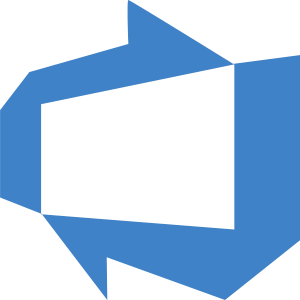
3. Best-Practices

**best-practises/AzureDevOps.md**

**Introduction**

***This page contains all the Best-Practices of Azure Dev0ps.***

[](https://github.com/techslateramu/allinone/blob/main/best-practises/images/AzureDevOps.png)

**Creating the perfect Azure DevOps culture for best practice.**

* Azure DevOps supports creating a collaborative working environment that combines both IT operations and development to generate services, products, and tools.
* This is important for any development team, as it’s the first step to creating a perfect working environment. If the development environment isn’t well organized, then it is very hard for any team to be productive.
* By using the Azure DevOps tool for best practice, you will get some exciting advantages like traceability, diagnosing resources, providing permissions, and more.
* For this purpose, Azure DevOps includes an exciting tool named Azure Pipeline. This tool is a combination of both Continuous Integration (CI) and Continuous Deployment (CD). It also supports you by building, testing, and releasing your code together.
* Through the Azure DevOps tool, it’s very easy to create a development environment through the Azure DevOps portal.
* To do this, you need to navigate to the Pipelines page, then choose environments, and lastly click on Create Environment. After that, provide the necessary information about the environment and then click on “create”.
* When you are creating an environment with Azure DevOps, you can easily keep an eye on the current working environment, like the commits or resources recently deployed to the environment.
* You can check the history of deployment. Through Azure DevOps, you can connect to the services that are defined in the environment automatically.

**Analysis of DevOps workflow**

* The most exciting benefit of Azure DevOps is that it enables integration with other tools and platforms. It allows you to analyze and select the best tools for the task. Azure DevOps allows you to track your work. For this purpose, you need to create work items. This work item can be issues, tasks, or something else.
* You can choose various system processes such as Basic, Agile, Scrum, or Capability Maturity Model Integration (CMMI) depending on your project needs. You have to fill in a form when creating a work item. In this form, you have to describe the work that needs to be done, assign new work to team members, check the current working status, or collaborate with other team members.
* There are some prerequisites. You have to create and connect a project before you can include a work item. Besides, you have to add members or competitors and must be granted basic access before creating a work item.
* Using the Azure DevOps tool, you can provide stakeholders access to a project that is private, but you can grant access to general stakeholders for a public project. You can create a checklist where you can open or modify work items, view boards, and add child tasks. Below are the fields that you may need to fill up when opening a work item.

1. **State** – When creating new work items, you need to provide the current state of the work, like progress. You need to continuously update it to visualize the current status.
2. **Reason** – You need to provide the reason here. But you need to use it as the default first.
3. **Area** – You can leave it blank. You have to choose the path of the area associated with the project.
4. **Iteration** – Here you need to choose the iteration or sprint in which the work needs to be done. But you can leave it blank for a later assignment.
5. **Description** – This is the important field for opening a work item. Here you need to describe the full details of the task. Here you can include the goal of the task, special interactions, etc.
6. **Acceptance Criteria** – Here you need to provide the criteria that need to be met to close the work item.
7. **Priority** – You can provide priority to the work items. There are different levels of priority available. When the priority is 1, it means the product can’t be shipped without successful completion of the task. When the priority is 2, that means the product needs successful completion and needs to be addressed immediately. When the priority level is 3, it means it’s an optional work item based on resources. Lastly, when the priority is 4, that means there is no requirement for the resolution of the work.
8. **Value Area** – You need to provide a rating for the work items related to the business. It is set to business by default , but you can set it to architectural if it’s related to technical services.
9. **Efforts, size, and Story points** – Provide the estimate of the work that is required to complete the issue or task.

**Using automation**

* Azure DevOps supports automation, which is the biggest weapon for any developer team. It can create your work items automatically. You can run automated tests from your test plans by using Azure DevOps. To create automated test cases, the test cases must be in your test plans and need to be run directly from Azure Test Plans. We will share some benefits of automating tests below.
* First of all, it provides a user-friendly process for testing that may not be well versed in running tests in build or release workflows.
* Secondly, there is flexibility to run tests that are selected based on demand and requirements. You will have the ability to run tests again and again that failed due to some test infrastructure issues. Please note that your project must be connected, otherwise create a new one. Your team members must be added to your project, otherwise, they can’t contribute to the project events. But you can go for a manual tasting too.

