What's the best way to manage Helm charts?

Merlin Carter
May 29, 2020 · 9 min read ★

Background photo by Joseph Barrientos on Unsplash

Whether you love it or hate it, Helm is a ubiquitous tool for managing Kubernetes applications. You can use it in many different ways which is great, but can also be overwhelming.

We've noticed that a couple of questions keep popping up:

- **Where do you put your Helm charts?**
  Do you keep them with the app files or use a chart repository?
- **How do you divide up your Helm charts?**
  Do you use one shared chart or maintain a chart for each service?

I'm going to try and address these questions based on our experience with the startups in our portfolio but I also draw on the perspectives of larger organizations too.

Here are the options that I'll be outlining:

1. Using a **chart repository** to store one big **shared chart**
2. Using a **chart repository** to store many **service-specific** charts.
3. Using **service-specific charts** which are stored in the same repository as the **service itself** (spoiler alert: we prefer this one).

I'll then cover factors such as dependency variance and team structure that you should consider when deciding on one of these options.

Option 1: Maintain one big shared chart in a chart repository

In one of our projects, we started out with a large chart that was used to deploy multiple services. It was stored in ChartMuseum and maintained by the person responsible for the deployment infrastructure.

A shared chart can save a lot of hassle if your services are very similar in nature. For example, take this scenario that Helm maintainer Josh Dolitsky described at KubeCon 2019:

> *I was recently working on a project and there [were] nine microservices … at about number two or three, I realized like they were all pretty much the same HTTP listening service. So instead I .. decided let's build a single helm chart that works exactly the same for every service and so we use a single helm chart to deploy nine different services doing a different configuration for each — just setting a new docker tag for that specific service.*

In this case, it makes sense to store the Helm chart in a chart repository such as ChartMuseum. Only the values would need to be kept in these service-specific repositories.

Option 2: Maintain several service-specific charts in a chart repository

Service-specific charts have the advantage that you can make a change to one service without worrying about breaking something for another service. But they can cause duplicated work. If you're updating a common configuration, you'll have to make the same change in each of your charts.

Whether you need to keep them in a chart repository is another question. If the charts are service-specific anyway, then there's no strong architectural argument for storing them together. However, often this is easier if you have a dedicated person or team maintaining all the charts.

For example, I worked with one DevOps engineer who maintained charts for 15 different microservices in a central chart repository. It was easier for him to update them all in one place rather than submitting pull requests to 15 different repositories. The developers themselves knew how to update the charts and sometimes did so, but he preferred to handle the resource-related settings such as pod disruption budgets.

Option 3: Maintain service-specific charts in the same repositories as the services themselves

Service-specific charts are a good choice for microservice-based applications where the services have significant differences. This pattern works better when you keep each chart in the same repository as the service code.

If you store the Helm chart in the service repository, it’s easier to continuously deploy the service independently from other projects. And you can commit your chart updates (such as adding a new variable) together with your changes to the application logic — making it simpler to identify and revert breaking changes.

However, the advantages of this option depend on how many microservices you maintain. If you‘re getting into double digits (like the scenario I described in the previous section) this option might be more of a hindrance than a help. This is especially true if you’re dealing with very homogenous services like in Josh Dolitsky’s example.

What to take into account when deciding on an option

There are generally two general dimensions to take into account:

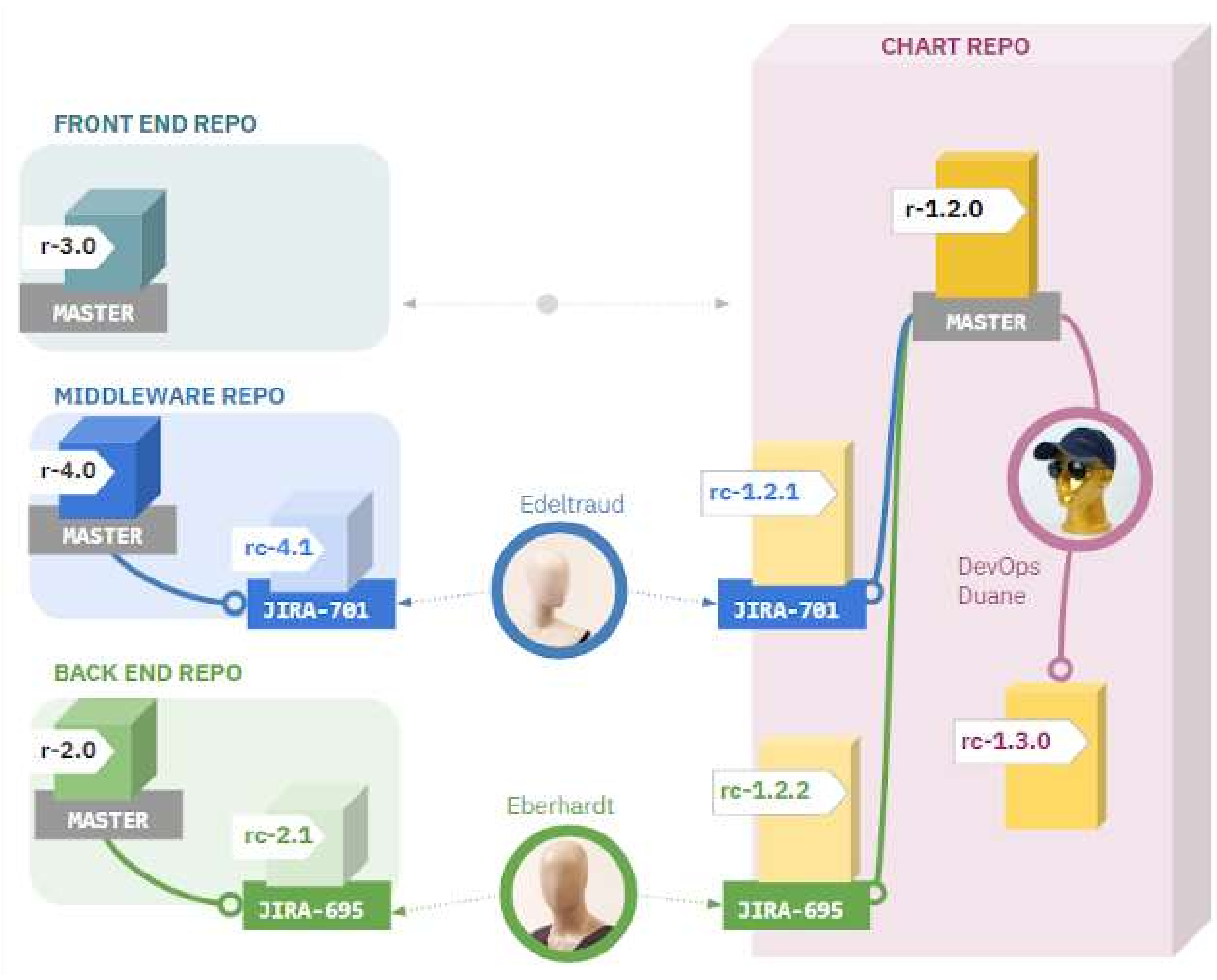
- **Dependencies and reproducibility:** How different are the dependencies for each service? What is the risk that a change to one service will break another? How do you reproduce the conditions of a specific development?
- **Team structure**: do you have small autonomous teams that are responsible for each service? Do you have developers who have some knowledge of DevOps? How prevalent is “DevOps Culture” in your team?

Dependencies and Reproducibility

If you maintain your charts separately from your application, they will be versioned distinctly from one another. If you have a problem with a deployment and need to reproduce the conditions that caused it, you will need to identify: a) the service version, and b) chart version used to deploy it. One might be tempted to take a shortcut and always test service x.x.x with chart "latest", but this is a terrible idea. You will never reproduce the exact conditions that caused an issue.

So what if you often have releases that require a change to the chart? Shouldn't you test those changes together?

Consider this illustrated scenario where many developers need to create branched versions of the same shared chart:



Each **service** branch needs to use the correct **chart** branch

- The developers (Edeltraud and Eberhardt) are both working on different feature branches and want to test their changes in a dev environment along with the chart changes — so they need to branch the charts too.

- In the meantime, the DevOps engineer, Duane, is updating some common components in his own branch of the shared chart.

If no one branches their chart changes, they risk breaking the deployment of another service.

We encountered exactly this problem not so long ago. The chart maintainer updated the shared chart with a new conditional block. The statement checked to see if a new variable "foo" was set to "enabled". However, the variable "foo" wasn't yet defined in the values files for all services. The deployment broke for the services that were missing the variable — unfortunately, there was no default fallback behaviour defined in the chart at the time.

