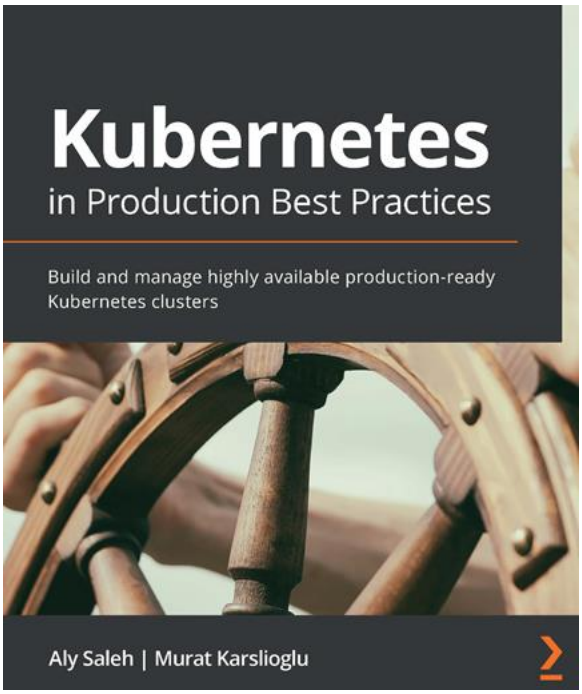
Book review **– Kubernetes in Production Best Practices by Aly Saleh and Murat Karslioglu** reviewed by **Renzo Tomà and Marcos Vale**

Book : https://www.amazon.com/Kubernetes-Production-Best-Practices-production-ready-ebook/dp/B08T5Y4CJP/ref=sr_1_1

Git Repo: https://github.com/PacktPublishing/Kubernetes-in-Production-Best-Practices

Summary: Excellent k8s Production best practices book - useful designs, principles, readiness checklist, tips, advices and examples! Good k8s sample codes with terraform and Ansible playbooks for automation/IaC!



Chap 1 – Introduction to Kubernetes Infrastructure and Production-Readiness

Chap 1 is important and fundamental. It has a good production-readiness checklist, 12 principles of infrastructure design and 10 CNCF stages. It also talks about application and deployment best practices. This chapter has many high level production readiness checklist. There is an excellent list of 12 principles of infrastructure design and management.

I suggest to add 1 more on top of the 12 design principles: Security (It is in the production readiness checklist but better if also in design principles). It is important to shift left and plan/add Security early in every k8s design and management.

I also suggest to add one more in Production readiness checklist: Knowledge and trainings! No matter how much Automation and Everything as Code done, we still need well trained and knowledgeable k8s admins/developers to develop, run and operate the k8s cluster and apps/deployments securely, smoothly and effectively! So, it is important to train your staff and CKA/D/S exams can be a good starting point.

Good Production k8s challenges:

Why Kubernetes is challenging in production

1. Kubernetes could be easy to install, but it is complex to operate and maintain. Kubernetes in production brings challenges and difficulties along the way, from scaling, uptime, and security, to resilience, observability, resources utilization, and cost management.
Kubernetes has succeeded in solving container management and orchestration, and it created a standard layer above the compute services. However, Kubernetes still lacks proper or complete support for some essential services, such as **Identity and Access Management** (**IAM**), storage, and image registries.

2. Usually, a Kubernetes cluster belongs to a bigger company's production infrastructure, which includes databases, IAM, **Lightweight Directory Access Protocol** (**LDAP**), messaging, streaming, and others. Bringing a Kubernetes cluster to production requires connecting it to these external infrastructure parts.

3. Even during cloud transformation projects, we expect Kubernetes to manage and integrate with the on-premises infrastructure and services, and this takes production complexity to a next level.

4. Another challenge occurs when teams start adopting Kubernetes with the assumption that it will solve the scaling and uptime problems that their apps have, but they usually do not plan for day-2 issues. This ends up with catastrophic consequences regarding security, scaling, uptime, resource utilization, cluster migrations, upgrades, and performance tuning.

5. Besides the technical challenges, there are management challenges, especially when we use Kubernetes across large organizations that have multiple teams, and if the organization is not well prepared to have the right team structure to operate and manage its Kubernetes infrastructure. This could lead to teams struggling to align around standard tools, best practices, and delivery workflows.

What is k8s Production readiness?

The production-readiness checklist

We have categorized the production-readiness checklist items and mapped them to the corresponding infrastructure layers. Each checklist item represents a design and implementation concern that you need to fulfill to consider your cluster a production-ready. Throughout this book, we will cover the checklist items and their design and implementation details.

Cluster infrastructure

The following checklist items cover the production-readiness requirements on the cluster level:

1. - **Run a highly available control plane**: You can achieve this by running the control plane components on three or more nodes. Another recommended best practice is to deploy the Kubernetes master components and `etcd` on two separate node groups. This is generally to ease `etcd` operations, such as upgrades and backups, and to decrease the radius of control plane failures.

     Also, for large Kubernetes clusters, this allows `etcd` to get proper resource allocation by running it on certain node types that fulfill its extensive I/O needs.

     Finally, avoid deploying pods to the control plane nodes.

2. - **Run a highly available workers group**: You can achieve this by running a group or more of worker nodes with three or more instances. If you are running these workers groups using one of the public cloud providers, you should deploy them within an auto-scaling group and in different availability zones.

     Another essential requirement to achieve worker high availability is to deploy the Kubernetes cluster auto scaler, which enables worker nodes to horizontally upscale and downscale based on the cluster utilization.

3 • **Use a shared storage management solution**: You should consider using a shared storage management solution to persist and manage stateful apps' data. There are plenty of choices, either open source or commercial, such as AWS **Elastic Block Store** (**EBS**), **Elastic File System** (**EFS**), Google Persistent Disk, Azure Disk Storage, ROOK, Ceph, and Portworx. There is no right or wrong choice among them, but it all depends on your application use case and requirements.

4 • **Deploy infrastructure observability stack**: Collecting logs and metrics on the infrastructure level for nodes, network, storage, and other infrastructure components is essential for monitoring a cluster's infrastructure, and also to get useful insights about the cluster's performance, utilization, and troubleshooting outages.

You should deploy a monitoring and alerting stack, such as Node Exporter, Prometheus, and Grafana, and deploy a central logging stack, such as ELK (Elasticsearch, Logstash, and Kibana). Alternatively, you can consider a complete commercial solution, such as Datadog, New Relic, AppDynamics, and so on.

