Kubernetes Resource Validation

Dag Bjerre Andersen, daga@itu.dk, IT University of Copenhagen, 15-12-2022, KIREPRO1PE

Abstract

Linters and static code analysis have proven to help developers catch bugs and speed up the development process in some domains. Kubernetes is growing in popularity, but the tooling for validating configuration files is limited. Most tools that exist only validate local manifests and do not handle Kustomize well. This results in incorrect and inadequate manifest validation. This paper presents a prototype of a Visual Studio Code Extension published in the VS Code Marketplace. The extension highlights object references found in local manifests, objects that exist in running clusters, and objects generated by Kustomize. The extension is tested by 61 people. After 15 days, 3 people filled out a survey with feedback. The main response was that reference-highlighting is useful and can prevent errors in certain situations. It proves that even simple highlighting can limit bugs and thus increase productivity.

The extension is named *Kubernetes Reference Highlighter* (KRH), and it can be installed from the *VS Code Marketplace*[1]. The source code can be found on *GitHub*[2].

Keywords

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

1. Introduction

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% [7], 30% [6]), and around one-fourth of the remaining developers want to work with Kubernetes in the future [7]. However, Kubernetes is known for being remarkably complex, and hence the learning curve can be quite steep. It is easy to make mistakes when deploying to Kubernetes because Kubernetes does not validate object references.
Also, most existing tools only validate local manifests and do not handle Kustomize well, which results in incorrect and inadequate manifest validation.

This paper presents a prototype of the *Kubernetes Reference Highlighter* (KRH) VS Code Extension[3]. The extension continuously validates object references in manifests while the developer is typing. The extension highlights object references in manifests based on local manifests, objects that exist in running clusters, and objects generated by Kustomize.

The goal of the extension is to provide the developer with valuable information that reduces the number of bugs encountered when deploying to Kubernetes and speeds up the debugging process. The goal of this paper is to demonstrate that simple reference highlighting can limit the number of bugs when working with Kubernetes.

It is assumed that the reader is familiar with Kubernetes concepts like *Services*, *Pods*, and *Ingress* objects and understands how they are defined in Kubernetes manifests. Furthermore, it is assumed that the reader knows the basics of how Kustomize works.

2. Problem described in detail

The problem is that it is easy to make mistakes when deploying to Kubernetes because Kubernetes does not validate object references when manifests are applied to a cluster. There are strict rules of what fields can be added in Kubernetes manifests, but there are no checks that the values provided are valid (besides the fact that it needs to be the correct type) - e.g., a *Pod* can request environment variables from a *Secret* that does not exist, or a *Services* can point to *Pods* that do not exist.

By design, Kubernetes does not do static validation checks (like reference validation) because Kubernetes objects are not expected to be created simultaneously or in a specific order. Kubernetes is managed with control planes, and there is no order to when each resource is created or scheduled. In Kubernetes, referenced objects will often not exist at creation time but will instead be created at a later point, which means there is no "compile time" where Kubernetes would be able to check all references. This explains why there is no static reference validation by default when creating Kubernetes resources.

A Kubernetes manifest is a YAML file that contains declared Kubernetes objects. YAML is a data serialization language that only consists of primitive data types like arrays, strings, numbers, etc., and therefore does not include the concept of references. All Kubernetes object references in manifests are written as a string, with no type checking or validation.

Since most IDEs, Kubernetes plugins, or tools do not verify any of the object references written as a string in Kubernetes manifests, debugging the code can be quite cumbersome. From my professional experience as a developer working with Kubernetes for more than a year, if a developer experiences a broken reference, it results in the developer reading through object references until the developer notices a typo or that of the referenced resources exists in two different namespaces and, therefore, cannot communicate. To the best of my knowledge, there is no public and free IDE feature or extension that checks this.

