

1. Abstract

Linters and static code analysis in some domains has proven to help developers catching bugs and speeding up their development. Kubernetes is growing in popularity, but the tooling for validating configuration files is limited. By design Kubernetes will accept Kubernetes-manifests with broken references because there is no order to the creation of objects, so a missing resources may exist in a short while. Furthermore, you typically 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 often give incorrect/inadequate valuation. The tools that exist only looks at plain local YAML-files. This paper presents a prototype of an Visual Studio Code Extension published to the VS Code Marketplace. The tool highlight references in YAML-files based on plain local files, objects that exists in running clusters, and objects generated generated output by *Kustomize*. The extension is tested on X people for a month. A small survey is conducted based on their response and feedback. The main response was that reference-highlighting is useful and can prevent errors in curtain situations. The tools is not perfect and does 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 is needed to ...

- The source code can be found on [Github](#) and it can installed from the [VS Code Marketplace](#).

3. Keywords

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

4. 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% [[StackOverflow Survey 2022](#)], 30% [[SlashData's Developer Economics survey](#)], and around one-fourth of the remaining wants to work with Kubernetes in the future [[StackOverflow Survey 2022](#)]. 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" [[source](#)]. 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 [[source](#)].

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 *service* 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 can not do that. Kubernetes objects are not expected to be created at the same time or in a given order, so we can not 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, 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 string 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* to template their YAML-files. This is done so multiple configurations can inherit/share common code across different configurations. This means that references that are hardcoded in the files 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 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. (This, of course, only works if you have read access to those Kubernetes endpoints/resources from your local machine. If the endpoints are only visible from inside an enclosed environment, this will not work).

This paper will showcase a prototype of an extension made for Visual Studio that validates object references in Kubernetes Object Definitions continuously while the developer is typing. The tool aims to incorporate both plain local YAML-files, and output generated by tools like Helm or Kustomize, and scan what resources already exist in a running Kubernetes Cluster. 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. I will refer to the extension as *Kubernetes Reference Highlighter* (KRH).

Ten developers are given a task to debug a system. Half of them will do it with the extension enabled, the other half without the extension. The time it takes for each developer to debug the system will be tracked and evaluated.

The goal with 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. Linters like *ESLint* can be integrated with popular IDEs like VS Code or IntelliJ IDEA [1].

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 amount of errors by using Linters running in your IDE.

Similar results has 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 the tool if they loose trust in it, because of too many false positives. [4] Also not all "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.

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]

Same study shows that users want feedback from the tool as fast as possible and it should preferably be integrated with the IDE. [3]

6. Existing tools

In this section, existing tools are listed that aim to help the developer create correct/valid Kubernetes Object Definitions and tackle some of the challenges explained in the introduction-section.

The tools are collected by reading blogposts online and reading "alternative tools" on Github.

- **JetBrains IntelliJ IDEA Kubernetes Plugin** [[link](#)]: The full version 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 like KRH. But JetBrains' plugin comes with extra functionality that lets the user jump between files by clicking the highlighted references (something that KRH does not).
- **KubeLinter** [[docs](#)] [[GitHub](#)]: This is a 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. The full list of features can be found [[Here](#)]. KubeLinter is ideal to integrate into testing-pipelines where it can function as an initial step for everything YAML before it goes into production. KubeLinter can validate Helm Charts but can not validate Kustomize templates. The tool only checks the local files it is provided, so it does not know what already exists in a cluster.
- **kube-score** [[GitHub](#)] [[Validation Checks](#)]: 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 and not as a whole (with e.g., reference checking). Furthermore, just like *KubeLinter*, it is a Command Line tool and, therefore, can be integrated as part of a testing-pipeline and does not provide any live feedback when coding in an IDE.
- **Kubevious** [[docs](#)] [[GitHub](#)]: This tool comes with 44 built-in Kubernetes Resource validation checks. The checks include cross-file checks that, e.g., validate if a `service` points to two deployments at the same time and that kind of misconfiguration. Kubevious has a powerful toolset just like KubeLinter, but this tool has a UI 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](#)], **Copper** [[GitHub](#)], and **Config-lint** [[GitHub](#)] are all similar tools that 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 against. All these tools are Command Line tools, so they do not provide any live feedback but only validation when the

command is run. These tools accelerate when you want to build your own custom checks (maybe on your own Custom Resource Definition (CRDs)).

What is shared across all the tools listed above is that none of them manage to do reference checking across multiple files in a continuous manner that both takes local files, templating tools, and actual running clusters into account and at the same time is free to use.

