HELM:

The idea is very close to the traditional understanding of a package manager. Helm is a package manager for Kubernetes that makes it easier to deploy and manage applications which simplifies the process. Developers can define and share charts, which contain all the necessary resources and settings for running applications on Kubernetes. Helm lets developers quickly install, upgrade, or roll back applications on Kubernetes clusters, reducing the complexity and manual work involved in managing deployments.

Why Use Helm?   
**Helm is a package manager for Kubernetes**. Like NPM helps you maintain your dependencies for a node project, similarly, helm helps to organize maintain and store applications built for running on Kubernetes.

Helm is a handy tool that helps with packaging, deploying, and managing applications on Kubernetes. It has two main parts: Helm Charts and the Helm Client. Helm Charts are like ready-to-use templates that describe how a Kubernetes application should be structured and configured. The Helm Client is a command-line tool that allows you to work with Helm and take care of application releases. It's like the control panel for managing your applications on Kubernetes.

With Helm you can:

* Install software.
* Automatically install software dependencies.
* Upgrade software.
* Configure software deployments.
* Fetch software packages from repositories.

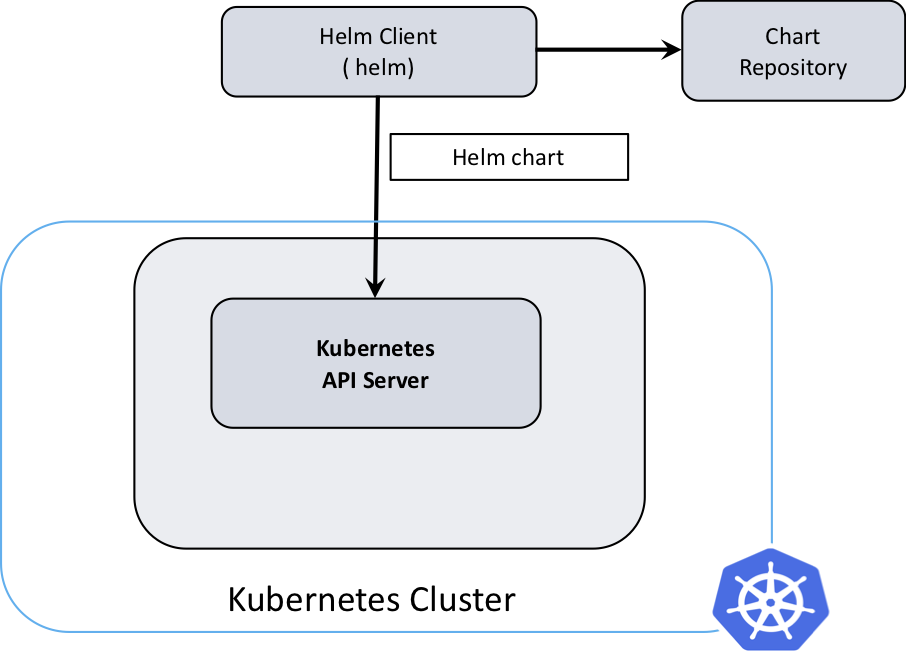
### Architecture

Helm’s architecture has two main components: client and library.

A Helm client is a command-line utility for end users to control local chart development and manage repositories and releases. Just like using the MySQL database client to run MySQL commands, you use the Helm client to run Helm commands.

The Helm library does the heavy lifting. It has the code needed to carry out the tasks specified in the Helm command. The library takes care of combining the configuration and chart files to create any release.

The Helm architecture has improved significantly between versions 2 to 3. In version 2, there was a Tiller server that acted as a middleman between the Helm client and the Kubernetes API server. It was responsible for keeping track of all the resources managed by Helm. However, in version 3, the Tiller server was removed in favour of a client-only approach. Now, Helm directly connects to the Kubernetes API server through an API connection.



Helm Chart Structure

The files and directories of a Helm chart each have a specific function:

nginx-chart/

|-- Chart.yaml

|-- charts

|-- templates

|   |-- NOTES.txt

|   |-- \_helpers.tpl

|   |-- deployment.yaml

|   |-- configmap.yaml

|   |-- ingress.yaml

|   |-- service.yaml

|   `-- tests

|       `-- test-connection.yaml

`-- values.yaml

**Charts**: The charts directory contains other charts that the main chart depends on. A single chart could depend on several charts. Thus, there might be multiple charts in this directory.

**Templates**: This folder keeps the important information for the chart being installed. For instance, if you're installing an app that requires a service, a config map and secrets, this folder would have files like deployment.yaml, service.yaml, config.yaml, and secrets.yaml. These files get their values from the values.yaml file.

**Chart.yaml:** It contains information about the helm chart like version, name, description, etc.

**Values.yaml:** In this file, we define the values for the YAML templates. For example, image name, replica count, HPA values, etc. As we explained earlier only the values.yaml file changes in each environment. Also, you can override these values dynamically or at the time of installing the chart using --values or --set command.

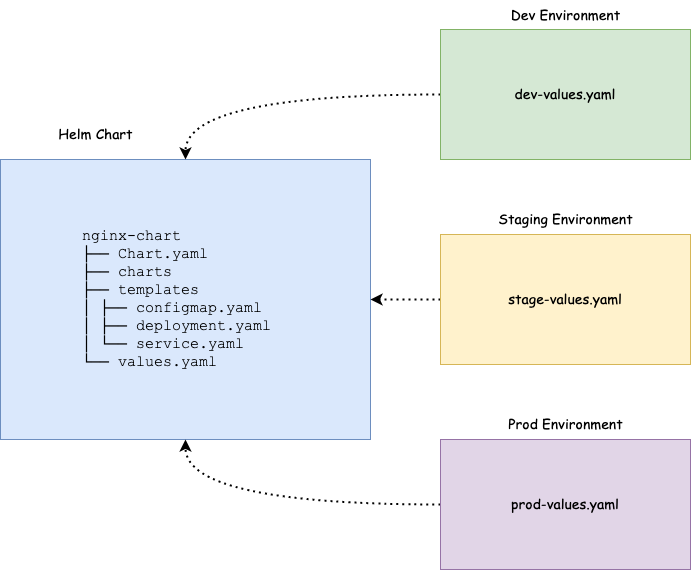
What problem does it solve?

Helm charts address the challenge of managing and deploying intricate applications on a Kubernetes cluster, simplifying the process and reducing complexity.

Let us look into a practical scenario where Helm is helpful. Suppose you want to deploy your application in a production environment with ten replicas. To accomplish this, you would define the desired number of replicas within the application's deployment YAML file and initiate the deployment.

Now, run the same application in a staging environment. Assume that you need three replicas in staging and that you will run an internal application build in the staging environment. To do this, update the replicas count and the Docker image tag in the deployment YAML file and then use it in the staging Kubernetes cluster

As your application becomes more complex, the number of YAML files increases. Eventually, the configurable fields in the YAML file also increase. Soon, updating many YAML files to deploy the same app in different environments becomes hard to manage.



Using Helm, you can parameterize the fields depending on the environment. In the previous example, instead of using a static value for replicas and Docker images, you can take the value for these fields from another file. This file is called [values.yaml](https://helm.sh/docs/chart_template_guide/values_files/)