on the local files provided to the tool and not what is actually running in a Kubernetes Cluster. Typically, you only have a subset of the full infrastructure configuration on your local machine, so many of the code references and objects will naturally be broken, and the tools will give incorrect/inadequate valuation. So in order to overcome this, we need to look outside the borders of the IDE/current folder and look at what objects exist in a running Kubernetes cluster.

This paper will showcase a prototype of an extension made for *Visual Studio Code* (VS Code) that validates object references in Kubernetes Object Definitions continuously while the developer is

[1] https://github.com/dag-andersen/kubernetes-reference-highlighter

[2] https://marketplace.visualstudio.com/items?itemName=dag-andersen.kubernetes-reference-highlighter

typing. The tool highlights object references in YAML-files based on plain local YAML-files, objects that exist in running clusters, and objects generated output by *Kustomize*. The aim is to provide the developer with valuable information that reduces the number of bugs encountered when deploying and speeds up the debugging process.

The goal of this paper/tool is not to write a perfect and fully optimized Code Linter but instead demonstrate that reference highlighting can limit the amount of bugs/issues when working with Kubernetes.

Related Work

Why developers use Linters

In dynamic languages like JavaScript, where there are no *compile time*-checks, it has been shown that developers use static analysis tools (*Linters*) like *ESLint* to prevent errors [1]. Linters like *ESLint* can be integrated with popular IDEs like VS Code or IntelliJ IDEA. Linters and tools should give feedback as fast as possible, and it should preferably be integrated with the IDE [3]. Preventing Errors is said to be the number one reason to use a linter because it catches the bugs early on, so you don't have to spend time debugging it on runtime [1]. The linter gives the developer continuous feedback on whether their code has errors or not. So even though a language like JavaScript does not have static compile time checks, it is still possible to reduce a substantial number of errors by using Linters running in your IDE.

Similar results have been shown in studies related to Android programming. Android Develops report that they use linters to save time by detecting bugs and identifying unused resources [2]. Furthermore, "Developers always want to maintain a good reputation among their peers, and the linter can help them to do that", by ensuring that their work meets the expectations of their superiors and colleagues.

Why developers do not use Linters

Studies have shown that some of the main reasons why a linting tools are underused is that they produce too many *false positives* and the amount of warnings are too high, which overloads the developer [3].

Users will stop using a tool if they lose trust in it, because of too many false positives. Also "false positives are not all equal". To get people to trust and use the tool, it is important that the user don't experience false positives on simple issues. It is less important if it is a false positive on a complicated issue [4].

Furthermore, the user will interpret correct feedback as false positive if they don't understand the feedback. "A misunderstood explanation means the error is ignored or, worse, transmuted into a false positive" [4]. "If people don't understand an error, they label it false" [4]. So, it is very important that the tool provide easy to understand feedback if the user should trust the linting tool.

Other reasons developers don't use linters are that they can be difficult to configure. [3]

Existing tools

The tools listed below aim to help the developer create correct/valid Kubernetes Object Definitions and tackle some of the challenges explained in the *problem*-section. The tools are collected by reading blog posts online and searching on GitHub.

- **JetBrains IntelliJ IDEA Kubernetes Plugin** *[website[3]]*:
  The full version of IntelliJ that includes support for plugins is a paid product and is therefore not publicly accessible. JetBrains' plugin only takes local files into account when highlighting references. The plugin does not check your current cluster or build anything with kustomize. But JetBrains' plugin comes with extra functionality that lets the user jump between files by clicking the highlighted references.

- **KubeLinter** *[GitHub[4]][Validation Checks[5]]*:
  KubeLinter is an open-source Command Line Tool that validates your Kubernetes manifests. KubeLinter is a feature-rich tool that informs and warns of all sorts of issues that may be their code. KubeLinter can validate Helm Charts but cannot validate Kustomize templates. The tool only checks the local files it is provided, so it does not know what already exists in a cluster. KubeLinter detects "dangling references", so it informs the user if a reference does not exist. That is good, but often you do not have the whole infrastructure configuration locally, meaning that the dangling references that it detects may not be dangling after all because the references actually exist in the cluster. This means that the KubeLinter can report false positives (detecting an issue that is not an issue), which can result in the developer not trusting the tool and thus disabling the tool as described in the *Related Work*-section

- **kube-score** *[Github[6]][Validation Checks[7]]*:
  kube-score is a tool that performs static code analysis of your Kubernetes object definitions. Even though there are over 30 different checks, the value is quite limited. The tool seems to only validate all the Kubernetes object definitions individually in a vacuum and not the relation between them. Furthermore, just like **KubeLinter**, it is a Command Line tool and, therefore, can be integrated as part of a testing-pipeline but does not provide any live feedback when coding in an IDE.

