

Appendix A

Solutions to the exercises

This appendix contains example solutions to the exercises from chapters 1 through 8. Keep in mind, there can be many reasonable solutions. If yours is different, focus on the outcomes.

Also note that this supplemental PDF has been created to bridge the gap between the book's in-chapter exercises and the companion GitHub repository. In the book, we guide you step by step through exercises, providing both context and reasoning alongside the solutions. Traditionally, the appendix would bring all of these solutions together in one place for easy reference. On GitHub, however, what you will mostly find are the *completed* versions of those solutions, which are fully runnable code samples, templates, and reference implementations.

The purpose of this PDF is to give you the best of both worlds. It combines the narrative context from the book with the consolidated solutions that would have appeared in the appendix, making it easier to follow along, revisit, and cross-reference your work. This way, you don't just see the end result in code, but also get the supporting explanations that connect back to the learning objectives of each chapter.

We view this as a *living resource*. Just as platforms evolve, so too will the materials that support this book. As the GitHub repository is updated, whether to fix bugs, reflect changes in best practices, or introduce enhancements, we will refresh this PDF accordingly. That means you always have a single, downloadable reference that stays aligned with the most current state of the companion code.

Use this document side by side with both the book and the GitHub repo: the book for guided learning, this PDF for consolidated solutions, and GitHub for the latest runnable code. Together, they form a complete ecosystem of resources designed to make your platform engineering journey as practical, accessible, and up-to-date as possible.

Table of Contents

Chapter 1	3
Chapter 2	4
Solution to Exercise 2.1: Stakeholder Map	4
Solution to Exercise 2.2: Feature set of an MVP engineering platform	4
Solution to Exercise 2.3: Observability-driven Design Requirements.	6
Listing 2.1 Example VirtualService definition	6
Solution to Exercise 2.4: Model the feature request for a browser-based means of integration a new git repo with the platform	7
Table 2.1 Shows the responses to the two questions that were posed above	8
Solution to Exercise 2.5: Fitness Functions	8
Decision:	8
Alternatives Considered:	9
Justification:	9
Fitness Functions:	9
Solution to Exercise 2.6: Writing a Domain-specific ADR	10
Chapter 3 Solutions	12
Solution to Exercise 3.1	12
Table 3.12 Baseline Numbers for computing the costs involved	13
Table 3.13 Computing build costs	13
Table 3.14 Savings calculations	14
Solution to Exercise 3.2	14
Solution to Exercise 3.3	15
Solution to Exercise 3.4	15
Figure 3.11 Illustration of solution to Exercise 3.4. The way to read this is from the bottom layer and each subsequent layer builds on top of the other.	15
Chapter 4 Solutions	16
Solutions to Exercise 4.1	16
Solution to Exercise 4.2	16
Solution to Exercise 4.3	17
Chapter 5 Solutions	19
Solution to Exercise 5.1: Set up an observability system for infrastructure_monitoring	19
Solution to Exercise 5.2: Correlating data with a demo application	20
Solution to 5.3: Case Study: Evaluate the TCO of continuing to support your observability system locally	21
Solution to Exercise 5.4: Use Prometheus data to create SLOs	23
Chapter 6 Solutions	24
Solutions to Exercise 6.1: Assess a platform developer tool according to the software selection criteria	24
Solutions to Exercise 6.2: Create and run a configuration test against a built-in AWS role	24
Solutions to Exercise 6.3: Perform static analysis of terraform code	24
Solutions to Exercise 6.4: Implement static analysis through commit hooks	25
Solutions to Exercise 6.5: Experiment with Self-Hosted Runners	26
Solutions to Exercise 6.6: Bootstrap our nonproduction and production AWS	

