Prometheus Operator - Extending The ScrapeConfig Surface

[Google Doc Link]

M Viswanath Sai

Table of Contents

About Me

Name	M Viswanath Sai
Date of Birth	21/08/2003
Country	India
Email ID	mviswanath.sai.met21@itbhu.ac.in
Github ID	mviswanathsai
Slack ID	@mviswanathsai
University	Indian Institute of Technology (BHU), Varanasi
Contact Number	+91 8778912753
Timezone	UTC+05:30 (Asia/Kolkata)
Preferred Language	English

Past Experience

- Part time at DoDAO: Worked on Technical writing on Web3 technologies, Frontend for the websites and Business logic for automated News generation.
- Multiple Frontend applications.
- XLang: Implementation of a compiler written in Golang based on the book by Thorsten Ball.
- Contributor at Kyverno.

Past Contributions

Since my time of joining the Prometheus-Operator community on Feb 29, 2024, I have made a few contributions to both the code base and the coding community.

Raised Issues:

- #6382: Update documentation with relevant links about ScrapeConfig
- #6461: Extend support to missing fields in AlertmanagerEndpoints

PRs:

- #6386: Add support for DockerSD in ScrapeConfig
- #6388: Update CONTRIBUTING.md
- #6390: Bug fix for ScrapeConfig selection across namespace
- #6451: Bug fix for empty replacement not possible
- #6450: Add support for `alert_relabel_configs` per AlertmanagerEndpoint

Discussions:

- Discussion regarding the Daemonset model.
- Code investigation for ScrapeConfig bug.
- Bunch of specialized discussions regarding the Daemonset mode and ScrapeConfig in separate groups with the maintainers.
- Other smaller discussions throughout my time in the Slack channel.

Abstract

The ScrapeConfig CRD was launched to bear a 1:1 relationship with the actual prometheus configuration (job) that will be generated by the Operator. It is currently an alpha CRD and the reason is that it does not yet resemble this 1:1 relationship. The API needs to be updated in order to graduate it to a stable state. This document lays out a potential roadmap that answers two questions:

1. What changes/updates need to be made in order to graduate the CRD.
2. How do we graduate it to stable?

Additionally, on making the CRD stable, it might be of our interest to ultimately generate ScrapeConfigs for every Prometheus CR. This is outlined in the original design document for ScrapeConfig CRD.

Motivation

When looking at the projects put up for GSoC from Prometheus Operator, this project looked like a good fit for my skill set. Hence, I started working on it. Going through code, I wanted to get a feel for myself about how a part of the problem could be tackled. As per Jayapriya Pai's suggestion, I started on adding DockerSD configuration support for the ScrapeConfig CRD.
I quickly realized that it was straightforward to code, the majority of the time was to be spent on writing tests and making sure everything works as expected. I have had many

discussions with both Simon and Jayapriya over the past few weeks and I think working with them would be equally smooth and educational.

Project Plan

Pitfalls of The Current Model

To understand where the need for a generic ScrapeConfig CRD comes from, it might be of interest for us to start from a valid use case.

Scraping Out of The Cluster
Within the cluster we're good with the existing Service-, PodMonitor + Probe. But for things outside the cluster, the users currently only have the additionalScrapeConfigs as a stable CRD where the users have to modify the additionalScrapeConfigs-secret and deploy the Prometheus again.
Compared to using the *Monitor custom resources which can be deployed independently alongside applications and for which the operator generates the scrape config in the relevant Prometheus. Another issue is that only one additionalScrapeConfigs is permitted for the user, this again, limits our abilities.
For this reason, it might be useful for us to support a stable, generic ScrapeConfig resource that is both dynamically generated and updated. The idea is that the ScrapeConfig is the resource that actually generates the final config and the *Monitors, Probes, etc generate a ScrapeConfig. This is also currently not done since we would not want to couple stable resources with an alpha resource.

Currently, the ScrapeConfig CRD does not yet support all the functionalities that the actual Prometheus configuration offers. For example, the extensive support for different service discoveries is a missing part of the ScrapeConfig CRD.
Since the development of the ScrapeConfig is not yet complete, it is in the alpha stages. This is a pain point for users who would want to scrape outside the cluster but cannot use an unstable CRD.