Fulfilling the previous requirements will ensure the production-readiness of the cluster infrastructure. Later in this book, we will show you in more detail how to achieve each of these requirements through infrastructure design, Kubernetes configuration tuning, and third-party tools usage.

suggest to add Fluentd, Splunk, etc..

Cluster services

The following checklist items cover the production-readiness requirements on the cluster services level:

5 • **Control cluster access**: Kubernetes introduces authentication and authorization choices and lets the cluster's admin configure them according to their needs. As a best practice, you should ensure authentication and authorization configuration is tuned and in place. Integrate with an external authentication provider to authenticate cluster's users, such as LDAP, **OpenID Connect** (**OIDC**), and AWS IAM.

For authorization, you need to configure the cluster to enable **Role-Based Access Control** (**RBAC**), **Attribute-Based Access Control** (**ABAC**), and webhooks.

6 • **Hardening the default pod security policy**: **Pod security policy** (**PSP**) is a Kubernetes resource that is used to ensure a pod has to meet specific requirements before getting created.

PSP is deprecated, suggest OPA/Kyverno !

As a best practice, we recommend that you limit any privileged pods within the `kube-system` namespace. For all other namespaces that host your apps pods, we recommend assigning a restrictive default PSP.

7 • **Enforce custom policies and rules**: Rules and policy enforcement are essential for every Kubernetes cluster. This is true for both a small single-tenant cluster and a large multi-tenant one. Kubernetes introduces native objects to achieve this purpose, such as pod security policies, network policies, resource limits, and quotas.

  For custom rules enforcement, you may deploy an open policy agent, such as OPA Gatekeeper. This will enable you to enforce rules such as pods must have resource limits in place, namespaces must have specific labels, images must be from known repositories, and many others.

suggest Kyverno too

8 • **Deploy and fine-tune the cluster DNS**: Running a DNS for Kubernetes clusters is essential for name resolution and service connectivity. Managed Kubernetes comes with cluster DNS pre-deployed, such as CoreDNS. For self-managed clusters, you should consider deploying CoreDNS too. As a best practice, you should fine-tune CoreDNS to minimize errors and failure rates, optimize performance, and adjust caching, and resolution time.

9 • **Deploy and restrict network policies**: Kubernetes allows all traffic between the pods inside a single cluster. This behavior is insecure in a multi-tenant cluster. As a best practice, you need to enable network policies in your cluster, and create a deny-all default policy to block all traffic among the pods, then you create network policies with less restrictive ingress/egress rules to allow the traffic whenever it is needed for between specific pods.

suggest zero trust with Calico, Cilium

10 • **Enforce security checks and conformance testing**: Securing a Kubernetes cluster is not questionable. There are a lot of security configurations to enable and tune for a cluster. This could get tricky for cluster admins, but luckily, there are different tools to scan cluster configuration to assess and ensure that it is secure and meets the minimum security requirements. You have to automate running security scanning tools, such as `kube-scan` for security configuration scanning, `kube-bench` for security benchmarking, and Sonobuoy to run Kubernetes standard conformance tests against the cluster.

suggest kube-linter and others

11 • **Deploy a backup and restore solution**: As with any system, Kubernetes could fail, so you should have a proper backup and restore process in place. You should consider tools to back up data, snapshot the cluster control plane, or back up the `etcd` database.

12 • **Deploy an observability stack for the cluster components**: Monitoring and central logging is essential for Kubernetes components such as `control-plane`, `kubelet`, container runtime, and more. You should deploy a monitoring and alerting stack such as Node Exporter, Prometheus, and Grafana, and deploy a central logging stack, such as EFK (Elasticsearch, Fluentd, and Kibana).

suggest Splunk connect 4 kubernetes

Apps and Deployments level Production Readiness checklist:

Fulfilling the previous requirements will ensure the production-readiness of the cluster services. Later in this book, we will show you in more detail how to achieve each of these requirements through Kubernetes configuration tuning and third-party tools usage.

Apps and deployments

The following checklist items cover the production-readiness requirements on the apps and deployments level:

1. **Automate images quality and vulnerability scanning**: An app image that runs a low-quality app or that is written with poor-quality specs can harm the cluster reliability and other apps running on it. The same goes for images with security vulnerabilities. For that, you should run a pipeline to scan images deployed to the cluster for security vulnerabilities and deviations from quality standards.
2. **Deploy Ingress Controller**: By default, you can expose Kubernetes services outside the cluster using load balancers and node ports. However, the majority of the apps have advanced routing requirements, and deploying an Ingress Controller such as Nginx's Ingress Controller is a de facto solution that you should include in your cluster.
3. **Manage certificates and secrets**: Secrets and TLS certificates are commonly used by modern apps. Kubernetes comes with a built-in `Secrets` object that eases the creation and management of secrets and certificates inside the cluster. In addition to that, you can extend secrets object by deploying other third-party services, such as Sealed Secrets for encrypted secrets, and Cert-Manager to automate certificates from certificate providers such as Let's Encrypt or Vault.
4. **Deploy apps observability stack**: You should make use of Kubernetes' built-in monitoring capabilities, such as defining readiness and liveness probes for the pods. Besides that, you should deploy a central logging stack for the applications' pods. Deploy a blackbox monitoring solution or use a managed service to monitor your apps' endpoints. Finally, consider using application performance monitoring solutions, such as New Relic APM, Datadog APM, AppDynamics APM, and more.

Fulfilling the previous requirements will ensure the production-readiness of the apps and deployments. Later in this book, we will show you in more detail how to achieve each of these requirements through Kubernetes configuration tuning and third-party tool usage.

Kubernetes infrastructure best practices

We have learned about the basics of Kubernetes infrastructure and have got a high-level understanding of the production readiness characteristics of the Kubernetes clusters. Now, you are ready to go through the infrastructure best practices and design principles that will lead you through the way building and operating your production clusters.

The 12 principles of infrastructure design and management

Building a resilient and reliable Kubernetes infrastructure requires more than just getting your cluster up and running with a provisioning tool. Solid infrastructure design is a sequence of architecture decisions and their implementation. Luckily, many organizations and experts put these principles and architectural decisions into real tests.