accounts with the initial service accounts and pipeline role	28
Solutions to Exercise 6.7: Create the CI and development test pipeline for IAM service accounts and roles	35
Solutions to Exercise 6.8: Add a credential rotation step to the first stage pipeline	39
Solutions to Exercise 6.9: Create the release pipeline for IAM service accounts and roles	41
Chapter 7 Solutions	43
Solutions to Exercise 7.1: Create a release pipeline for hosted zone and zone delegation	43
Solutions to Exercise 7.2: Create a release pipeline for a role-based network	51
Solutions to Exercise 7.3: Create a build and release pipeline for the control plane base	61
Chapter 8 Solutions	69
Solution to Exercise 8.2 Run Trivy scan on metrics-server chart and create a values.yaml to correct the findings	69
Solution to Exercise 8.3: Create the set-environment command for our control-plane-services pipeline.	69
Exercise 8.4 Add the necessary steps to the control plane services deployment job for our pipeline	70
Solution to Exercise 8.5: Add the release stage to our control-plane-base pipeline	74
Solution to Exercise 8.6: Complete the definition of our pipeline trigger filters in these anchors.	75
Exercise 8.7 Complete the steps we will need for the set-environment command in our extensions pipeline.	76
Exercise 8.8: Create AwSeps test to confirm roles	76
Exercise 8.9 Create deployment scripts and values files for cert-manager and external-dns deployments.	76
Exercise 8.10 Create state tests using Bats to confirm the extensions report a healthy status.	81
Exercise 8.11 Create a script that uses the Httpbin deployment to test our extensions.	81
Exercise 8.12 Add the release workflow to the aws-control-plane-extensions pipeline.	83

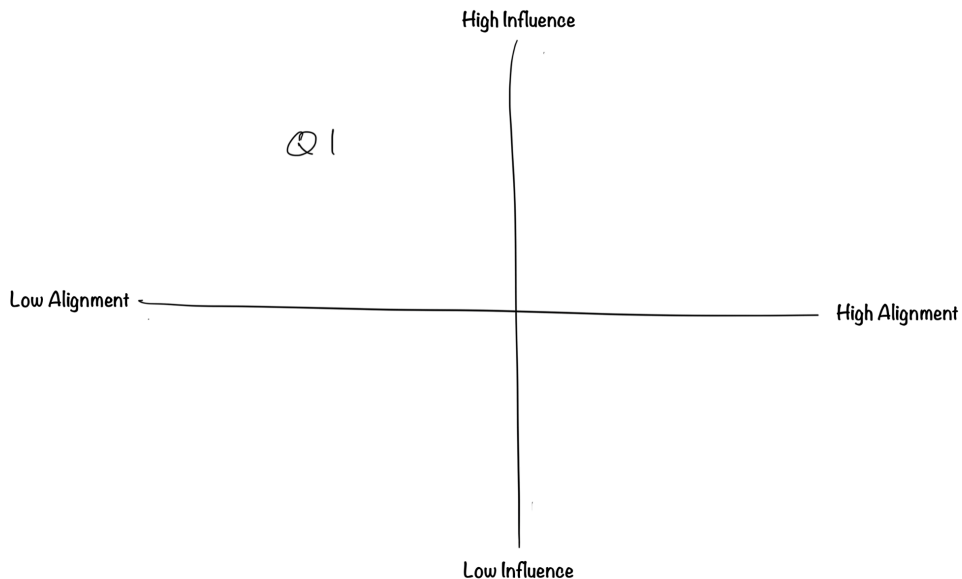
Chapter 1

No exercises.

Chapter 2

Solution to Exercise 2.1: Stakeholder Map

Example Stakeholder map and alignment issues:



Role: Director - Office of Program Management

Level of Influence: High. OPM defines organization approval processes.

Alignment: Low.

Issue: Every deployment requires a very time-consuming human checklist and review approval process. The PMO states the objectives as broad internal visibility into changes and assurance of PO approval. The information required originates in Jira and with Product Owners.

Possible alternatives: Create automation to merge the info from Jira into the Change Calendar (visibility tooling) and create admission controllers that confirm commit SHA with PO approval in Jira. This would be a rapid and automated method to perform the same tasks currently performed in the approval meeting.

Solution to Exercise 2.2: Feature set of an MVP engineering platform

A proposed list of capabilities for an engineering platform's initial, minimum valuable product definition.

1. **Identity.** Programmatic access to an authoritative source for authentication and authorization. It must be able to effectively integrate with the PE product roadmap for creating team-level tenancy and resilient management of technology and tool integrations.

Why this is needed: This is the most basic access and security requirement. Any solution that is not sufficiently programmatically manageable by the Platform team to create a product experience will impact every feature in the roadmap to some degree. Identity is a dependency for every other capability of the platform. Getting this right first will create more flexibility in prioritizing other capabilities than any other architectural decision.

2. **General Developer Tools:** source version control, secrets management, artifact stores that support OCI images and related metadata, CI and CD pipeline orchestration.

Why this is needed: These tools are likely already available in some fashion. Where the current tool is a good long-term choice for integration with the platform, the timing of that integration may be flexible. Even if the current solution is inadequate, a new tool or integration may not technically be needed at the start if users can continue with their existing solution without modification. In either case, programmatic access (nearly always at administrator level) to these tools is a minimal requirement for the Platform product team to provide a successful self-service experience.