- **Kubevious** *[GitHub[8]][docs[9]]*:
  This tool comes with 44 built-in Kubernetes Resource validation checks. The checks include cross-file issues like if a *Service* points to two different deployments at the same time. Kubevious has a powerful toolset just like KubeLinter, but this tool has a GUI and needs to be run in a cluster or as a stand-alone program. It can only run its checks on resources already deployed to Kubernetes and therefore does not do any live validation for the developer while coding. This tool is mainly for finding and debugging issues after it has already been deployed.

- **Conftest** *[GitHub[10]]*, **Copper** *[GitHub[11]]*, and **Config-lint** *[GitHub[12]]*:
  These three tools are similar, and they try to validate Kubernetes Manifests/YAML, but none of them come with built-in checks, so the users have to make their own "rules" that the YAML will be validated up against. All three tools are Command Line tools, so they do not provide any live feedback but only validation when the command is run. These tools

[3] https://plugins.jetbrains.com/plugin/10485-kubernetes
[4] https://github.com/stackrox/kube-linter
[5] https://docs.kubelinter.io/#/generated/checks
[6] https://github.com/zegl/kube-score
[7] https://github.com/zegl/kube-score/blob/master/README_CHECKS.md
[8] https://github.com/kubevious/kubevious
[9] https://kubevious.io/docs/built-in-validators
[10] https://github.com/open-policy-agent/conftest
[11] https://github.com/cloud66-oss/copper
[12] https://github.com/stelligent/config-lint

accelerate when you want to build your own custom checks (maybe on your own Custom Resource Definition (CRDs)).

None of the tools listed above manage to do reference checking in plan local files, templating tools, and actual running clusters and, at the same time, is free to use and runs in a continuous manner (gives live feedback).

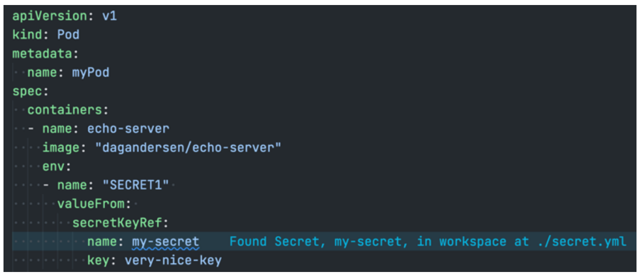
Prototype

This section will describe a prototype of the *Kubernetes Reference Highlighter* VS Code Extension.

Overview

The tool is built as a *Visual Studio Code* (VS Code) extension since VS Code is free and is the most popular IDE/Editor in the industry [9]. The extension is developed in Typescript since that is the default for developing VS Code Extensions since *VS Code* is written in Typescript.

The extension highlights the name of an object if it has found the referenced object in a local file, a cluster, or in output generated by kustomize. The highlighting updates in real-time while you type and gives you constant feedback on whether your references are found or not.



```
apiVersion: v1
kind: Pod
metadata:
  name: myPod
spec:
  containers:
  - name: echo-server
    image: "dagandersen/echo-server"
    env:
    - name: "SECRET1"
      valueFrom:
        secretKeyRef:
          name: my-secret    Found Secret, my-secret, in workspace at ./secret.yml
          key: very-nice-key
```

Figure 1: Shows how the extension highlights the string `my-secret` because it found an object with that name `./service.yml` in the local VS Code workspace. In this figure, the VS Code extension, Error Lens, is enabled to better visualize the highlighting.

A tool like this is close to useless if the developer cannot rely on the validity of the feedback it gives. Therefore, this tool aims only to highlight a string if it is sure that the reference exists. It is better to have *false negatives* than *false positives* in this case.