Furthermore, developers use tools like *Helm* and *Kustomize* for templating their manifests. This is done so multiple configurations can inherit/share common code across different configurations. This means that an object referenced in a manifest may not exist in any of the other local manifests 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 the names of the Kubernetes objects are dynamically generated by the tool. I have not found any free extension that tries to tackle this challenge, so currently, developers do not get any assistance validating their manifests live while coding when using templating tools like *Helm* and *Kustomize*.

Currently, plenty of open-source validation tools exist for Kubernetes manifests, but most of them are command-line tools and do not provide continuous feedback while the developer is coding. Furthermore, the tools only validate the manifests based on the local files provided to the tool and not what is actually running in a Kubernetes cluster. When the developer only has a subset of the complete infrastructure configuration on the local machine, many of the object references will naturally be broken locally, and thus the tools will often give incorrect and inadequate valuations. So, to overcome this, a tool needs to look outside the IDE/current folder and look at what objects exist in a running Kubernetes cluster.

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

3. Method

In this project, I build a prototype of the KRH, do a survey on users of my prototype, and evaluate the survey's results.

To evaluate if KRH can limit the number of bugs/issues when working with Kubernetes, I need users to test it. I have shared the extension in private slack-channels in my professional network (Devoteam G Cloud & Eficode), on Reddit[4], and in private messages to friends and colleagues. Since the tool is publicly available in the VS Code Marketplace, people may also have found it by chance.

When the extension has been installed for 15 days, a message asks users if they would be willing 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 5 days later. 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.

4. Related Work

4.1. Why developers use Linters

In dynamic languages like JavaScript, where there are no *compile time*-checks, studies have shown that developers use static analysis tools (*Linters*) like *ESLint* to prevent errors [9]. Linters like *ESLint* can be integrated with popular IDEs like VS Code or IntelliJ IDEA.

Preventing Errors is said to be the number one reason to use a linter because it catches the bugs early on, so developers do not have to spend time debugging it at runtime [9] [8]. A JavaScript developer reports that "*(...) you might have a typo in your variable name and because JavaScript is often not compiled, you'll only discover that much later when you run the code*." [8]. This is the same problem a developer can have in Kubernetes because object references are written as strings and are not checked. So even though a language like JavaScript does not have static compile time-checks, it is still possible to reduce the number of errors by using linters running in IDEs.

Similar results have been shown in studies related to Android programming. Android Developers report that they use linters to save time by detecting bugs and identifying unused resources [3]. Furthermore, Android Developers report, "*Developers always want to maintain a good reputation among their peers, and the linter can help them to do that*"[3], by ensuring that their work meets the expectations of their superiors and colleagues.

Developers report that linters and tools should give feedback as fast as possible and should preferably be integrated with the IDE [4]. The linter should provide the developer with continuous feedback on whether their code has errors or not, and it should not interrupt their workflows [4].

Other studies have shown that new developers can be sensitive to criticism, and it may be more comfortable for the new developer to be informed about coding mistakes from a linter instead of another developer [8].

4.2. Why developers do not use Linters

Studies have shown that some of the main reasons linters are underused are that they produce too many *false positives* and the number of warnings is too high, which overloads the developer [4].

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. (...) A stupid false positive implies the tool is stupid* " [1]. So, to get people to trust and use a tool, it is important that users do not experience false positives on simple issues. False positives are less important on more complicated issues [1].

Furthermore, the user will interpret correct feedback as a false positive if they do not understand the feedback. "*A misunderstood explanation means the error is ignored or, worse, transmuted into a false positive*" [1]. "*If people don't understand an error, they label it false*" [1]. So, the tool must provide easy-to-understand feedback if the user should trust the linting tool.

Another reason developers do not use linters is that they can be difficult to configure [4].

5. Existing tools

The tools listed below aim to help the developer write correct and valid manifests and tackle some of the challenges explained in this paper's section: *2. Problem described in detail*. The tools are collected by reading blog posts[5] online and searching on GitHub for "Kubernetes linter".