3. **Public or private cloud.** Regardless of the infrastructure capabilities that make up the MVP, we must begin with a target location where the platform engineers have programmatic access to automate the lifecycle of infrastructure resources as a self-serve experience for users.

Why this is needed: Infrastructure is obviously a core capability. It's not just one thing of course but many. Let's use networking as an example. The underlying network is frequently the biggest source of friction at the start. If an architecture that enables the Platform team to self-manage networking and IP needs can not be agreed upon, then minimally provide enough capacity and connectivity to cover the expected scale for the first 18-24 months after users are on the platform. Shortcuts here become ticking time bombs for later problems, even outages. The next two items probably come from here.

4. **Dynamic compute.** The single most frequent behavior for the customers of the platform will be deploying and interacting with the software they are building. As most new strategic custom software is non-mobile and follows a stateless or distributed service architecture, the most needed type of compute will be Kubernetes orchestration.

Why this is needed: Kubernetes also meets our MVP need for a control plane to support the service interface to extended infrastructure capabilities. You don't have to assume containers are the most needed compute; Maybe your internal customer pool has a broader need. Regardless, you still need a control plane and building one from scratch is expensive and risky.

5. **Ingress and traffic routing.** DNS and load balancing of some form is required. It is possible that there is a sufficiently large pool of users building software that is only accessed from other internal systems that the MVP does not need to solve for external access. Given the breadth and maturity of dynamic routing capabilities a service mesh provides a mesh should be included in the MVP even if use is limited to ingress. If external access is necessary, this covers things like firewalls and other required edge capabilities.

Why this is needed: Software that can't be accessed provides no value.

6. **Observability tools.** Aggregated logs, metrics, events, and tracing from all of the MVP technologies, along with a self-serve means of searching and filtering the aggregated data and a software-defined means of managing dashboards, monitoring, and alerting.

Why this is needed: We have to be assessing the production experience and value, and of course we can't be in production without effective observability.

7. **Software security and provenance.** This refers to the security activities of the CI and CD pipelines or other security services directed at the software platform users deploy. If the platform won't initially provide language starter kits that include pipelines because existing pipelines can be used to start, and those pipelines include the basics, such as vulnerability scans, then you could assess leaving this out of the MVP. Given the potentially bankrupting impact of security vulnerabilities, the platform MVP should at least include provenance. The secure configuration of the platform's components is a requirement of each component implementation, whether the technology is included in the MVP or not.

Solution to Exercise 2.3: Observability-driven Design Requirements.

Practice defining the observability data needed for a feature using ODD principles, listed in the previous section, by considering the following feature of an engineering platform:

The platform provides a number of predefined ingress domains for Epetech API developers. For example, *dev.api.epetech.io* is an ingress url reserved for all teams initial testing environments and a dedicated gateway has been defined that receives all traffic to this domain. Teams specify the *dev* gateway and the path for the *dev* instance of their API among the values passed during a deployment. Their Helm chart will include a VirtualService resource that includes these values, directing the service mesh to send such traffic to their API. Assuming the *customer* domain team deploys a *profile* service, their VirtualService definition would define and direct traffic as follows:

Listing 2.1 Example VirtualService definition

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: profile
  namespace: customer-dev
spec:
  hosts:
    - "dev.api.epetech.io"
  gateways:
    - dev-api-epetech-io-gateway
  http:
    - name: profile-route
      match:
        - uri:
            prefix: /customers/profile
      route:
        - destination:
            host: profile.customer-dev.svc.cluster.local
            port:
              number: 8000
```

Results in traffic direction of:

`https://dev.api.epetech.io/customers/profile => profile.customer-dev.svc.cluster.local`

As the platform engineering team responsible for maintaining this capability, what kinds of observability data will we need to be able to effectively operate, upgrade, and maintain this feature?

- How do I know with this is working correctly?
- What metrics, logs, traces, or event info would I need to diagnose success or failure?
- What kinds of behaviors do I want to be alerted about?

Solution to Exercise 2.4: Model the feature request for a browser-based means of integration a new git repo with the platform

This request is targeted at reducing the amount of time it takes folks other than the developer members of a development team to create or add a new github repo to our platform automation.

Here is the kind of information we could use to model and assess the impact. You may think of additional ways to measure.

- How many new repositories are created and integrated with the platform on a weekly, monthly, and annual basis?
- What is the breakdown by role and team of the people who perform this task across all platform customers?
- By role and team, how long on average does it take a person to perform this task?
- How could I create an experiment to calculate how long it would take on average to perform the same task via a browser interface?

Assuming we think the above data would be sufficient to calculate an average time savings, how would we gather this info?

The first two we could pull from our API logs, with potential some change needed if the logs don't currently collect all the info.

The third bullet isn't something we can track in automation. Maybe we could extrapolate some data based on the time between when a person hits a specific page in the platform documentation about integrating repos and subsequently triggers the API call, but how often would that even reflect the way people really went about the task? Surveying the platform users is probably the only option.

Question 1 : Do you create and integrate Git repositories with the Engineering Platform?

1. Yes
2. Rarely, I have but not normally part of my job
3. No
4. What is Git?

If you answered 1 or 2 respond to the rest of the questions, if not then youre done! ;)

Question 2 : Which answer best describes how long it typically takes you to create and integrate a Git repositories with the Engineering Platform?

1. Just a couple of minutes
2. 10 to 15 minutes
3. More than a half-hour
4. Couple of hours
5. It's the only thing I accomplish for the day

Question 3 : In your day-to-day role, do you:

1. Routinely use git and other CLIs and prefer such tools
2. Use CLIs and UIs, but don't have a preference one way or another
3. Use CLIs but prefer a browser or app option if available

Question 4 : What is your role on the team? _____

We could send this out as a simple Slackbot 2-part question, making it as easy for people to answer it and return to their jobs. We don't want to inundate our users with lengthy or wordy surveys; it's best to model these new capabilities with questions that are most likely to get as many responses as possible. Keeping it short ensures we'll at least get a majority (50% or higher) response rate.

Our results come in, and we get:

50 overall responses

Table 2.1 Shows the responses to the two questions that were posed above

	Q1	Q2
Responses	35 CLIs, 30 APIs, 25 GUIs 10 All of the Above, 5 other	30 CLIs, 20 GUIs

Surprisingly, there are more users than we expected who want to use a GUI to deploy to the platform! Had we just used our assumptions and not modeled and validated them, we never would have developed this feature, leaving 40% of our users using a set of features they would prefer to do in a different interface.

Solution to Exercise 2.5: Fitness Functions

PE Tech is building an Engineering Platform on AWS. However, with the pending merger with AllTech, which operates entirely on Google Cloud, the platform must be adaptable to different cloud environments. This necessitates designing cloud-agnostic APIs that can work across various cloud providers while also adhering to test-driven development (TDD) principles for reliability and quality. Observability and monitoring are crucial for ensuring platform stability and for guiding decisions based on data.

Decision:

1. **Cloud-Agnostic APIs:**
 - Implement platform APIs to isolate cloud-specific features within dedicated interfaces, keeping the main service logic cloud-agnostic.
 - Write fitness functions that ensure APIs follow this pattern.
2. **Test-Driven Development:**
 - Adopt TDD for all platform custom software.
 - Write fitness functions that enforce tests for new service layers.
3. **Monitoring and Observability:**
 - Ensure each new API feature includes appropriate monitoring and

- observability tools.
- Write fitness functions that enforce this requirement.

Alternatives Considered:

- **Cloud-Specific APIs:** Building APIs specific to AWS would simplify implementation but reduce flexibility.
- **No TDD Enforcement:** Relying on manual code reviews for quality assurance, but risking missed test coverage.
- **Ad-hoc Monitoring:** Allowing teams to decide monitoring independently, risking inconsistencies.

Justification:

- **Future-Proofing:** Cloud-agnostic APIs ensure easier migration and integration with different cloud providers.
- **Reliability:** TDD ensures code reliability and catches defects early, improving software quality.
- **Proactive Monitoring:** Consistent observability ensures platform stability and allows data-driven decision-making.

Fitness Functions:

Cloud-Agnostic APIs:

- Ensure platform APIs are implemented in a cloud-agnostic way by isolating cloud-specific features:

```
layeredArchitecture()
    .layer("CloudSpecific").definedBy("..cloudspecific..")
    .layer("Service").definedBy("..service..")