Goals

- Point out what fields need to be added to the CRD
- Suggest a graduation strategy for the ScrapeConfig CRD
- Kick-off discussions related to the ScrapeConfig CRD

Non-Goals

- Not meant to act as a strict implementation roadmap

How

The problem statement can be handled at 2 different levels:

1. What changes to the API need to be made, to make the CRD stable?
2. What would the graduation strategy look like?

Changes To The API

Adding Service Discoveries

We can start easy and look for the Service Discoveries that need to be supported. The Prometheus config boasts of support for 25+ service discoveries. Though it might not be necessary to support all of them, it would be valuable for us to extend support to at least 15 of the most common service discoveries used. From there on, we could extend support as users raise issues.
This is the most straightforward part of the solution, the service discoveries are rather similar and to integrate each new one can be done by taking inspiration from the implementation of the existing service discoveries.

Extending Support To Newer Fields

Our objective would be to extend support to all, if not almost all of the most-used fields before we can graduate ScrapeConfig to stable. This will include adding these fields to the scrape config struct with a valid data structure.

Adding Tests and Validation

For the fields added, we also need to write tests in `resource_selector_test.go` and `prom_cfg_test.go` where we will check the string output of processing the scrape config against .golden files. This is necessary to ensure that the configuration output is the same as we expect it.
Considering our scenario, it seems like a good option to do test driven development.

With the above three factors we might be able to tangibly determine how close we are to a stable ScrapeConfig CRD. The more boxes we tick, the better.

Graduation Strategy To Stability

To Beta

It would be wise for us to take inspiration from the Kubernetes API versioning architecture which suggests a way to tangibly rank our CRDs into alpha, beta and stable versions.
The Kubernetes API versioning architecture states the following about a beta API:

> *Completeness: all API operations, CLI commands, and UI support should be implemented; end-to-end tests complete; the API has had a thorough API review and is thought to be complete, though use during beta may frequently turn up API issues not thought of during review*

We would like to ensure that we have covered all of the fields that we have in mind, reviewed said API and also thoroughly tested the changes/additions. This need not be perfect however, in beta, we still have headroom for users to let us know of issues that we did not yet think of or write a test case for.
As far as upgradability goes,

> *Upgradeability: the object schema and semantics may change in a later software release; when this happens, an upgrade path will be documented; in some cases, objects will be automatically converted to the new version; in other cases, a manual upgrade may be necessary; a manual upgrade may require downtime for anything relying on the new feature, and may require manual conversion of objects to the new version; when manual conversion is necessary, the project will provide documentation on the process*

In case we wish to update some of the fields after making it beta, we are still allowed to. Provided that we document the change as a function of version (v1beta1, v1beta2,etc.,.). This also means that we lay out any steps necessary to be done by the end user to adapt to the changes.

It is also important to note that there might be minor bugs with our ScrapeConfig CRD, but we have to make sure that no new bugs are introduced to the existing architecture. This could be ensured by the existing test suites and extending said test suites as we seem fit with the growth of the ScrapeConfig development.

> *Support: the project commits to complete the feature, in some form, in a subsequent Stable version; typically this will happen within 3 months, but*

> *sometimes longer; releases should simultaneously support two consecutive versions (e.g. v1beta1 and v1beta2; or v1beta2 and v1) for at least one minor release cycle (typically 3 months) so that users have enough time to upgrade and migrate objects*

With the above rules of thumb in mind, we can draw some conclusions:
Beta might be the best stage for us to consolidate all Monitor resources into a final ScrapeConfig resource, as outlined in the original design document. The reason is multi faceted,

1. In Alpha, we would not want to consolidate the Monitor resources (stable) into a ScrapeConfig (unstable).
2. In Stable, once we have graduated ScrapeConfig to stable, we might not have the same flexibility to deal with the consolidation logic. It does not make sense to make the API stable yet have unnoticed bugs crawl into the architecture due a new feature related to the said API.
3. In Beta, we have the flexibility to make mistakes and the API itself is in a stage where most of the features we want are complete. This makes it a good choice for us to experiment with the consolidation feature.
4. However, it might be worth it to make it an optional feature by passing it as a flag to the container. Something like `--consolidate-monitors`, making it a boolean flag which enables the consolidation feature seems reasonable.