- **KubeLinter** *[GitHub[6]][Validation Checks[7]]*:
  KubeLinter is a feature-rich open-source command-line tool that validates Kubernetes manifests. KubeLinter can validate Helm Charts, but it cannot validate Kustomize templates. The tool only checks the local manifest, so it does not know what Kubernetes objects already exist in a cluster. KubeLinter detects "dangling references" (broken references), so it informs the user if a reference does not exist. That is good, but when the users do not have the whole infrastructure configuration locally, 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 actual issue), which can result in the developer not trusting the tool and thus disabling the tool as described in section: *4.2. Why developers do not use Linters*.

- **kube-score** *[Github[8]][Validation Checks[9]]*:
  kube-score is an open-source command-line tool that performs static code analysis of local manifests. The tools have over 30 different checks, but it only validates the manifests individually, not the relation between them. Thus, it does not solve the problem of detecting broken references. Furthermore, just like KubeLinter, it is a command-line tool and, therefore, can be integrated as part of an automated pipeline but does not provide any live feedback when coding in an IDE.

- **Kubevious** *[GitHub[10]][Docs[11]]*:
  This tool comes with 44 built-in Kubernetes object validation checks. The checks include cross-file issues, like if a *Service* points to two different *Deployments* simultaneously. Kubevious has a powerful toolset just like KubeLinter, but this tool instead has a GUI and needs to be run in a container inside the cluster or as a stand-alone application. 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 useful for finding and debugging issues after deployment.

[4] https://www.reddit.com/r/kubernetes/comments/yoy0w8/kubernetes_yaml_linter_for_vscode/
[5] https://kubevious.io/blog/post/top-kubernetes-yaml-validation-tools
[6] https://github.com/stackrox/kube-linter
[7] https://docs.kubelinter.io/#/generated/checks
[8] https://github.com/zegl/kube-score
[9] https://github.com/zegl/kube-score/blob/master/README_CHECKS.md
[10] https://github.com/kubevious/kubevious
[11] https://kubevious.io/docs/built-in-validators

- **Conftest** *[GitHub[12]]*, **Copper** *[GitHub[13]]*, and **Config-lint** *[GitHub[14]]*:
  These are three similar tools built to validate Kubernetes manifests, but none of them come with built-in checks, so the users must make their own validation "rules" for their manifests. These tools are best suited when users want to write rules for their own custom resources (CRDs). All three are command-line tools, so they do not provide any live feedback in an IDE.

- **JetBrains IntelliJ IDEA Kubernetes Plugin** *[website[15]]*:
  This plugin is only available through the full version of IntelliJ IDEA. I will not look into this plugin since its documentation is sparse, and IntelliJ IDEA is a paid product and hence is not widely accessible.

None of the tools listed above supports reference checking in local manifests, templating tools, and actual running clusters and, at the same time, is free to use and provides live feedback from within an IDE.

6. Prototype

This section describes the prototype of the *Kubernetes Reference Highlighter* (KRH) VS Code Extension[16].

6.1 Overview

KRH 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 [7]. KRH is written in Typescript.

KRH 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 a user types and gives continuous feedback on whether references are found or not. An example of KRH highlighting a reference can be seen in Figure 1.

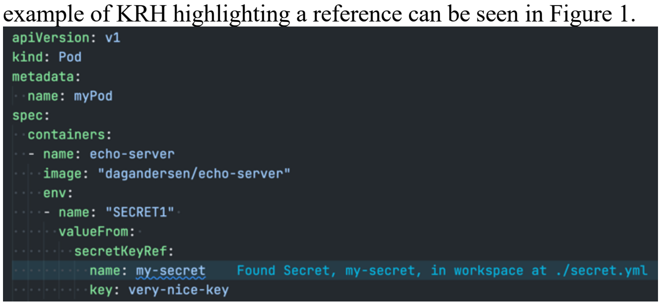

Figure 1: A Kubernetes manifest, where the reference to a secret named `my-secret` is highlighted by the KRH extension. The highlighting shows that the secret `my-secret` exists in the local VS Code workspace.