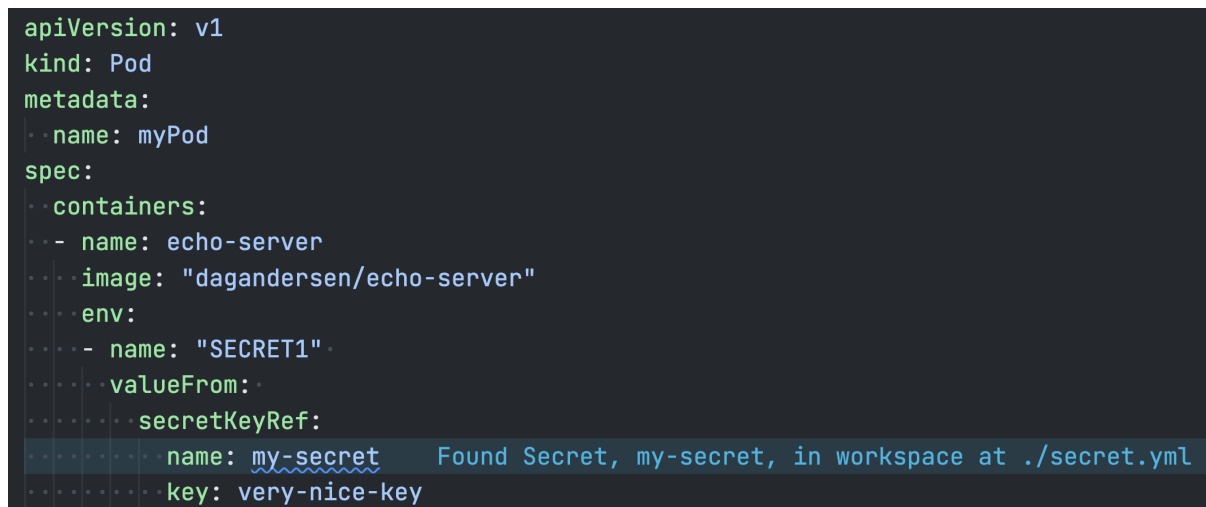
7. Prototype | Implementation

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

7.1. Overview

The tool is built as a Visual Studio Code (VS Code) extension since it is free and is the most popular IDE/Editor in the industry [\[source\]](#). The extension is made 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 finds a reference in the open file. 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
```

Fig. 1 shows how the extension would highlight the string `svc` in the VS Code because it found an object with that filename `./service.yml`. On this image, the VS Code extension, *Error Lens*, is enabled to better visualize the highlighting.

A tool like this is close to useless if the developer can not rely on the validity of the feedback it gives. This is why this tool aims to only 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 `vscode.diagnostics` for highlighting strings. Diagnostics are the squiggly line under code lines, and the severity can either be marked as an `Severity.Error` , `Severity.Warning` , `Severity.Information` , or `Severity.Hint` . This extension uses the `Severity.Information` for all object-references, and it only uses `Severity.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.

7.2. Reference Detection

As of Version 0.0.2, the extension highlights references to `Services` , `Deployments` , `Secrets` , and `ConfigMaps` . More kinds can easily be added in the future.

The extension use YAML-parsing 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 `Service` s 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 `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. The chance of detecting these references is more challenging, 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 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
                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
name: echo-server-service Found Service, echo-server-service, in 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).

7.3. Reference Collection

The tool is able to 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:** KRH uses the user's current context found in the `kube_config` to call the Kubernetes Cluster and collects the names of the Kubernetes objects that way.
- **Workspace Scanning:** KRH traverses all files ending with `.yaml` or `.yml` 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.yml` or `kustomization.yaml` in the currently open workspace in VS Code and builds the kustomize files. All the Kubernetes objects generated by `kustomize` is collected. In addition, the extension will also highlight if the Kustomization-file builds or not.

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

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

If *Kustomize* is not installed on the computer and part of the default shells PATH, it will notify the user.

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 CommandLine or through settings.

Kustomize is extra challenging to handle because it is built on the idea of "[bases](#)" and "[overlays](#)".

Overlays use a base-configuration and add/delete/modify/override 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.

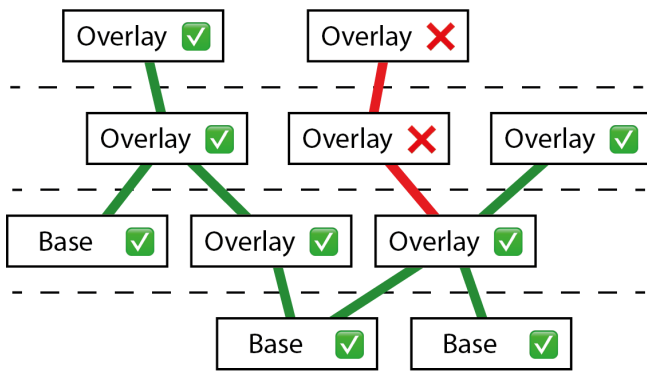


Figure X: Visualization of how the extension shows where overlays fail in a chain of overlays. The bases are at the bottom levels, while each layer above the bases are overlays.

8. Evaluation

The proper way of implementing a linter is to use some kind of abstraction syntax tree and pass tokens, but with the limited amount of time for this project Regex-matching acceptable is proof of concept. Using Regex has the risk of producing false positives or false negatives. If there is an edge-case my 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 amount of bugs/issues when working with Kubernetes, I have asked developers through slack-channels, Reddit, and other social medias/groups to try using the tool in their daily work for an month. Since the tool is publicly available in the VS Code Marketplace it is possible that people have also found it by chance. The participants was asked to fill in a survey. The survey was conducted in Google Forms and was available from the extension's marketplace-page.

When the extension have 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 have opened the survey or click "later" 3 times the message will not longer popup.

The questions and answers from the survey is listed here:

Question \ participant	person1	person2	person3	person4	person5	person6
Years of professional experience	0	0	0	0	0	0

Question \ participant	person1	person2	person3	person4	person5	person6
Years of professional experience with Kubernetes	0	0	0	0	0	0
role/job position	0	0	0	0	0	0
How likely are you to recommend this to others working with Kubernetes? (1-5)	0	0	0	0	0	0
Over the last month, did the extension help your daily work?	0	0	0	0	0	0
How many bugs did it help you catch	0	0	0	0	0	0
How often does the extension give false positives (highlighting a wrong reference)	0	0	0	0	0	0
Percentage estimate of how often it gives false negatives (not highlighting a reference that is actually there)	0	0	0	0	0	0
What type of scanning do you find most useful? (Cluster, plaintext/workspace, Kustomize)	0	0	0	0	0	0

9. Conclusion

Based on the fact X out of X persons reported they experienced the bug help 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 an success. This paper shows that even with 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 is needed to evaluate if an even better extension (with more complicated YAML parsing) would help developer catch even more bugs.

10. 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>