This extension makes use of *VS Code Diagnostics* for highlighting strings. Diagnostics are the squiggly line under code lines, and the severity can either be marked as *Error*, *Warning*, *Information*, or *Hint*. This extension uses *Information* for all object reference, and it only uses *Error*, when `Kustomize build <path>` fails. Since these Diagnostics are built into the VS Code IDE, then other extensions or tools can also read and act upon the information provided by this extension. This makes this extension integrate well with other VS Code features/extensions. In particular, I suggest installing *Error Lens* together with this extension to visualize the diagnostics even better for the user.

The extension's architectural structure can be split up into two components/steps: *Reference Collection* and *Reference Detection*. *Reference Collection* is the step that collects and builds a list of all the objects the extension can find. This step is triggered every time you save a YAML-file in the workspace. *Reference Detection* is the step that parses the currently open file and tries to find references that match objects found in the *Reference Collection*-step. This step is triggered every time a YAML-file is changed in the workspace.

Reference Detection

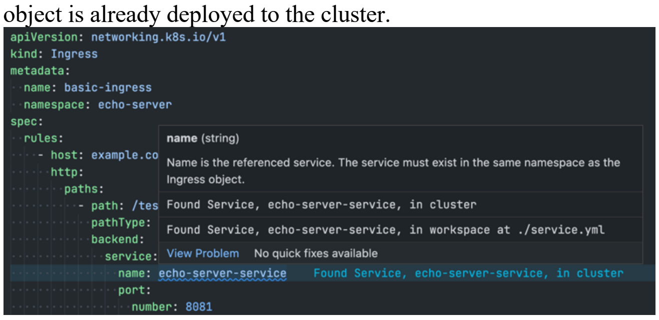
As of Version 0.0.10, the extension highlights references to Services, *Deployments*, *Secrets*, and *ConfigMaps*. More *Kinds* can easily be added in the future.

The extension uses YAML-parsing libraries and Regex as the main method for reading the files in the workspace.

All resources are namespace-sensitive. An object-reference will not be highlighted if the resource exclusively exists in another namespace.

Raw hostname addresses pointing to *Services* are a bit different and the most difficult ones to handle. You can either call a *Service* by using its name as an address if you call it from inside its namespace (`my-service`) or call it from anywhere within the cluster by suffixing the *Service*-name with its namespace (`my-service.its-namespace`). This tool will only highlight an address pointing to a service if it exists inside the same namespace or the address is suffixed with its namespace. This is an important distinction that can prevent many issues related to traffic routing, where you try to access a service that may be in a different namespace.
Raw addresses are, in general, difficult to detect because they can be written as part of a string in many places. E.g., the address of a *Service* can be written as part of an environment variable accessible from a *Pod*, or it can be stored as a field in a *ConfigMap*, which is read from a Pod. Detecting these references is more challenging than the other *Kinds*, and there is a bigger risk of detecting false positives.

If the string matches with multiple objects, the extension will show all the references. A reference can, as an example, exist both in the running cluster and in the local file at the same time, because the object is already deployed to the cluster.



```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: basic-ingress
  namespace: echo-server
spec:
  rules:
    - host: example.co
      http:
        paths:
          - path: /tes
            pathType:
            backend:
              service:
                name: echo-server-service    Found Service, echo-server-service, in cluster
                port:
                  number: 8081
```

```
name (string)

Name is the referenced service. The service must exist in the same namespace as the
Ingress object.

Found Service, echo-server-service, in cluster

Found Service, echo-server-service, in workspace at ./service.yml

View Problem    No quick fixes available
```

Figure 2: Multiple matches on `echo-server-service`

Reference Collection

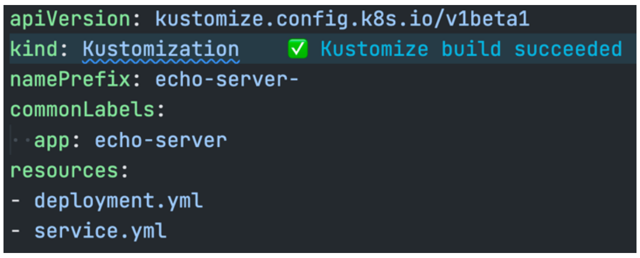
The tool can read plain YAML-files in the workspace, read generated output by Kustomize, and fetch objects from a running cluster. The tools do not parse the output of *Helm Charts*. Further research and development are needed to conclude if such a feature is feasible.