    .assertLayer("Service").mayOnlyBeAccessedByLayer("Handler")
    .assertLayer("CloudSpecific").mayOnlyBeAccessedByLayer("Service")
```

Test-Driven Development:

- Fail if a new service layer is checked in without associated tests:

```
serviceLayerWithoutTests := // Logic to find services without tests
assert serviceLayerWithoutTests.empty()
```

Monitoring and Observability:

- Fail if a new API feature lacks monitoring:

```
newAPIFeaturesWithoutMonitors := // Logic to find features without monitors
assert newAPIFeaturesWithoutMonitors.empty()
```

Solution to Exercise 2.6: Writing a Domain-specific ADR

Exercise 2.6: Writing a Domain-specific ADR

Suppose the security team at Epotech establishes a new requirement that all the custom software we deploy must provide an associated software bill of materials. For the first year, developers can comply with the policy by providing materials information covering the contents of the software at the time of release, leaving information about the build process and responsible parties for a later stage.

Using what we learned so far, write a top-level ADR describing the primary components of automation that could achieve this outcome. Keep in mind the domain boundaries that will enable the platform engineers, developers, and stakeholders to get to this outcome without introducing breaking product changes.

Architectural structure for the Software Bill of Materials capabilities of the Epotech Engineering platform

Required outcome:

1. Custom software running on the Epotech engineering platform must have an associated bill of materials that documents all external or 3rd party libraries, data, or any other content not the result of Epotech custom software creation. This list of resources must include any known vulnerabilities at the time of release. This SBOM policy is not a statement of the acceptable or non-acceptable of any vulnerability. Organization policy related to acceptable risk is managed independently.
2. Users of the platform should be able to successfully meet or assess this requirement through entirely self-serve means.
3. Users deploying software to the platform should be alerted to non-compliance at the earliest responsible moment and prevented from deploying if the requirement is not met.

Based on these required outcomes the following architectural attributes are necessary. Existing architectures, capabilities, or process that are believed able to achieve these outcomes are listed and should be followed unless it is discovered they are ineffective.

1. Developers using the platform must have a self-serve means of generating a compliant SBOM.

New capability: SBOM Generation. Based on a review of available third-party options, we will initial assess the results from using Syft (link) to generate a contents SBOM in the following format (type) as the proposed standard going forward.

Based on existing capabilities: A new orb will be created that can be integrated into the Build stage of the platform customer pipelines. Existing starter kits will be updated to have this functionality already integrated.

When integrated the orb will generate an SBOM for each build SHA and upload the SBOM to the image registry of the same image. When completed the orb will also call a custom API that will compare the build log of the image with the SBOM to confirm package accuracy. Upon successful comparison, the API will generate a matching event log, searchable within the observability platform tooling.

2. Users of the platform should be able to successfully assess this requirement through entirely self-serve means.

Existing Capability: The platform 'runtime versions' dashboard already provides a self-serve means for users and stakeholders to see the current versions of all software running in production or nonproduction platform environments, with links to the image registries that will now also contain the SBOM.

3. Users deploying software to the platform should be alerted to non-compliance at the earliest responsible moment and prevented from deploying if the requirement is not met.

Existing Architecture: The platform implements a point-of-change compliance architecture whereby all compliance controls run as validating admission controllers running on the Kubernetes control plane.

New capability: A new admission controller to be created that on any deployment will first validate the deployment object SHA as having a matching, successful SBOM validate event log (from #1). As a result, developers will be alerted to a noncompliant build from the very first environment deployment attempt (their dev environment).

4. A grace period of 90 days will be provided, where the admission controller will not fail but only provide Notice, in order for platform customers to have time to adopt the SBOM orb into their existing build pipelines.

Chapter 3 Solutions

Solution to Exercise 3.1

While there are no right or wrong answers for this exercise you can use the following responses as a guideline to compare against what you generated.

1. **Business Context:** PETech is building financial service products on the public cloud for business-to-consumer clients. The consumers log into the PETech portal and integrate their technology systems to generate appropriate invoices and other payment workflows. A loosely coupled architecture is used for the overall software system and implemented using primarily containerized microservices. PEech wants to improve the effectiveness of its engineering team in developing and delivering these containerized microservices faster, in better quality, and with minimal additional training for its financial software developers.