The following list summarizes the core principles that may lead the decision-maker through the Kubernetes infrastructure design process, and throughout this book, you will learn about these principles in detail, and apply them along the way:

1. **Go managed**: Although managed services could looks pricier than self-hosted ones, it is still preferred over them. In almost every scenario, a managed service is more efficient and reliable than its self-hosted counterpart. We apply this principle to Kubernetes managed services such as **Google Kubernetes Engine** (**GKE**), **Azure Kubernetes Service** (**AKS**), and **Elastic Kubernetes Service** (**EKS**). This goes beyond Kubernetes to every infrastructure service, such as databases, object stores, cache, and many others. Sometimes, the managed service could be less customizable or more expensive than a self-hosted one, but in every other situation, you should always consider first the managed service.
2. **Simplify**: Kubernetes is not a simple platform, either to set up or operate. It solves the complexity of managing internet scale workloads in a world where applications could scale up to serve millions of users, where cloud-native and microservices architectures are the chosen approach for most modern apps.

   For infrastructure creation and operation, we do not need to add another layer of complexity as the infrastructure itself is meant to be a seamless and transparent to the products. Organization's primary concern and focus should remain the product not the infrastructure.

Here comes the simplification principle; it does not mean applying trivial solutions but simplifying the complex ones. This leads us to decisions such as choosing fewer Kubernetes clusters to operate, or avoiding multi-cloud; as long as we do not have a solid use case to justify it.

The simplification principle applies to the infrastructure features and services we deploy to the cluster, as it could be very attractive to add every single service as we think it will make a powerful and feature-rich cluster. On the contrary, this will end up complicating the operations and decreasing platform reliability. Besides, we can apply the same principle to the technology stack and tools we choose, as unifying the tools and technology stack across the teams is proven to be more efficient than having a set of inhomogeneous tools that end up hard to manage, and even if one of these tools is best for a specific use case, simplicity always pays back.

3. **Everything as Code (XaC)**: This is the default practice for modern infrastructure and DevOps teams. It is a recommended approach to use declarative **infrastructure as code** (**IaC**) and **configuration as code** (**CaC**) tools and technologies over their imperative counterparts.
4. **Immutable infrastructure**: Immutability is an infrastructure provisioning concept and principle where we replace system components for each deployment instead of updating them in place. We always create immutable components from images or a declarative code, where we can build, test, and validate these immutable systems and get the same predictable results every time. Docker images and AWS EC2 AMI are examples of this concept.

This important principle leads us to achieve one of the desired characteristics of Kubernetes clusters, which is treating clusters as cattle instead of pets.

5. **Automation**: We live in the era of software automation, as we tend to automate everything; it is more efficient and easier to manage and scale, but we need to take automation with Kubernetes to a further level. Kubernetes comes to automate the containers' life cycle, and it also comes with advanced automation concepts, such as operators and GitOps, which are efficient and can literally automate the automations.
6. **Standardization**: Having a set of standards helps to reduce teams' struggle with aligning and working together, eases the scaling of the processes, improves the overall quality, and increases productivity. This becomes essential for companies and teams planning to use Kubernetes in production, as this involves integrating with different infrastructure parts, migrating services from on-premises to the cloud, and way more complexities.

Defining your set of standards covers processes for operations runbooks and playbooks, as well as technology standardization as using Docker, Kubernetes, and standard tools across teams. These tools should have specific characteristics: open source but battle-tested in production, support the other principles, such as Infrastructure as code, immutability, being cloud-agnostic, and being simple to use, and deploy with minimum infrastructure.

7. **Source of truth**: Having a single source of truth is a cornerstone and an enabler to modern infrastructure management and configuration. Source code control systems such as Git are the standard choice to store and version infrastructure code, where having a single and dedicated source code repository for infrastructure is a recommended practice.
8. **Design for availability**: Kubernetes is a key enabler for the high availability of both the infrastructure and the application layers. Having high availability as a design pillar since day 1 is critical for getting the full power of Kubernetes, so at every design level, you should consider high availability, starting from the cloud and **Infrastructure as a Service** (**IaaS**) level by choosing multi-zone or region architecture, then going through the Kubernetes layer by designing a multi-master cluster, and finally, the application layer by deploying multiple replicas of each service.
9. **Cloud-agnostic**: Being cloud-agnostic means that you can run your workloads on any cloud with a minimal vendor-lock, but take care of getting obsessed with the idea, and make it as a goal on its own. Docker and Kubernetes are the community's answer to creating and managing cloud-agnostic platforms. This principle also goes further to include other technologies and tool selection (think Terraform versus CloudFormation).
10. **Business continuity**: Public cloud with its elasticity solved one problem that always hindered the business continuity for the online services, especially when it made scaling infrastructure almost instant, which enabled small businesses to have the same infrastructure luxury that was previously only for the giant tech companies.

However, coping with the increased scaling needs and making it real-time remains a challenge, and with introducing containers to deploy and run workload apps become easy to deploy and scale in seconds. This put the pressure back on Kubernetes and the underlying infrastructure layers to support such massive real-time scaling capabilities of the containers. You need to make a scaling decision for the future to support business expansion and continuity. Questions such as whether to use a single large cluster versus smaller multiple clusters, how to manage the infrastructure cost, what the nodes' right sizes are, and what the efficient resource utilization strategy is… all of these questions require specific answers and important decisions to be taken!

11. **Plan for failures**: A lot of distributed systems characteristics apply to Kubernetes containerized apps; specifically, fault tolerance, where we expect failures, and we plan for system components failures. When designing a Kubernetes cluster, you have to design it to survive outages and failures by using high-availability principles. But you also have to intentionally plan for failures. You can achieve this through applying chaos engineering ideas, disaster recovery solutions, infrastructure testing, and infrastructure CI/CD.
12. **Operational efficiency**: Companies usually underestimate the effort required to operate containers in production – what to expect on day 2 and beyond, and how to get prepared for outages, cluster upgrades, backups, performance tuning, resource utilization, and cost control. At this phase, companies need to figure out how to deliver changes continuously to an increasing number of production and non-production environments, and without the proper operations practices, this could create bottlenecks and slow down the business growth, and moreover, lead to unreliable systems that cannot fulfill customers' expectations. We witnessed successful Kubernetes production rollouts, but eventually, things fell apart because of operations teams and the weak practices.

These 12 principles are proven to be a common pattern for successful large scale cloud infrastructure rollouts. We will apply these principles through most of this book's chapters, and we will try to highlight each principle when we make a relevant technical decision based on it.