- **Cluster Scanning**:
  The extension uses the user's current context found in the kubeconfig to call the Kubernetes Cluster and collects the names of the Kubernetes objects that way.

- **Workspace Scanning**:
  The extension traverses all files ending with `.yml` or `.yaml` in the currently open workspace in VS Code. It collects the *kind*, *name*, and *namespace* of all the Kubernetes objects found in the files.

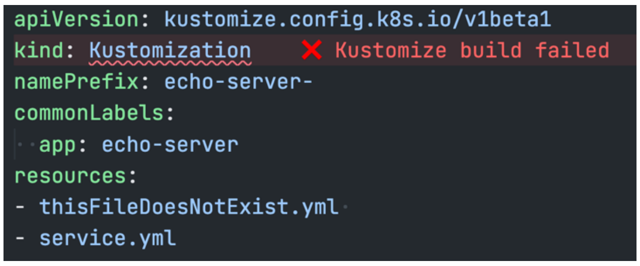
- **Kustomize Scanning**:
  KRH traverses all files named `kustomization.yaml` or `kustomization.yml` in the currently open workspace in VS Code and builds the kustomize files. All the Kubernetes objects

generated by *kustomize* are collected. In addition, the extension will also highlight if the Kustomization-file builds or not. If *Kustomize* is not installed on the computer and part of the default shells PATH, Kustomize Scanning will be disabled.



```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization     ✅ Kustomize build succeeded
namePrefix: echo-server-
commonLabels:
  app: echo-server
resources:
- deployment.yml
- service.yml
```

Figure 3: Kustomize Builds Successfully



```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization     ❌ Kustomize build failed
namePrefix: echo-server-
commonLabels:
  app: echo-server
resources:
- thisFileDoesNotExist.yml
- service.yml
```

Figure 4: Kustomize Builds Unsuccessfully

Each time you save a YAML-file in VS Code, the extension will update its internal list of Kubernetes objects. Each of these scanning techniques can be disabled through the *Command Palette* or through settings.

Kustomize is extra challenging to handle because it is built on the idea of *bases*[13] and *overlays*[14]. Overlays use a base-configuration and then modify the existing configuration. Multiple overlays can use the same base, and a base has no knowledge of the overlays that refer to it. Overlays can be chained to infinity, which means that when the extension builds a kustomization, it doesn't know if this is the final configuration or if one or more overlays override it in a different folder. Traversing the dependency tree of overlays depending on each other is beyond the scope of this Paper. Instead, the tool will inform the user of references it finds in all the kustomization layers, and then it is up to the developer to check which layer he/she meant to refer to.
The fact that the extension highlights which kustomize-files build or do not build can be very useful in debugging a long chain of kustomize layers.

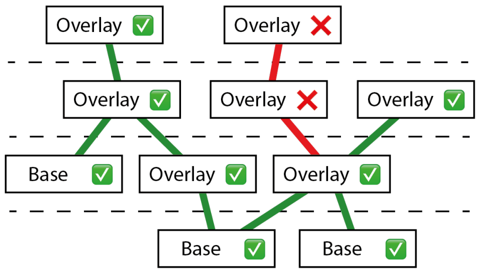


Figure 5: Visualization of how the extension shows where overlays fail in a chain of overlays. The bases are at the bottom levels, while each layer above the bases is an overlay.

Evaluation

The proper way of implementing a linter is to use some kind of abstract syntax tree and pass tokens, but with the limited amount of time for this project, Regex-matching is acceptable for a prototype. Using Regex has the risk of producing false positives or false negatives. If there is an edge case the regex does not catch, then a reference will not be found and produce a false negative (not highlighting something that should be highlighted).

To evaluate if this VS Code extension reference highlighting can limit the number of bugs/issues when working with Kubernetes, I have asked developers through slack-channels, Reddit, and other social media/groups to try using the tool in their daily work for 2-4 weeks. Since the tool is publicly available in the VS Code Marketplace, it is possible that people have also found it by chance.