**Working with extension**

When you are done with the other practices, it’s time to work with some additional features of Azure DevOps. Azure DevOps also can provide you with additional tools for any development process by extensions. You can explore the extensions and find your required tool. Azure DevOps has a large collection of extensions for numerous purposes. There are some exciting extensions that can add value to your DevOps environment immediately. You can easily find them on Microsoft’s official marketplace for visual studios. These extensions are also known as ADO (Azure DevOps). Below shared some highly recommended extensions that you can use,

**Retrospectives** – It’s an extension that enriches the Azure Boards to a place where a can have to reflect, inspect, opportunity, and consider their last sprint. This extension is the key pillar of the scrum.

**Estimate** – It’s another extension that helps a team to come to an agreement transparently. It helps to create plans for their upcoming sprint. The most interesting thing about this extension is all the members of a team can vote simultaneously on the same item.

**Code Search** – This extension can go through the whole repositories of an organization. It also can create indexes by using the code search.

**best-practises/DevOps principles.md**

# Introduction

## This page contains all key principles of DevOps

## DevOps is a software development methodology that emphasizes collaboration and communication between development and operations teams to improve the software delivery process. The following are some of the key principles of DevOps:

1. **Collaboration**:

Collaboration between development and operations teams is essential to ensure that software is delivered quickly and with high quality. Teams should work together closely to identify and resolve issues as quickly as possible.

1. **Continuous integration and delivery**:

Continuous integration and delivery (CI/CD) is a set of practices that enables teams to build, test, and deploy software quickly and reliably. CI/CD pipelines automate the process of building, testing, and deploying software, allowing teams to deliver new features and bug fixes quickly.

1. **Infrastructure as code**:

Infrastructure as code (IaC) is the practice of managing infrastructure using code. This enables teams to automate the deployment and management of infrastructure, reducing the risk of errors and increasing the speed of deployment.

1. **Monitoring and logging**:

Monitoring and logging are critical components of DevOps. Teams should monitor the performance of their systems and applications, and collect logs and metrics to help them identify and diagnose issues.

1. **Continuous improvement**:

Continuous improvement is an essential principle of DevOps. Teams should constantly seek to improve their processes, tools, and systems, and look for ways to optimize and automate their workflows.

1. **Automation**:

Automation is a key principle of DevOps. Teams should automate as many tasks as possible, including testing, deployment, and infrastructure management. Automation helps teams reduce errors and increase the speed and reliability of their processes.

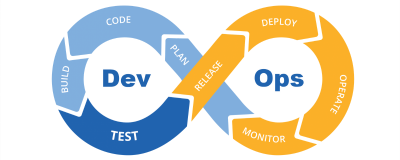
1. **Security**:

Security is a critical consideration in DevOps. Teams should integrate security into every stage of the software development lifecycle, and use tools and processes to ensure that software is secure and compliant with relevant regulations and standards.

**best-practises/DevOps.md**

**Introduction**

***This page contains all the Best-Practices of DevOps.***

[](https://github.com/techslateramu/allinone/blob/main/best-practises/images/devops_2.png)

**DevOps Compose-Best Practices**

DevOps (Development Operations) is an approach to software development that emphasizes collaboration and communication between development and operations teams. Here are some best practices for DevOps:

**Practice 1:Automate as much as possible:**

* Automating repetitive tasks such as testing, deployment, and monitoring can save a lot of time and reduce errors.

**Practice 2: Continuous integration and delivery**

* Adopt a continuous integration and delivery (CI/CD) pipeline to ensure that changes to code are quickly and safely integrated into the main codebase.

**Practice 3: Infrastructure as Code (IaC)**

* Use IaC tools such as Ansible, Chef, or Puppet to automate the deployment and management of infrastructure.

**Practice 4: Monitoring and logging**

* Implement monitoring and logging tools to proactively detect and troubleshoot issues in real-time.

**Practice 5: Collaboration and communication**

* Foster a culture of collaboration and communication between development and operations teams to ensure that everyone is working towards the same goals.

**Practice 6: Security**

* Implement security best practices throughout the entire DevOps pipeline, including code reviews, vulnerability scanning, and access controls.