How do we change the API in the beta version?
Similar to Alpha, when newer fields need to be added could even be because of an addition to the actual scrape configuration, we might need to change the API again. In such a case, we would move with (v1)beta1, (v1)beta2, etc. For example, in the issue #4656, when `AlertManager` CRD was in beta, the actual alert manager configuration was changed and a subsequent change was made to the `AlertManager` CRD in the v1beta1 version.

To Stable

The Kubernetes API versioning architecture suggests the following about a Stable API version:

> *Availability: in official Kubernetes releases, and enabled by default*
> *Audience: all users*
> *Completeness: must have conformance tests, approved by SIG Architecture, in the appropriate conformance profile (e.g., non-portable and/or optional features may not be in the default profile)*

> *Upgradeability: only strictly compatible changes allowed in subsequent software releases*

The suggestions are reasonable and consequential to thorough testing and multiple API reviews, both from fellow developers and users. A sufficiently long period in beta stages should be enough to sniff out the troublesome bugs, leaving the very edge case, minor bugs behind.

If we need to make changes to the API after it has been made stable, we can feature-gate the newly added fields and continue development. More information related to changing the API can be found here.

Handling API Graduation

When moving from one version to another, for example from alpha to beta, we would like to give the users some headroom to migrate their stack from the old version. This will usually happen over a few releases. During this time, both the versions of the CRD are supported. This can be achieved readily if there is no difference between the two versions, i.e they are virtually identical. In case there are differences, which is usually the case, we would then like to enable conversion between the versions.

This is implemented by means of a webhook similar to validation webhook that is already in place for the AlertManager CRD, we also have a conversion webhook which is used to convert to and from the beta and alpha versions of the resource. We already have the webhook server in place, handling the conversion should not be challenging.

Timeline of Graduation

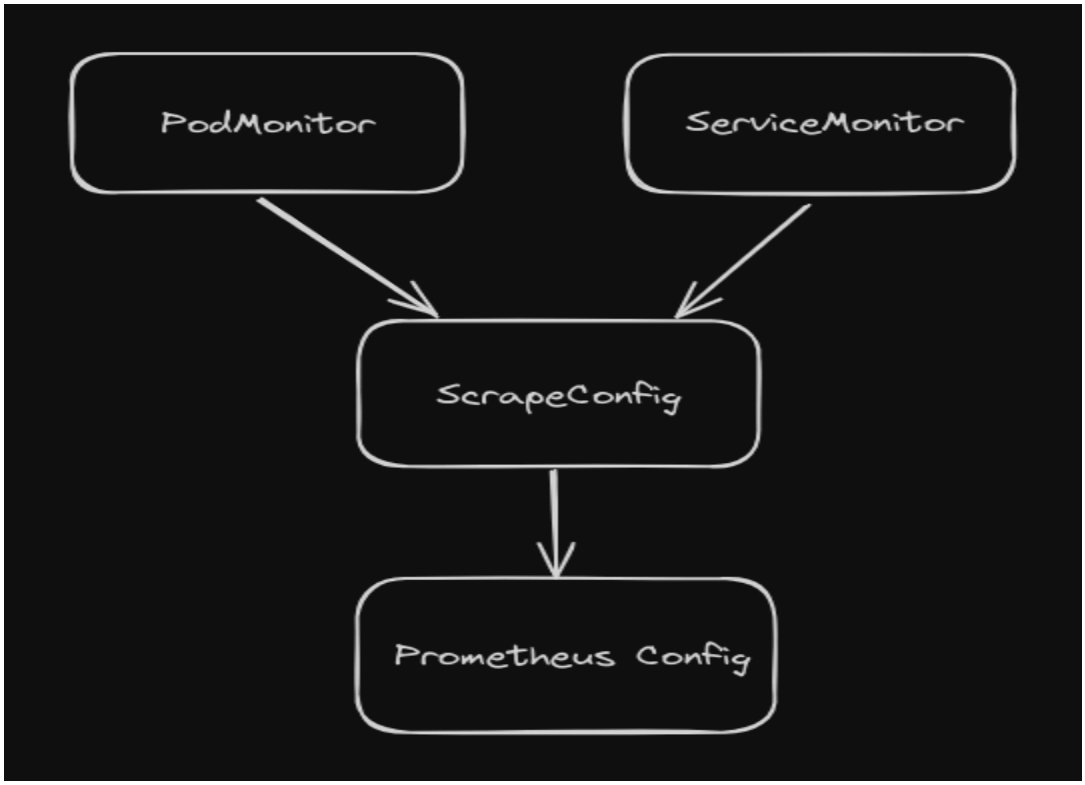
We can follow the guidelines in place by Kubernetes and follow a roughly similar path to the AlertManager CRD:

- [v0.72.0] -> [v0.73.0] (~3 months): ScrapeConfig CRD v1beta1 is opt-in (e.g. not included in bundle.yaml).
- [v0.74.0] -> [v0.77.0] (~6 months): ScrapeConfig CRD v1beta1 is deployed by default, v1alpha1 marked as deprecated but still defined as the stored version and used by the operator.
- [v0.78.0]: ScrapeConfig v1alpha1 is removed, the operator switches using v1beta1.

Additionally, the following needs to be communicated clearly to the users:

- The deprecation plan, and an agreed upon timeline similar to the example above.
- Documentation on migration for users to migrate their objects to the new version.

ScrapeConfig Generation



As described in the original design doc for ScrapeConfig CRD, it might be our motive to finally generate a ScrapeConfig for each Prometheus/PrometheusAgent resource from the underlying Monitor resources. But to achieve this, we ought to have a stable version of the ScrapeConfig running. This has also been discussed in an ad-hoc office hour meeting, the notes for which are here.
On graduating the resource to beta or stable, we might want to proceed with development in this direction. This might continue past the 12 weeks of the program, or within the 12 weeks, depending on when we deem ScrapeConfig to be stable enough. Regardless, my participation will be consistent.

Project Timeline

Weeks 1-2: Project Planning and Requirement Analysis
Identify missing features and configuration options.
Align on success criteria and expectations from the project.
Align upon a potential timeline for the project: Includes a graduation timeline.

Weeks 3-4: Add New Fields
Start writing tests that expect any new fields we might want to add: likely in `promcfg_test.go`
Start development to pass the failing tests.

Weeks 5-7: Add New Service Discoveries
Write tests for the missing SDs we want to cover: likely in `promcfg_test.go`
Start development to pass the failing tests.
Ideally, we might want to do one SD at a time.

Weeks 7-8: e2e Testing
Apart from the tests written alongside development which are closer to unit-tests than e2e tests, we would like to ensure that there are no breaking bugs.
Writing new e2e tests to cover as many scenarios as possible and letting users give feedback might be a good way to go: likely in `scrapeconfig_test.go` and referencing it in `main_test.go`.
Extending test coverage would be integral.

Weeks 8-10: Graduation
Graduate the API to beta.
If the API has undergone changes, we would need to create a conversion webhook like we did for the AlertManager CRD.
At this point, the beta version is opt-in and storage of the CRD would likely be in beta too.

Weeks 11-12: Documentation
Enhance the documentation to include up-to-date information on the CRD.
Create migration guides for users upgrading if necessary.

Additional Information

Working on Prometheus-Operator so far has been a joy, the maintainers relentlessly answer our questions (no matter how naive) and indulge in meaningful and educational discussions about multiple aspects of the project. I have some experience in working with similar projects to Prometheus-Operator and working on the project so far has helped me extend my understanding further.
Prometheus-Operator has so far both included me as a part of the community and has improved my knowledge in the domain as well. This makes it fun to contribute and “work” with the community and I hope to continue doing it!

During the GSoC period, I will be on my summer vacation. I will be available to work normal hours for the most part, with the possible exception of a few days long trip with the family. Though, I expect that I will be working on the project for the most part. It really does keep me engaged.

After the GSoC period, If I am allowed to be as ambitious, I would like to contribute and become wmaintainer at some point. This does not only come from my liking of the project itself but also from the community of maintainers who have been immensely friendly and helpful thus far. I am unaware of what qualifications are expected from a maintainer, but I am positive that I will build myself up to however high that is and eventually become an integral part of the community even if not as a maintainer!