Applications definition and deployment

Probably, a successful and efficient Kubernetes cluster will not save an application's poor design and implementation. Usually, when an application does not follow containerization best practices and a highly available design, it will end up losing the cloud-native benefits provided by the underlying Kubernetes:

1. **Containerization**: This is the de facto standard delivery and deployment form of cloud workloads. For production reliability, containerization best practices play a vital role. You will learn about this principle in detail over the upcoming chapters. Bad practices could lead to production instability and catastrophic outages, such as ignoring containers' graceful shutdown and processes termination signals, and improper application retries to connect to dependent services.
2. **Applications' high availability**: This is by deploying two or more app replicas and making use of Kubernetes' advanced placement techniques (node selectors, taints, Affinity, and labeling) to deploy the replicas into different nodes and availability zones, as well as defining pod disruption policies.
3. **Application monitoring**: This is done by defining readiness and liveness probes with different checks, deploying **Application Performance Monitoring** (**APM**), and using the famous monitoring approaches, such as RED (Rate, Errors, and Duration), and USE (Utilization, Saturation, and Errors).
4. **Deployment strategy**: Kubernetes and cloud-native make deployments easier than ever. These frequent deployments bring benefits to the businesses, such as reducing time to market, faster customer feedback on new features, and increasing product quality overall. However, there are downsides to these as well, as frequent deployments could affect product reliability and uptime if you do not plan and manage properly. This is when defining a deployment and rollback strategy (rolling update, recreate, canary, blue/green, and deployment) comes in place as one of the best practices for application deployments.

The consideration of these four areas will ensure smooth application deployment and operations into the Kubernetes cluster, though further detailed technical decisions should be taken under each of these areas, based on your organization's preferences and Kubernetes use case.

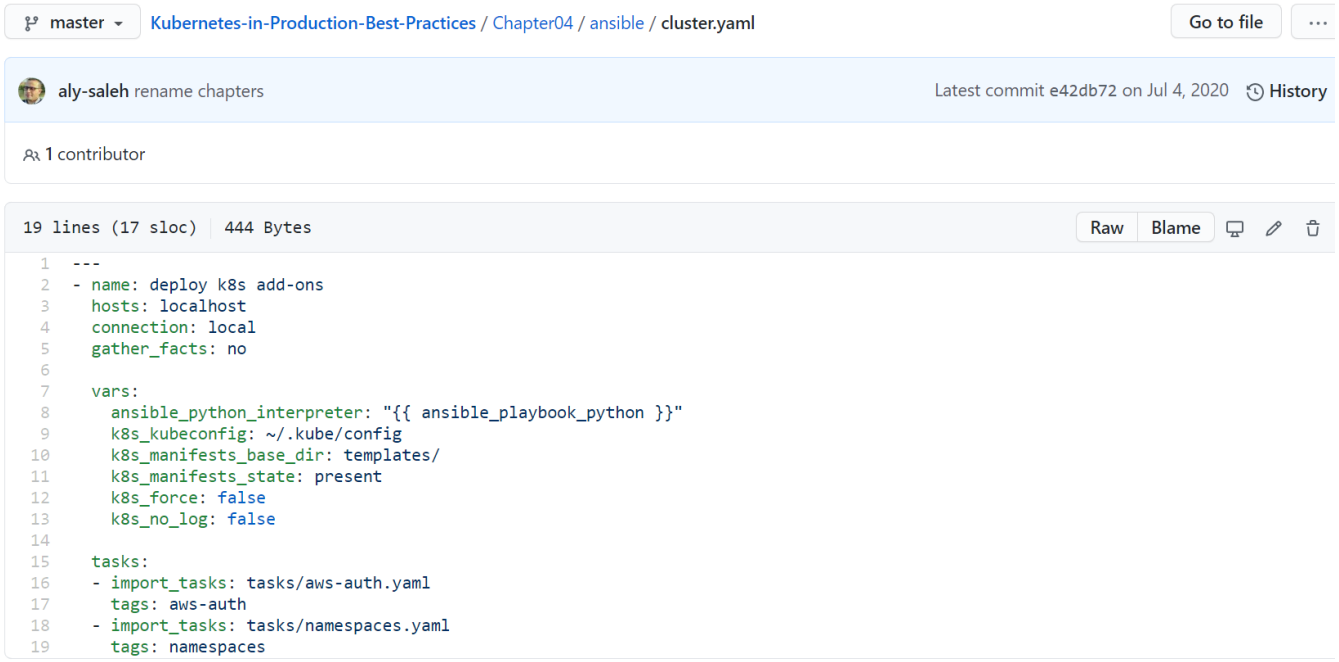
Chap 2 – shows k8s design considerations, Kubernetes deployment strategy alternatives and demo with Amazon EKS infrastructure, e.g. how to choose cluster, node size, tools, and config management.

Chap 3 shows how to provision k8s cluster with AWS and Terraform. There is a GitHub repo with sample terraform files, e.g. eks-cp.tf file defines an instance of the EKS module and workers.tf file defines an instance of the workers.

e.g. https://github.com/PacktPublishing/Kubernetes-in-Production-Best-Practices/blob/master/Chapter03/terraform/modules/eks-workers/main.tf

Chap 4 shows how to manage cluster config with Ansible. Configuration as Code (CaC) is demo with Ansible to deploy more services and add-ons.

e.g. https://github.com/PacktPublishing/Kubernetes-in-Production-Best-Practices/tree/master/Chapter04



Chap 5 shows how to config and enhance k8s networking services, e.g. Amazon CNI plugin, CoreDNS, Ingress Controller.

e.g. https://github.com/PacktPublishing/Kubernetes-in-Production-Best-Practices/tree/master/Chapter05

Create cluster VPC and network

```
cd packtclusters-vpc
terraform workspace new prod1-vpc
terraform init
terraform apply
```

Create cluster resources

```
cd packtclusters
terraform workspace new prod1
terraform init
terraform apply
```

Configure the cluster

Retrieve the cluster kubeconfig

```
cd terraform/packtclusters

aws eks --region us-east-1 update-kubeconfig --name packtclusters-prod1
```

Create Python virtual environment

```
virtualenv ~/ansible
source ~/ansible/bin/activate
```

Install ansible and k8s modules

```
pip install 'ansible<3.0' openshift pyyaml requests 'kubernetes<12.0'
```