**Practice 7: Continuous improvement**

* Regularly evaluate the DevOps pipeline and processes, and continuously make improvements to optimize efficiency and productivity.

By following these best practices, organizations can accelerate software development, reduce errors, and improve overall product quality.

**Practice 8: Cloud-native Technologies**

* Use cloud-native technologies like Kubernetes and Docker for containerization and orchestration

**Practice 9: Disaster Recovery and Business Continuity**

* Implement disaster recovery and business continuity plans
* Test and update plans regularly

**Practice 10: Release Management**

* Use release management tools like GitLab or Octopus Deploy to manage software releases
* Ensure all changes are thoroughly tested and documented before release

**Practice 11: Scalability and Flexibility**

* Ensure systems are scalable and flexible to handle changing workloads.
* Use tools like autoscaling to adjust resources based on demand

**Practice 12: Version Control**

* Use version control tools like Git for code management.
* Ensure all changes are tracked and documented

**Practice 13: Code Review**

* Conduct code reviews to ensure code quality and adherence to best practices
* Use tools like GitHub or Bitbucket for code review

**Practice 14: Feedback and Continuous Improvement**

* Use feedback loops to continually improve processes and systems
* Collect feedback from customers and stakeholders to identify areas for improvement.

**best-practises/Docker.md**

**Introduction**

***This page contains all the Best-Practices of Docker.***

[](https://github.com/techslateramu/allinone/blob/main/best-practises/images/bestprac.jpg)

**Docker Compose-Best Practices**

**Practice 1:**

* Keep your docker-compose file version controlled (SCM) and next to the application code. Your Docker Compose file is there to describe your application components. This means any changes in your application should be reflected in your Docker Compose file. Changes to the port of a service or a new service added to the application should be made in your Docker Compose file as well.
* That’s why the best place for your Docker Compose file is next to your app and version-controlled in lockstep with your application. Keep Docker images explicitly versioned.

**Practice 2: Always use defined volumes.**

* Docker Compose allows you to define volumes for each service the same way they are defined with the docker run command. While this is possible, it is not advisable. The best way to use volumes in Docker Compose is to define the named volumes on top of your Compose file and use those names.

**Practice 3: Make your containers traceable to codebase.**

* Use a naming convention that allows you to look at a running container and tell you exactly how that container was built, using which git repo and using which Dockerfile.
* Make your containers traceable to git commit — Use git ref as the tag for the containers running.

**Practice 4: Make sure your services return the correct exit code.**

* We all use logging or console output to show errors in our apps. In a containerized world, exit codes have a special place. Orchestrators (like Compose, Swarm, and Kubernetes) use exit codes to tell if a service started or failed to start.
* The change of behavior in starting containers between Compose and Swarm means exist codes are even more important. While Docker Compose allows defining service dependencies, Docker Swarm (and Kubernetes) keep trying to start a crashing container (that exits with a code other than 0) several times.
* This means you need to make sure not only errors are logged but the exit codes are accurate as they will show up in the orchestrator’s logs and can give you valuable clues as to what’s going on with your application.

**Practice 5: In a docker-compose file, mention the important services first—it is usually considered good practice to first mention the image for a service.**

For a service, group the parameter categories-

* Networking configurations should be kept close together.
* Security things as well.
* It’s also good to keep entrypoint and command next to each other.

**Practice 6: Sort services alphabetically-**

* If docker-compose files grow with time, then it’s tedious to find services. If the services are ordered alphabetically, then it will be easy to find them.

**Practice7: Using volume folder mapping (instead of a file mapping)**

* Folder mapping allows us to place multiple environment files that may be required for different scripts
* After restarting your containers, they will use the updated variables.

**Practice8: Use docker-compose.yml as much as possible.**

**Docker File-Best Practices**

**Practice 1 : Use a.dockerignore file.**

* The best way is to put the Dockerfile inside the empty directory and then add only the application and configuration files required for building the Docker image.
* To increase the build’s performance, you can exclude files and directories by adding an a.dockerignore file to that directory as well.

**Practice 2: Containers should be immutable and ephemeral.**

* The container created with the image produced by Dockerfile should be ephemeral and immutable. In other words, the container should be destroyed and a new one built and put in place with an absolute minimum set-up and configuration.

**Practice 3: Minimize the number of layers and consolidate instructions.**

* Each instruction in the Dockerfile adds an extra layer to the Docker image. The number of instructions and layers should be kept to a minimum as this ultimately affects build performance and time.