If developers cannot rely on the validity of the feedback a tool like this provides, they will stop using it [1]. Therefore, KRH aims only to highlight a string if it is sure that the reference exists. It is better to have *false negatives* than *false positives* so the developer does not get overloaded [4].

KRH makes use of *VS Code Diagnostics* for highlighting strings. Diagnostics are shown as the squiggly lines under code lines. 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 KRH integrate well with other VS Code features/extensions. In particular, I suggest installing *Error Lens*[17] together with KRH to visualize the diagnostics even better for the user. The manifests shown in Figures 1, 2, 3, and 4 all have Error Lens installed and enabled, which prints the diagnostic information as a blue line next to the diagnostics (squiggly lines).

[12] https://github.com/open-policy-agent/conftest
[13] https://github.com/cloud66-oss/copper
[14] https://github.com/stelligent/config-lint
[15] https://plugins.jetbrains.com/plugin/10485-kubernetes

The extension's architectural structure is divided into two components: *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 a YAML-file is saved 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.

6.2 Reference Detection

KRH highlights references to *Services*, *Deployments*, *Secrets*, and *ConfigMaps*.

KRH uses YAML-parsing libraries[18] and regular expressions as the primary method for reading the files in the workspace.

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

Raw hostname addresses pointing to *Services* are a bit different and the most difficult references to handle. A *Service* can either be called using its name as an address if referenced from inside its namespace (`my-service`) or be called from anywhere within the cluster by suffixing the *Service*-name with its namespace (`my-service.its-namespace`). KRH highlights only an address pointing to a service if it exists inside the same namespace, or if the address is suffixed with its namespace. This is an important distinction that can prevent issues related to traffic routing, e.g., a *Pod* not being able to reach a *Service* because it is placed in a different namespace.
Raw addresses are generally 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 and false negatives.

If the string matches with multiple objects, KRH will show all the references. A reference can, as an example, exist both in the running cluster and in the local file simultaneously because the object is already deployed to the cluster. An example of this can be seen in Figure 2.

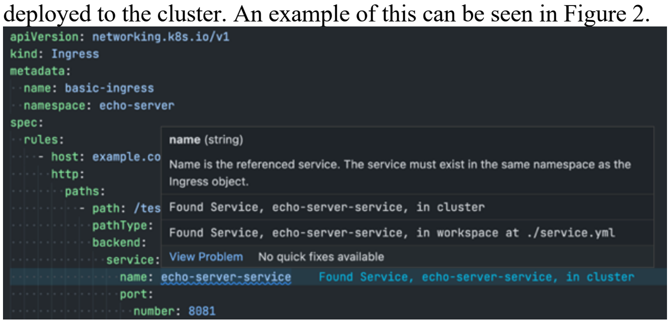

Figure 2: A Kubernetes manifest, where there are multiple reference matches on `echo-server-service`

6.3 Reference Collection

KRH has three scanning techniques for collecting references. KRH can read manifests in the local workspace (Workspace Scanning), read generated output by Kustomize (Kustomize Scanning), and fetch objects from a running cluster (Cluster Scanning).

- **Cluster Scanning**:
  KRH uses the current context found in the user's kubeconfig to call the Kubernetes cluster and collects the *kind*, *name*, and *namespace* of its Kubernetes objects.

[16] https://marketplace.visualstudio.com/items?itemName=dag-andersen.kubernetes-reference-highlighter
[17] https://marketplace.visualstudio.com/items?itemName=usernamehw.errorlens
[18] https://www.npmjs.com/package/yaml

- **Workspace Scanning**:
  KRH finds 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 finds 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 Kustomize-file builds or not (as seen in Figures 3 and 4).