Execute ansible-playbook to configure the cluster

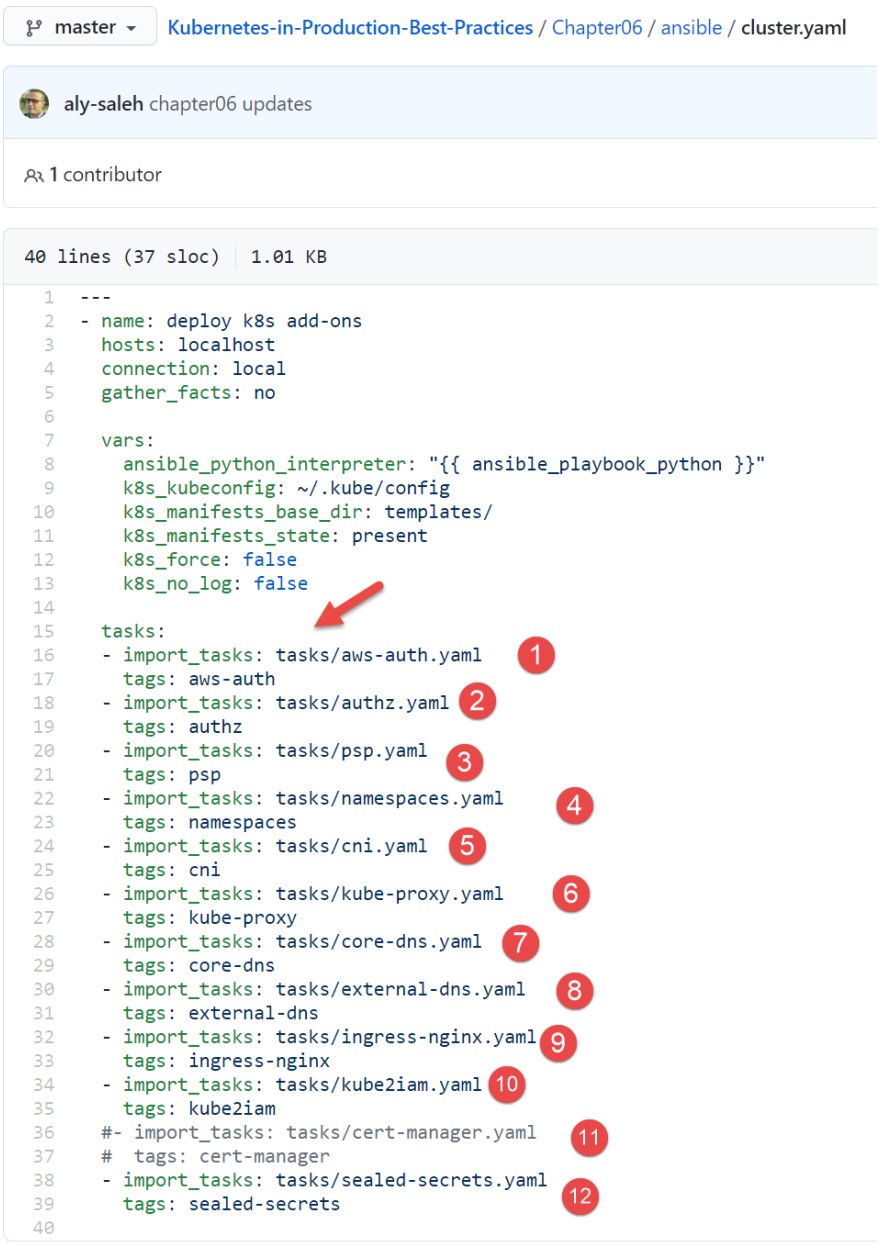
```
ansible-playbook -i \
            ../../ansible/inventories/packtclusters/ \
            -e "worker_iam_role_arn=$(terraform output worker_iam_role_arn) \
                cluster_name=$(terraform output cluster_full_name) \
                aws_default_region=$(terraform output aws_region)" \
            ../../ansible/cluster.yaml
```

Verify the cluster

```
kubectl get nodes
kubectl get namespaces
```

Chap 6 shows how to secure k8s effectively, e.g. OIDC, IAM, RBAC, secrets, certs, PSP, Netpol with Calico, runtime with Falco, CIS benchmark, Sonobuoy conformance tests, kube-scan, kube-bench, audit logging, ...etc.

e.g. https://github.com/PacktPublishing/Kubernetes-in-Production-Best-Practices/tree/master/Chapter06



master Kubernetes-in-Production-Best-Practices / Chapter06 / ansible / cluster.yaml