**Practice 4: Avoid installing unnecessary packages**

In order to reduce complexity, dependencies, file sizes, and build times, avoid installing unnecessary packages.

**Practice 5 : Sort multi-line arguments**

Sorting multiline arguments alphanumerically will help avoid duplication of packages and make the list much easier to update.

**Practice 6 : Build cache**

* While building an image, Docker will step through the instructions mentioned in the Dockerfile, executing them in chronological order. As each instruction is examined, Docker will look for an existing image layer in its cache that it can reuse, rather than creating a new image layer.
* However, when Docker is not allowed to use its cache, then the basic rules Docker will follow to find a matching image are mentioned below:
* Starting with a base image that is already in the cache, the next instruction is compared against all child images derived from that base image to see if one of them was built using the exact same instruction. If not, the cache is invalidated.
* For the ADD and COPY instructions, the contents of the file(s) in the image are examined and a checksum is calculated for each file. During the cache lookup, the checksum is compared against the checksum in the existing images. If anything has changed in the file(s), such as the contents and metadata, then the cache is invalidated.
* Aside from the ADD and COPY commands, cache checking will not look at the files in the container to determine a cache match. For example, when processing a RUN apt-get -y update command the files updated in the container will not be examined to determine if a cache hit exists. In that case just the command string itself will be used to find a match.
* Once the cache is invalidated, all subsequent Dockerfile commands will generate new images and the cache will not be used.

**Practice 7 : Build every time**

* Building docker images is very fast as docker makes use of previously cached build steps (default). By building the image every time, one can use containers as reliable artifacts. For example, one can go back and run a container from previous docker image to inspect a problem, or can run long tests on the previous version image while editing the code.

**Practcie 8 :Dockerfile for Development Environment**

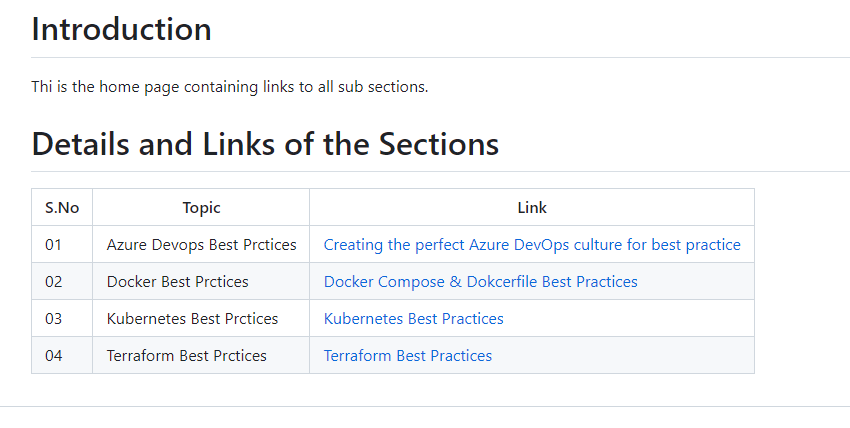
* For a development environment, map your source code on the host to a container using a volume. This enables to choose the editor of your choice on the host and test the application right away in the container. This is done by mounting the application build folder as a volume rather than copying the build artifact using the ADD command in the Dockerfile.

**Practice 9: Understand CMD and ENTRYPOINT**

CMD simply sets a command to run in the image if no arguments are passed to docker run, while ENTRYPOINT is meant to make your image behave like a binary.

* If the Dockerfile uses only CMD, the specified command is executed if no arguments are passed to docker run.
* If the Dockerfile uses only ENTRYPOINT, the arguments passed to docker run are always passed to the entrypoint; the entrypoint is executed if no arguments are passed to docker run.
* If the Dockerfile declares both ENTRYPOINT and CMD and no arguments are passed to docker run, then the argument(s) to CMD are passed to the declared entrypoint.

**best-practises/README.md**



**best-practises/Terraform.md**

Introduction

This page contains all the Best-Practices of Terraform.

DevOps

Terraform Best Practices

Practice 1: Host Terraform Code in the Git Repository

GitHub is a comprehensive DevOps and collaboration platform which is well known for its version control features.

By using GitHub for version control and collaboration, operators can better cooperate with application developers throughout the software lifecycle.