When the extension has been installed for 15 days, a message will ask the user if they would like to fill out a survey. If they click "open", the survey will open in their browser. If they click "later", they will be asked again in 5 days. After the user has opened the survey or clicked "later" 3 times, the message will no longer pop up. The survey was created in Google Forms and was also available from the extension's marketplace page.

As of November 9, 2022: The Reddit post[15] has shown up in 12.500 people's feeds, gotten 17 upvotes and a few comments. The extension has been installed by 23 people, and X people have filled out the survey.

Survey Results

Question \ participant	person1	Person2	Person3	Person4
Years of professional experience				
Years of professional experience with Kubernetes				
role/job position				
How likely are you to recommend this to others working with Kubernetes? (1-5)				
Over the last month, did the extension help your daily work?				
How many bugs did it help you catch				
How often does the extension give false positives (highlighting a wrong reference)				
Percentage estimate of how often it gives false negatives (not highlighting a reference that is actually there)				
What type of scanning do you find most useful? (Cluster, plaintext/workspace, Kustomize)				

< Insert evaluation on the results >
< Insert evaluation on the results >
< Insert evaluation on the results >
< Insert evaluation on the results >
< Insert evaluation on the results >

< Insert evaluation on the results >
< Insert evaluation on the results >
< Insert evaluation on the results >
< Insert evaluation on the results >
< Insert evaluation on the results >

[13]https://kubectl.docs.kubernetes.io/references/kustomize/glossary/#base
[14]https://kubectl.docs.kubernetes.io/references/kustomize/glossary/#overlay
[15]https://www.reddit.com/r/kubernetes/comments/yoy0w8/kubernetes_yaml_linter_for_vscode/

< Insert evaluation on the results >
< Insert evaluation on the results >
< Insert evaluation on the results >
< Insert evaluation on the results >
< Insert evaluation on the results >
< Insert evaluation on the results >
< Insert evaluation on the results >
< Insert evaluation on the results >
< Insert evaluation on the results >
< Insert evaluation on the results >

Conclusion

<Here I will add some more>
Based on the fact X out of X persons reported they experienced the bug helped them catch a least 1 bug and no participant straight up said it had a negative impact on their productivity, I would consider this extension a success. This paper shows that even with a simple extension based on Regex, it is possible to give developers valuable live validation of Kubernetes Object Definition to catch bugs and increase productivity.
Further research and development are needed to evaluate if an even better extension (with more complicated YAML parsing) would help developer catch even more bugs.

References

1. K. F. Tómasdóttir, M. Aniche and A. van Deursen, "Why and how JavaScript developers use linters," 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2017, pp. 578-589, doi: 10.1109/ASE.2017.8115668. https://pure.tudelft.nl/ws/files/26024522/ase2017.pdf
2. S. Habchi, X. Blanc and R. Rouvoy, "On Adopting Linters to Deal with Performance Concerns in Android Apps," 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2018, pp. 6-16, doi: 10.1145/3238147.3238197. https://lilloa.univ-lille.fr/bitstream/handle/20.500.12210/23088/https:/hal.inria.fr/hal-01829135/document?sequence=1
3. B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in 2013 35th International Conference on Software Engineering (ICSE). IEEE, 2013, pp. 672–681. https://homepages.dcc.ufmg.br/~figueiredo/disciplinas/2019b/ese/paper1joao.pdf
4. A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. HenriGros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," Communications of the ACM, vol. 53, no. 2, pp. 66–75, 2010. https://www.cs.jhu.edu/~huang/cs718/spring20/readings/bugs-realworld.pdf
5. StackOverflow Survey 2022, https://survey.stackoverflow.co/2022/#section-most-popular-technologies-other-tools
6. SlashData's Developer Economics survey, https://developer-economics.cdn.prismic.io/developer-economics/527f60d6-d199-4db8-bd31-6dde43719033_The+State+of+Cloud+Native+Development+March+2022.pdf
7. StackOverflow Survey 2022, https://survey.stackoverflow.co/2022/#most-loved-dreaded-and-wanted-tools-tech-want
8. https://deepsource.io/blog/exponential-cost-of-fixing-bugs/
9. https://survey.stackoverflow.co/2022/#most-popular-technologies-new-collab-tools