aly-saleh chapter06 updates

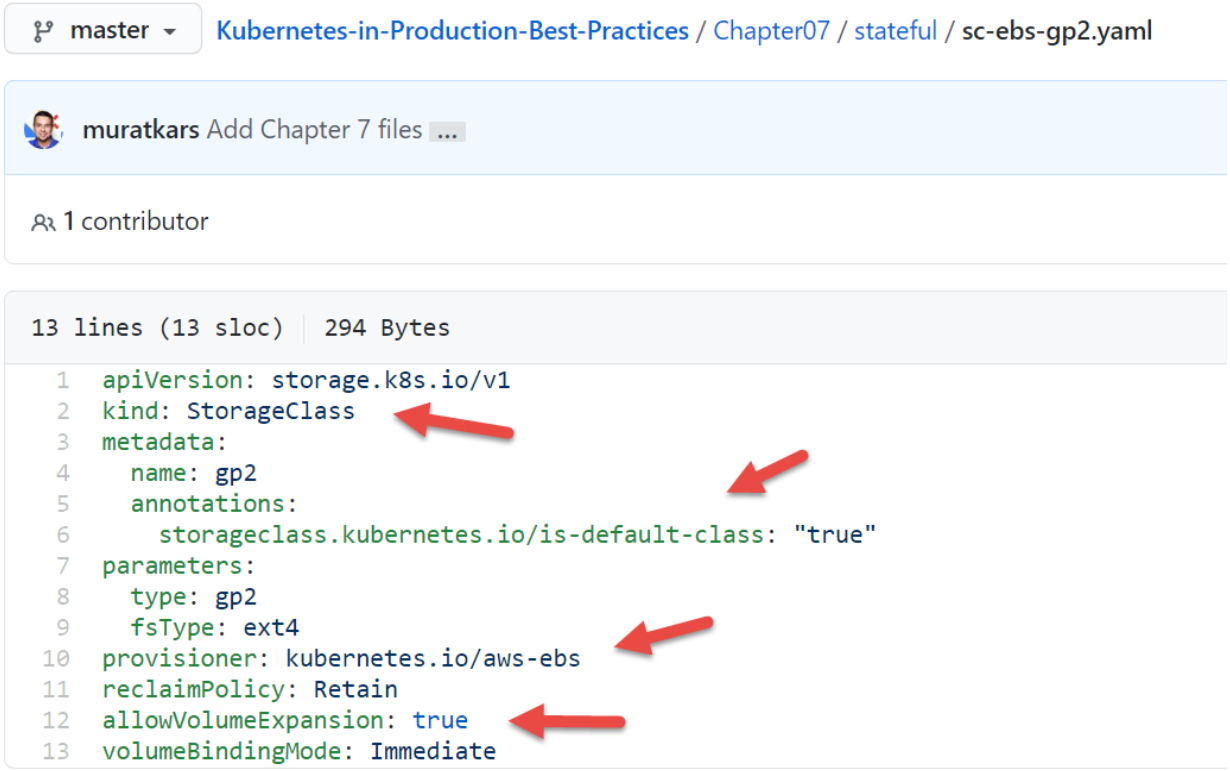
1 contributor

40 lines (37 sloc) | 1.01 KB

```
1  ---
2  - name: deploy k8s add-ons
3    hosts: localhost
4    connection: local
5    gather_facts: no
6
7    vars:
8      ansible_python_interpreter: "{{ ansible_playbook_python }}"
9      k8s_kubeconfig: ~/.kube/config
10     k8s_manifests_base_dir: templates/
11     k8s_manifests_state: present
12     k8s_force: false
13     k8s_no_log: false
14
15   tasks:
16   - import_tasks: tasks/aws-auth.yaml      1
17     tags: aws-auth
18   - import_tasks: tasks/authz.yaml         2
19     tags: authz
20   - import_tasks: tasks/psp.yaml           3
21     tags: psp
22   - import_tasks: tasks/namespaces.yaml    4
23     tags: namespaces
24   - import_tasks: tasks/cni.yaml           5
25     tags: cni
26   - import_tasks: tasks/kube-proxy.yaml    6
27     tags: kube-proxy
28   - import_tasks: tasks/core-dns.yaml      7
29     tags: core-dns
30   - import_tasks: tasks/external-dns.yaml  8
31     tags: external-dns
32   - import_tasks: tasks/ingress-nginx.yaml 9
33     tags: ingress-nginx
34   - import_tasks: tasks/kube2iam.yaml      10
35     tags: kube2iam
36   #- import_tasks: tasks/cert-manager.yaml  11
37   #  tags: cert-manager
38   - import_tasks: tasks/sealed-secrets.yaml
39     tags: sealed-secrets                   12
40
```

Chap 7 shows how to manage Storage and Stateful Applications, e.g. CSI drivers, CAS, implementation principles, challenges with stateful Applications (e.g. persistency and scalability!), demo with OpenEBS, etc.

e.g. https://github.com/PacktPublishing/Kubernetes-in-Production-Best-Practices/tree/master/Chapter07



Chap 8 shows how to deploy Seamless and Reliable Applications, e.g. build good container images, security (e.g. OVAL, PIE, PaX, ASLR, etc..), how to reduce container image size with multi-stage build, scan container images for vulnerabilities with Trivy, application deployment strategies (e.g. A/B, Blue/green, Canary, etc..), Monitoring deployments with readiness and liveness container probes, Scaling applications and achieving higher availability, ...etc.

e.g. https://github.com/PacktPublishing/Kubernetes-in-Production-Best-Practices/tree/master/Chapter08

master Kubernetes-in-Production-Best-Practices / Chapter08 / hpa / hpa-nginx.yaml

muratkars Add Chapter 8 files ...

1 contributor

13 lines (13 sloc) | 272 Bytes

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: nginx-autoscale
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx-hpa
  minReplicas: 1
  maxReplicas: 5
  targetCPUUtilizationPercentage: 50
```

master Kubernetes-in-Production-Best-Practices / Chapter08 / probes / liveness / busybox.yaml

muratkars Add Chapter 8 files ...

1 contributor

21 lines (21 sloc) | 391 Bytes

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-execaction
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/alive; sleep 30; rm -rf /tmp/alive; sleep 30
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/alive
      initialDelaySeconds: 3
      periodSeconds: 3
```

Chap 9 shows Monitoring, Logging, and Observability, e.g. Kubernetes cluster health and resource utilization metrics, Application deployment and pods resource utilization metrics, Application health and performance metrics, Prometheus stack on Kubernetes, Grafana, Installing the EFK stack on Kubernetes, ...etc.

Exploring the Kubernetes metrics

When we explored the components of container images in *Chapter 8*, *Deploying Seamless and Reliable Applications*, we also compared the monolithic and microservices architectures and learned about the function of a **container host**. When we containerize an application, our container host (**2**) needs to run a container runtime (**4**) and Kubernetes layers (**5**) on top of our OS to orchestrate scheduling of the Pod. Then our container images are (**6**) scheduled on Kubernetes nodes. During the scheduling operation, the state of the application running on these new layers needs to be probed (see *Figure 9.1*):