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

Figure 3: A Kustomize-file, where Kustomize builds unsuccessfully

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

Figure 4: A Kustomize-file, where Kustomize builds successfully

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

Kustomize is challenging to support because it is built on the idea of *bases*[19] and *overlays*[20]. 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 arbitrarily, 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 tree of overlays depending on each other is beyond the scope of this project. Instead, the tool informs the user of references it finds in all the kustomization layers, and then it is up to the developer to check which layer they meant to refer to. The fact that KRH highlights which Kustomize-files build or do not build can be helpful in debugging a long chain of Kustomize layers.

7. Evaluation

To evaluate if the prototype KRH can limit the number of bugs/issues when working with Kubernetes, I have conducted a survey.

As of December 14th, 2022, the Reddit post[21] has shown up in 13.800 people's feeds, gotten 16 upvotes and a few comments. 61 people have installed KRH, and 3 people (respondents) have filled out the survey. Hence the response rate is 4,9%.

7.1 Results

The survey contains 10 mandatory questions and two optional questions with free-text answers. Only one person answered the optional questions. The full survey can be seen in Table 1.

Here are the results summarized:
- All three respondents said KRH helped them in their daily work and were likely to recommend it to others.
- All three respondents had at least 1 year of experience working with Kubernetes.
- All three respondents reported a relatively low number of false negatives.
- Two out of three respondents experienced a low false positive rate, while the third experienced a significantly higher rate of false positives.
- Two respondents reported that KRH helped them catch between 1-5 bugs, while one reported that KRH helped them catch between 5-10 bugs during the testing period.
- Each respondent found a different scanning technique to be most useful.
- One respondent requested Helm Support.

***Table 1:* Survey Results**

Mandatory questions \ participant	**Person1**	**Person2**	**Person3**
Years of professional experience	3	1	5
Years of professional experience with Kubernetes	1.5	1	3
Role/job position	Consultant	Cloud Engineer	DevOps Consultant
How likely are you to recommend this extension to others working with VS Code and Kubernetes? Scales go from 1:"very unlikely" to 5:"very likely".	5	4	4
Over the last month, did the extension help your daily work?	Yes	Yes	Yes
How many bugs did it help you catch?	1-5	5-10	1-5
How often does the extension give false positives (highlighting a wrong reference)	0 out of 10 times	1 out of 10 times	5 out of 10 times
How often does the extension give false negatives (not highlighting a reference that should be highlighted)	0 out of 10 times	3 out of 10 times	2 out of 10 times
Do you use Kustomize?	No	Yes	Yes
What type of scanning do you find most useful? (Cluster Scanning, Workspace Scanning, Kustomize Scanning, None)	Cluster Scanning	Workspace Scanning	Kustomize Scanning
Optional questions \ participant	**Person3**		
What feature would you like implemented?	*Helm support. It's quite rare that I am actually editing 'raw' manifests, they are almost always wrapped in some amount of helm.*		
Do you have other comments?	*I think the idea of having in-editor checking of references is awesome! And I really think that we need better tooling to handle working with k8s manifests and yaml in general.*		

[19] https://kubectl.docs.kubernetes.io/references/kustomize/glossary/#base
[20] https://kubectl.docs.kubernetes.io/references/kustomize/glossary/#overlay
[21] https://www.reddit.com/r/kubernetes/comments/yoy0w8/kubernetes_yaml_linter_for_vscode/

7.2 Discussion

All three respondents said KRH helped them in their daily work and were likely to recommend it to others. I interpret this as KRH being valuable and helping the user significantly enough to recommend it to others. As Person3 states: "*I really think that we need better tooling to handle working with k8s manifests and yaml in general*".

Since Kubernetes does not validate object references, a tool like KRH can help developers in their daily work by highlighting object references found in local manifests, objects that exist in running clusters, and objects generated by Kustomize.

Since all respondents mentioned that it helped them catch at least 1 bug (two answered 1-5, one answered 5-10), and the goal of the tool was to reduce the number of bugs, I would consider KRH a success. However, I don't know how much each respondent used the extension, so it is difficult to conclude if the reduced number of bugs is to be perceived as relatively high or low.