If you are a Terraform user, you should save your configuration files in a Version Control System (VCS).

Practice 2: Use .gitignore to exclude Terraform State Files, State Directory Backups, and Core.

Always have an a.gitignore file in your repository with all the required rules to ignore unnecessary files by Git. This way, you won’t push files unknowingly.

Practice 3: Using a Consistent File Structure

Use folders in your project structure when using modules.

Modules are folders that contain config files that have been created in a way that allows for the code to be reused.

A README.md file should be included in each repository.

Create main.tf to call modules, store locals, and data sources to create all the required resources.

It’s a good idea to have a provider.tf with provider details.

Have a variables.tf file where you can store declarations of variables used in main.tf and outputs.tf should contain outputs.

Practice 4: Auto Format Terraform Files.

Should we just avoid writing ugly code? After all, we all despise reading it.

The HashiCorp Terraform language follows the same style guidelines as most other computer languages.

A single missing bracket or excessive indentation can make your Terraform configuration files difficult to read, maintain, and understand, resulting in a negative experience.

However, you can use the ‘terraform fmt’ command to repair all code discrepancies at once. Terraform configuration files are rewritten in a consistent structure and style using the ‘terraform fmt’ command.

Practice 5: Avoid Hard Coding Resources

Its best practice to avoid hard coding resources in Terraform configuration files. Instead, the values should be placed as variables.

Its best practice to avoid hard coding resources in Terraform configuration files. Instead, the values should be placed as variables.

Practice 6: Follow Naming Convention

Instead of – (dash), use \_ (underscore) everywhere (in resource names, data source names, variable names, outputs, etc.).

Use lowercase letters and numbers whenever possible.

When it comes to names, single nouns should always be used.

Use -(dash) inside arguments and in locations where the value will be visible to a human (e.g.:, name of Resource group, name of Storage account) .

Practice 7: Use Modules

Every Terraform practitioner should employ modules in accordance with the following guidelines:

Begin writing your setup.

Organize and encapsulate your code using local modules.

Find relevant modules by searching the public Terraform Registry.

Share modules with your team after they’ve been published.

Practice 8: Run Terraform Command with var-file.

Maintain multiple .tfvars files with the definition of variables so that you can pass the required file with var-file flag to the ‘terraform plan’ or ‘terraform apply’ command.

Practice 9: Manage Terraform State on a Remote Storage.

When you’re working on a project with multiple users, you should always use Terraform backend to save the state file in a shared remote store.

Practice 10: Generate README for each Module with Input and Output Variables.

You must have a self-explanatory README.md as a part of all your Terraform projects.

Practice 11: Use Workspaces.

Using multiple working folders is the simplest way to manage numerous instances of a setup with totally distinct state data.

Make use of Terraform workspaces to create multiple environments like Dev, QA, Pre-Prod, Prod, and more using the same Terraform configuration files and saving the state files for each environment in the same backend.

Practice 12: Avoid Storing Credentials in the Terraform Code.

Do not store sensitive information in Terraform configuration files; instead use Secret Management Systems like HashiCorp Vault, Azure key vault.

Practice 13: Use Terraform Import.

Even if you have resources that were provisioned manually, import them into Terraform. This way, you’ll be able to use Terraform to manage these resources in the future and throughout their lifecycle.

Practice 14: Automate your Deployment with a CI / CD Pipeline.

Decide on whether you want to store your Terraform Configuration in a separate repository or combine it with your application code and have a CI/CD pipeline in place to create the infrastructure.

Practice 15: Stay Up to Date

Always update your Terraform version and code upon major releases. Otherwise, if you skip numerous major releases, upgrading becomes quite difficult.

Practice 16: Pin your Terraform and provider version

Always set required\_providers version and Terraform required\_version in the terraform {} configuration block.

Practice 17: Validate your Terraform Code

Always run the ‘terraform validate’ command while you are working on writing Terraform configuration files and make it a habit to identify and address problems as early as possible.