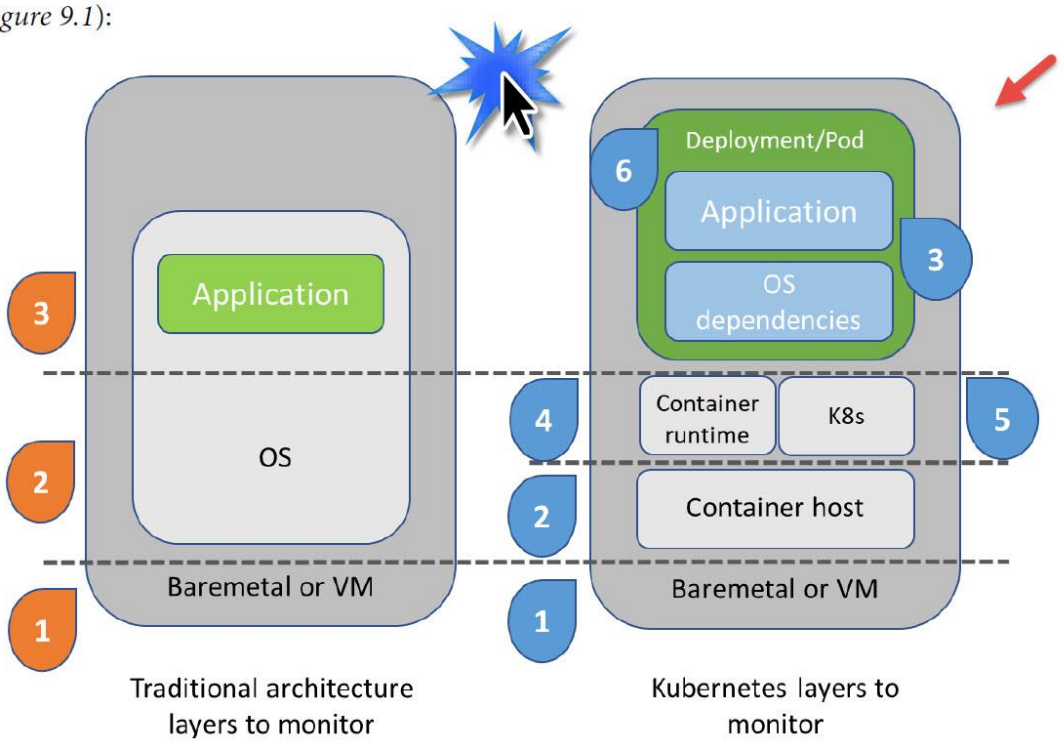
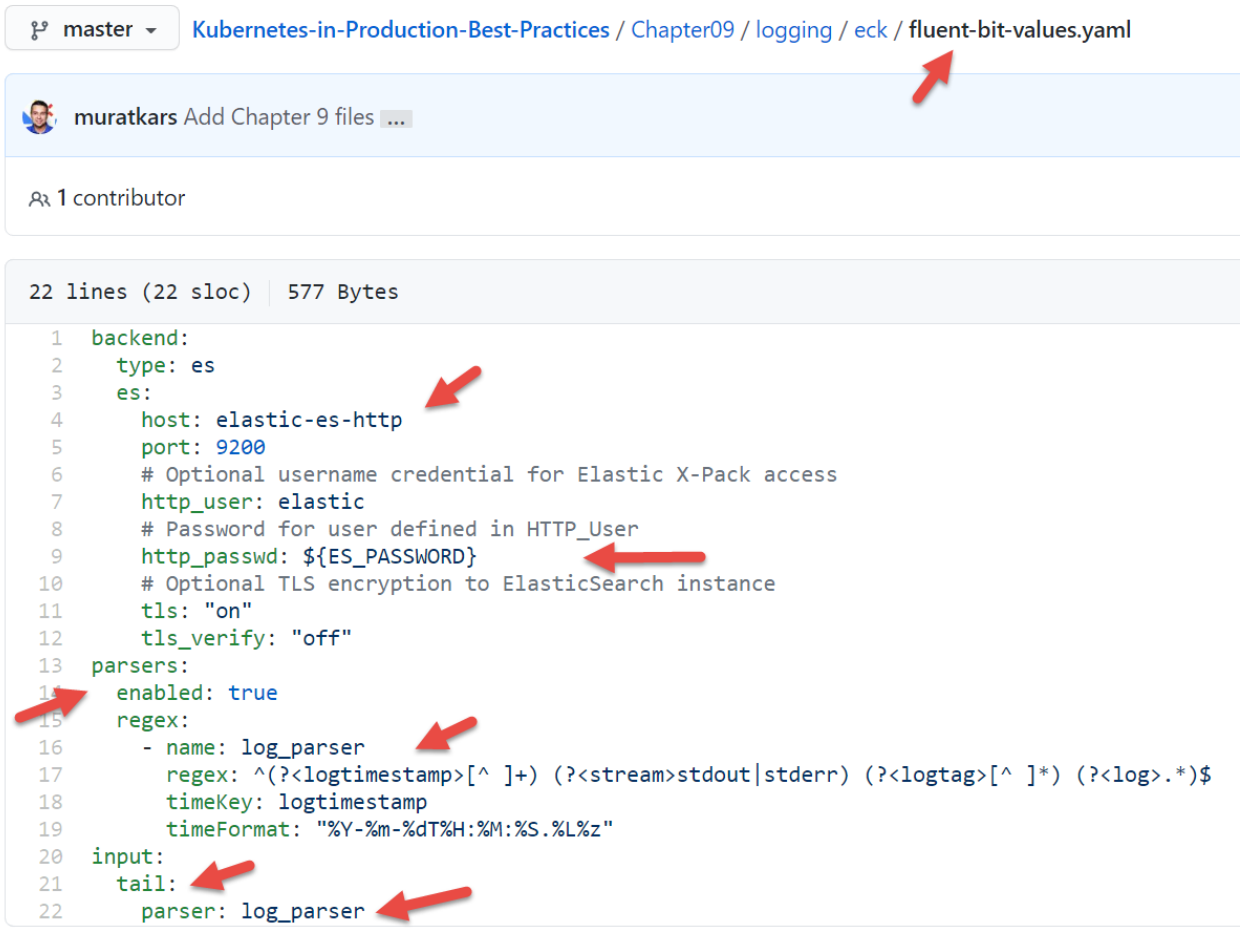


Figure 9.1 – Comparison of monolithic and microservices architecture monitoring layers

Considering all the new levels and failure points we have introduced, we can summarize the most important metrics into three categories:

1. **Kubernetes cluster health and resource utilization metrics**
2. **Application deployment and pods resource utilization metrics**
3. **Application health and performance metrics**

e.g. https://github.com/PacktPublishing/Kubernetes-in-Production-Best-Practices/tree/master/Chapter09



master Kubernetes-in-Production-Best-Practices / Chapter09 / logging / eck / fluent-bit-values.yaml

muratkars Add Chapter 9 files ...

1 contributor

22 lines (22 sloc) 577 Bytes

```
backend:
  type: es
  es:
    host: elastic-es-http
    port: 9200
    # Optional username credential for Elastic X-Pack access
    http_user: elastic
    # Password for user defined in HTTP_User
    http_passwd: ${ES_PASSWORD}
    # Optional TLS encryption to ElasticSearch instance
    tls: "on"
    tls_verify: "off"
parsers:
  enabled: true
  regex:
    - name: log_parser
      regex: ^(?<logtimestamp>[^ ]+) (?<stream>stdout|stderr) (?<logtag>[^ ]*) (?<log>.*)$
      timeKey: logtimestamp
      timeFormat: "%Y-%m-%dT%H:%M:%S.%L%z"
input:
  tail:
    parser: log_parser
```

Chap 10 shows Operating and Maintaining Efficient Kubernetes Clusters, e.g. how to perform cluster maintenance and upgrades with terraform, backup and restore with Heptio Velero, Validating cluster quality, Generating compliance reports with Sonobuoy, Managing and improving the cost of cluster resources with kubecost, ...etc.