It's much easier to test for these kinds of issues if the chart and the code are in the same repo and can be tested in the same branch.

We do this even when it seems like overkill at the beginning. We've worked on services that have few dependencies: For each service, the Helm chart just deploys one main container with a specific Docker tag. The image name and docker tag are passed in through variables. Despite this, we still avoided using a shared chart — instead opting to put separate charts in each service repository.

Primarily this was because we were only dealing with four services. But our developers also prefer to own all configurations that affect CI/CD. This is not always the case however, so it's a good time to examine the other dimension:

Team structure

The issue of chart maintenance also depends on who owns the deployment process.

There's a great article from another Helm maintainer, Matt Farina, where he talks about the complexity that Helm is trying to solve. He elucidates the three main roles who have to deal with complexity in Kubernetes. For the sake of clarity, I'm going to paraphrase him and describe the roles as follows:

1. **App Developers —** these people build the service, add features, and fix bugs.
2. **Deployers —** these people are in charge of pushing the application put into the world. Hopefully, there's decent automation that deploys the app for them, but they know how it works and can change it if they need to.

3. **System Engineers —** these people maintain the Kubernetes environment that the deployers deploy *into*. They're specialists in managing computational resources and try to minimize any service downtime.

The first and third roles match up to normal job titles that you might find in the wild. The "deployer" role is a little more ambiguous.

This role is often taken over by someone who is also one of the other two roles — and this influences how you'll manage your Helm charts.

We've learned this through experience.

DevOps in early-stage startups

As mentioned, we're in the business of providing operational support to startups that often have to scale up at a rapid pace.

We've seen a lot of "unconventional" setups and divisions of labour. In the early stages, app developers might wear different hats and some even help out with sysadmin tasks such as setting up the printer or configuring the office VPN. They'll try their best to get their heads around the other two roles because there isn't anyone else to help them (until we get involved).

Once they've figured out Helm, most app developers will put their charts in the easiest place for them to deal with — i.e. the same repo that they maintain.

DevOps in large organizations

You might work in a larger, structured team, you might have read the previous section with varying degrees of horror.

In this case, you probably have your own DevOps engineer or even an entire DevOps department. And that person or team often feels responsible for the "Deployer" role too. Chances are, they'll favour a more centralized approach such as storing all charts in a chart repository like ChartMuseum. There is less willingness to let app developers get too involved with Helm charts (often for good reason).

For example, I recently watched an old tech talk called "Building Helm Charts From the Ground Up" by VMWare systems engineer Amy Chen. In her intro statement, she said:

> *"In infrastructure, your main goal is to always prepare for failure right? there is no trust — in the sense of like I don't really want to trust my app developers and I don't really need*

> *to trust my app developers”*

This is understandable. You don’t want app developers messing with settings like CPU and memory limits, or pod disruption budgets. But the whole concept of “DevOps culture” evolved specifically to improve the sometimes distant relationship between infrastructure maintainers and developers.

Applying DevOps culture

Atlassian (owner of JIRA and Trello) published a “Team Playbook” which defines DevOps culture like this:

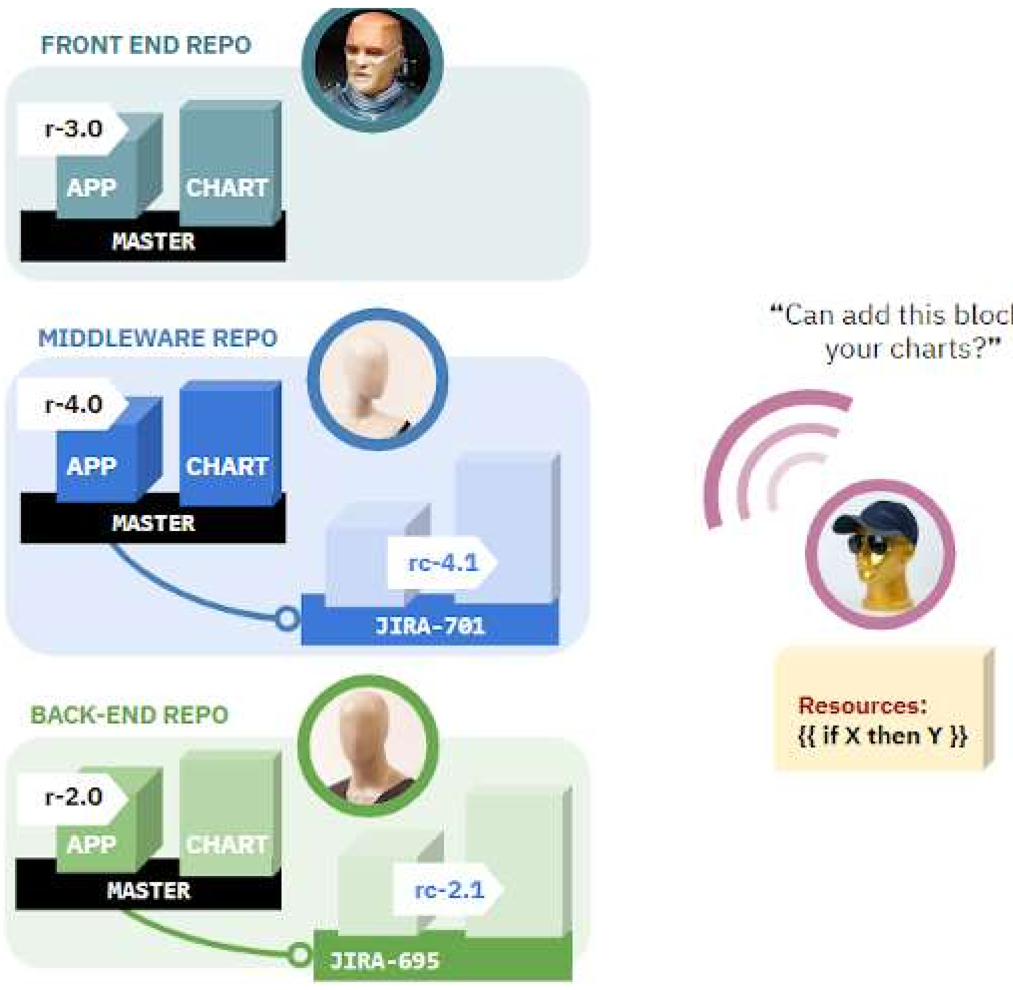
> *‘DevOps culture is all about a shared understanding between developers and operations, and sharing responsibility for the software they build. That means increasing transparency, communication, and collaboration across development, IT/operations, and “the business”.’*

If you apply this practically to Helm chart maintenance and infrastructure configuration in general, you would put much of the responsibility in the hands of application developers. They also assume the role of “Deployer” and change the configuration in repositories that they own.

System engineers can still centralize the settings that they maintain exclusively. For example, some teams also maintain a central infrastructure repo that holds common resources such Terraform configurations or Helm files that are needed to bootstrap new projects (e.g. for setting up the ingress controller and cert manager). Helm 3 also supports so-called “library charts” which only be deployed as part of another chart. This makes it easier to separate responsibility for common and service-specific changes.

Even when charts live in the service repos, system engineers can still act as gatekeepers for important changes. For example, you can use a GitHub CODEOWNERS file to ensure that system engineers are added as reviewers for any changes to the chart directory if your repo.

If system engineers need to proactively make changes that aren’t related to the application development, they can direct developers to make changes for them and explain why the changes are essential. Consider the previous illustration updated to reflect this scenario:



The developers get to learn more about the infrastructure and how those changes affect their services.

The Guidelines

If there's a simple rule of thumb to go by it's this: *look at option #3 first*. Try to maintain a Helm chart for each service in the service repository. Or at least consider the hybrid approach that I've described previously.

If you have tens of services that are all very similar, then a shared chart is the better bet. Just bear in mind that you'll have to maintain it in a central repo. This increases the risk of an accidental coupling which could break a service deployment. Heightened risk means you'll be more cautious about deploying, which in turn, means that you'll deploy less often.

Even if you do have service-specific charts, you might need to store them centrally because you don't have the people or expertise to look after them in a distributed way. Or perhaps your organization requires a clear separation of responsibility between "deployer" and "application developer".

Regardless of what you decide to do — I hope I've clarified what you need to take into account when making your final call. Being a "deployer" ain't easy, especially when it's not your day job.

Thanks to Paul Sturm.

DevOps Helm Deployment Helm Chart Infrastructure As Code