**best-practises/kubernetes.md**

**Introduction**

***This page contains all the Best-Practices of kubernetes.***

[](https://github.com/techslateramu/allinone/blob/main/best-practises/images/kuberneteslogo.png)

**Kubernetes Best Practices**

**Practice1: Choosing the Right Kubernetes Object (Mandatory).\*\***

* Kubernetes supports a variety of objects. Depending on the use case, one should choose the right Kubernetes object. People get confused about when to use deployment and when to use stateful sets.
* **Deployment:** Deployment can be used when your application doesn’t require persistent data to be stored in a block storage(say EBS, Azure Disks , etc) and doesn’t require clustering. If the persistent data needs to be stored in a file share, then deployment can be chosen.
* **StatefulSet:** StatefulSet Kind can be used when your application stores persistent data in block storage, or clustering is required, or you want to have static name for the pods. StatefulSets can dynamically spin up PVC(provided you run on cloud providers like EKS, AKS, GKE, etc) using the volumeClaimTemplate .

**Practice2: Startup, Liveness, and Readiness Probe (Mandatory)**

* **Startup Probe**: Sometimes you would be deploying legacy applications, which would take a long time to get started initially. In such cases, it can be tricky to set up liveness probe parameters without compromising the fast response to deadlocks that motivated such a probe.
* That’s when a startup probe can be used, which would have the same kind of health check pattern as that of liveness, except the fact that failureThreshold and periodSeconds are long enough to cover the worst-case startup time.
* **Liveness Probe**: Liveness Probe ensures that the application is healthy by making a health check periodically(based on periodSeconds given). If the health check fails for a consecutive 3 times(by default failureThreshold is 3 and users can change it), then container gets killed and subject to the pod’s restartPolicy . It is always a must to have liveness probe for the applications .
* **Readiness Probe**: Readiness probe ensures that application is ready to receive the traffic. Until the readiness health is satisfied, the container will not accept any traffic. This will make sure that no traffic comes in when the application is not yet ready. Readiness Probe failures will not kill the container.

**Practice3: Setting the Resource Caps(Mandatory)**

* Kubernetes has different types of quality of service.
* **Guaranteed**: Every container in the pod should have CPU and memory limits and requests set. Also, the request and limit values should be the same. Pods of this QoS will be the last choice when performing eviction.
* **Burstable**: At least one container in the pod has a memory or CPU request or limit. If the container has both request and limits but the value of request and limits are different, it still falls into the burstable category. The scheduler will look into the requests section of the resources section and will allocate the pod to the corresponding node.

**Practice4: Assigning Pods to Nodes(Mandatory)**

* For making the application fault-tolerant, we should ensure replicas of the pods are distributed across nodes/availability zones. This can be achieved by anti-affinity for pods. In few scenarios, we would like to run another application(say application B) in the same node where application A runs.

**Practice5: HA Setup(Mandatory)**

* Each of the applications that we deploy should have at least of 2 replicas, so that the load is distributed and the application is highly available(by following the above practices as well .

**Practice 6 : Setting Pod Disruption Budgets(Mandatory)**

* Pod Disruption Budgets(PDB) will give users the privilege to decide how many instances can be down at the same time for a short period due to a voluntary disruption. The best example for voluntary disruption is putting the node under maintenance mode(draining the node) or performing upgrades.

**Practice 7: Proper Labels(Mandatory)**

* In Kubernetes, we can add labels to almost all the objects. In many occasions people give least importance to labelling. Labels play vital role to forward the traffic when accessing applications using Kubernetes Service(which is how we access the applications always, since pod’s IP are dynamic). Apart from that, if user wants to group/list the objects, at that time labels plays a key role. Have some meaningful and useful labels based on the use cases.

**Practice 8 : Pod’s Privilege(Mandatory)**

* Ensure that the pod’s have least privilege. Avoid running the containers with root permission unless the application requires such privilege. This will have security risks.

**Practice 9 : Horizontal Pod Autoscaling(Optional)**

* Kubernetes supports Horizontal Pod Autoscaling(HPA), where it auto scales the number of replicas for the pod based on the metric user gives. By default Kubernetes Metric Server provides CPU and Memory metrics with which users can auto scale.
* If user wants to use some other metric apart from CPU/Memory, then that can also be achieved by adopting Prometheus Adapter. Reason why HPA is put under Optional category is, not many applications can adopt HPA(For example: Redis, Elasticsearch, Mongo DB , etc).

**Practice 10 : Auto Re-deployment on Config Map Update(Optional)**

* When a config is updated in a config map that is attached to a pod, we would have to delete the pods to make the changes getting reflected immediately. This is a manual activity.
* If user wants to bypass this manual activity and instead wants the pods to get re-deployed on config map updates, then it can be achieved using a package manager like Helm.