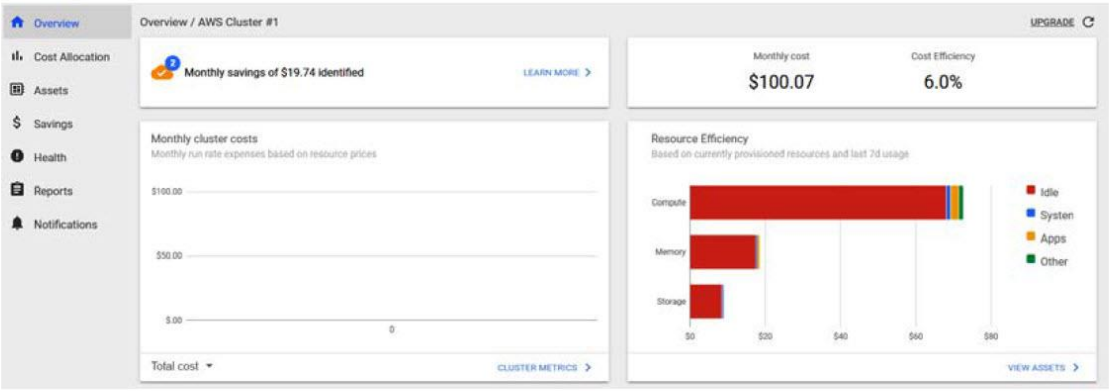


Figure 10.18 – Kubecost dashboard

9. Let's scroll down the dashboard screen to find a summary of the controller component and service allocation. At the bottom of the dashboard, we will see the health scores. A health score is an assessment of infrastructure reliability and performance risks:

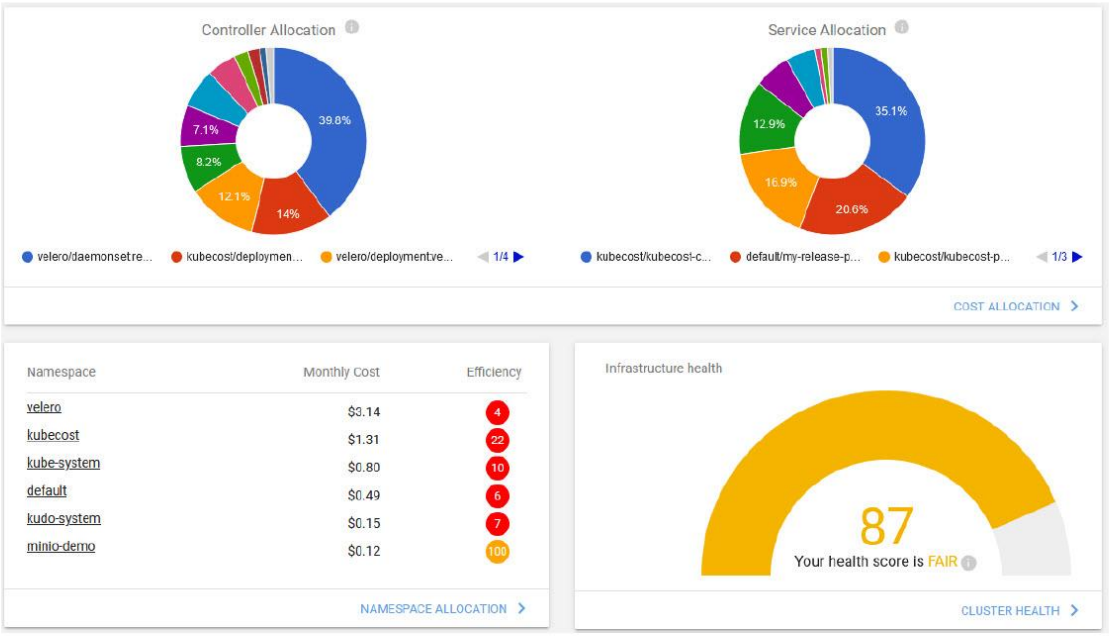


Figure 10.19 – Kubecost dashboard showing the cluster health assessment score

e.g. https://github.com/PacktPublishing/Kubernetes-in-Production-Best-Practices/blob/master/Chapter10

master Kubernetes-in-Production-Best-Practices / Chapter10 / velero / backup / deployment-minio.yaml

muratkars Add Chapter 10 files ...

1 contributor

115 lines (113 sloc) 2.19 KB

```yaml
---
apiVersion: v1
kind: Namespace
metadata:
  name: minio-demo
---
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: minio-demo
  name: minio
  labels:
    component: minio
    app: minio
spec:
  strategy:
    type: Recreate
  selector:
    matchLabels:
      component: minio
  template:
    metadata:
      labels:
        component: minio
    spec:
      volumes:
      - name: storage
        persistentVolumeClaim:
          claimName: minio-pv-claim
      - name: config
        emptyDir: {}
      containers:
      - name: minio
        image: minio/minio:latest
        imagePullPolicy: IfNotPresent
        args:
        - server
        - /storage
        - --config-dir=/config
        env:
        - name: MINIO_ACCESS_KEY
          value: "minio"
        - name: MINIO_SECRET_KEY
          value: "minio123"
        ports:
        - containerPort: 9000
        volumeMounts:
```

master Kubernetes-in-Production-Best-Practices / Chapter10 / terraform / packtclusters / terraform.tfvars

muratkars Add Chapter 10 files ...

1 contributor

18 lines (18 sloc) | 439 Bytes

```
aws_region = "us-east-1"
private_subnet_ids = [
  "subnet-xxxxxxxx",
  "subnet-xxxxxxxx",
  "subnet-xxxxxxxx",
]
public_subnet_ids = [
  "subnet-xxxxxxxx",
  "subnet-xxxxxxxx",
  "subnet-xxxxxxxx",
]
vpc_id = "vpc-xxxxxxxxxx"
clusters_name_prefix  = "packtclusters"
cluster_version       = "1.16""        #Upgrade from 1.15
workers_instance_type = "t3.medium"
workers_number_min    = 2
workers_number_max    = 5
workers_storage_size  = 10
```