Yet, this indicates that even simple highlighting can limit bugs, speed up the debugging process, and thus increase the productivity of developers.

So just like shown with JavaScript developers and Android developers (described in section: *4.1. Why developers use Linters*), a linter like KRH helps developers working with Kubernetes detect bugs by providing continuous feedback from inside an IDE.

All three scanning techniques seem to be valuable for the users since each respondent thought a different scanning technique was most useful. It shows that all three features (Cluster Scanning, Workspace Scanning, and Kustomize Scanning) are needed and are helpful for developers in their daily work.

A high number of false negatives means the tool fails to highlight many of the references, meaning that it doesn't help the developer significantly. However, since all three respondents reported a low number of false negatives, it means KRH finds most object references correctly and therefore helps the developer identify correctly defined references.

When developing KRH, I tried to be very cautious of not producing false positives because a high number of false positives results in developers getting overloaded and makes them disable the tool [4]. Person3 reported significantly more false positives than the other two. From this survey alone, it is not clear why and what kind of false positives Person3 was getting. Since Person3 reported that the most useful feature is *Kustomize Scanning* and that "*It's quite rare that I am actually editing 'raw' manifests*", my guess would be that Person3 mainly used the extension in relation to Kustomize. As stated in section: *6.3 Reference Collection*, Kustomize is more challenging to handle, so it is not surprising that my prototype produces more false positives or false negatives working with those files. Even though Person3 was the respondent who experienced the most false positives, the person still found the extension helpful in the daily work and was likely to recommend it to others.

Using regular expressions has the risk of producing false positives or false negatives. A better way of implementing a linter may be to use some kind of abstract syntax tree (AST)[22], but with the limited amount of time for this project, matching on regular expression is acceptable for a prototype. I would expect that using an AST would produce fewer false positives and false negatives by catching more edge cases than KRH.

Person3 also requests Helm support. This would be the natural next feature to implement since Kustomize and Helm are both popular templating tools for Kubernetes manifests. Whether or not such a feature is feasible (especially when created with regular expressions) is a question for further research.

[22] https://python.plainenglish.io/asts-and-making-a-python-linter-from-scratch-79e66e8d99b8

8. Conclusion and future work

This paper presents a prototype (KRH) of a Visual Studio Code Extension that highlights object references found in local manifests, objects that exist in running clusters, and objects generated by Kustomize. KRH tested by 61 people, and 3 people filled out a survey with feedback.

Since all three survey respondents reported that KRH helped them catch bugs, helped them in their daily work, and were likely to recommend it to others, I would consider KRH a success.

This paper shows that even with a simple extension based on regular expression, it is possible to support developers with valuable live validation of Kubernetes manifests to limit bugs and thus increase productivity.

Further research and development are needed to evaluate if it is feasible to extend KRH with Helm support and if parsing manifests using ASTs will lower the false positive rate.

9. References

1. 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
2. S. Habchi, X. Blanc and 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
3. 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
4. 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
5. Sanket, "The exponential cost of fixing bugs", DeepSource, January 29 2019, https://deepsource.io/blog/exponential-cost-of-fixing-bugs/
6. SlashData's Developer Economics survey, "The state of cloud native development", March 2022, Section: 2.A., https://developer-economics.cdn.prismic.io/developer-economics/527f60d6-d199-4db8-bd31-6dde43719033_The+State+of+Cloud+Native+Development+March+2022.pdf
7. StackOverflow, "StackOverflow Survey 2022", June 2022, Section: "Most popular technologies: Other tools", https://survey.stackoverflow.co/2022/
8. K. F. Tómasdóttir, M. Aniche and A. Van Deursen, "The Adoption of JavaScript Linters in Practice: A Case Study on ESLint," in IEEE Transactions on Software Engineering, vol. 46, no. 8, pp. 863-891, 1 Aug. 2020, doi: 10.1109/TSE.2018.2871058. https://pure.tudelft.nl/ws/files/46810031/tse_js.pdf
9. 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