2. **Platform Capability:** Using the Kubernetes-based container orchestration is the industry standard recommended approach for every developer and the development team. However, the learning curve to install, operate, troubleshoot, and optimize the clusters for the k8s applications is very high. The capability that we need to build is a layer that resides between Kubernetes and the developers, helping them focus on their applications by having knowledge of the concepts and the approach of containerizing the applications and orchestrating them, but abstracts out the repeatable set of steps associated with it.
3. **Assumptions and Exceptions:** PETech's platform engineering team has decided that there will only be one option to build this technical capability. An architectural decision tree is in place to recommend the appropriate implementation for a given microservice across the spectrum of choices, such as Kubernetes, Serverless, Virtual machines, and others.
4. **Measures of Success:** This effort's success is measured by the number of workloads deployed across the organization and the number of teams adopting the platform for their container orchestration.
5. **Costs of building the platform:** The cost of building the platform includes three distinct components: Developer costs, infrastructure costs, and any related support and onboarding costs
6. **Savings goals:** This initiative will have clear savings goals. These include eliminating the costs of building the capabilities by individual teams, replicated and infrastructure costs, downtime as the teams repeatedly wait for the availability of capabilities and opportunity costs.
7. **Identify a clear timeline:** The organization would like to understand the return on investment for the platform capabilities within the first year, twelve months from the start of the effort.
8. **Metrics to track:** We would like to follow the value-to-cost ratio (VCR) as it becomes the driving factor for investing in capability building.

*Value-to-cost ratio = value generated *100 / cost of generating the value*

9. **Computations:**
Before you start your build costs and savings calculation, establish specific baseline numbers, as shown below.

	Business Domain Product
Fully loaded developer cost/hour	\$100
Average hours worked / month	160
Number of months to develop the capability	3
Adoption period to measure value	9
Average number of developers/application	6

Table 3.12 Baseline Numbers for computing the costs involved

Calculate each of the three components to compute the build costs, as shown in step 5 above. The assumption is that it takes three months for a team of four developers to build this capability.

		Total Costs
Developer Costs		
Number of developers	4	\$192,000
Infrastructure Costs		
Average cloud costs/month	\$1000	\$3000
Support & Onboarding		
Number of support engineers	4	\$576,000
Average cloud costs/month	\$1000	\$9000
Total Build Costs		\$780,000

Table 3.13 Computing build costs

To compute the savings, we have to consider the adoption of the capability by a certain number of applications every month from month four to month twelve based on the assumptions made in step seven.

		Total Costs
--	--	--------------------

Adoption starting month 4		
Average applications adopted/month	10	
Hours saved/developer/week	5	
Total Savings		\$1,080,000

Table 3.14 Savings calculations

10. **Metrics calculation:** Now that we have the cost of building the platform capability and the potential value generated by this capability, you can calculate the value-to-cost ratio.

$$\text{Value-to-cost ratio} = \$1,080,000 * 100 / \$780,00 = 1.38$$

The solution provided above is a simplified version of calculating the value of a platform product. Depending on other costs and savings goals, your answer can be as detailed as your decision-makers require.

Solution to Exercise 3.2

The solution to this exercise will be a table with ten entries similar to the example given below.

Note that the column 3 is a high level overview of how to solve the issue and would typically require more detailed analysis to be accurate. Same thing goes for the effort in person weeks. What you are providing is a high level swag that will help set the context. In most cases you might want to provide a T-shirt size estimate for this effort.

Cognitive load issue	Platform Capability that can fix the issue	How will it solve the issue?	Effort in person weeks
Developers often have to switch between multiple tools and environments to manage their workflows, such as coding, testing, debugging, and deploying. This frequent context switching can significantly increase cognitive load, leading to reduced productivity and increased risk of errors.	<p>An IDE that integrates all necessary tools and environments into a single interface. Key capabilities would include:</p> <ul style="list-style-type: none"> • Source control management • Continuous integration and deployment pipelines • Built-in code analysis and testing tools • Real-time collaboration 	<p>The solution will solve the issue across four axes.</p> <ol style="list-style-type: none"> 1. Unified Interface 2. Seamless Integration 3. Automation Features 4. Collaboration and Remote Work 	<p>Research:2 Core Features:6 Testing:4 Deployment /Enablement:2</p>

	features		
--	----------	--	--

Solution to Exercise 3.3 .

The questions posed in Exercise 3.3 require subjective opinions from you. Once you develop the response, identify a peer of yours and setup some time to explain the scenario. Once you explain the scenario, describe your approach to them and brainstorm ideas on what they would have done differently from how you have suggested the solution.

Solution to Exercise 3.4 .

Technical outcome sought: A single pane of glass to see all issues related to my microservice around application scaling, functional errors, infrastructure scaling, resource constraints, service downtime, security vulnerabilities, and performance problems.
Platform capability required: An Observability platform that brings together all aspects of the SDLC for my microservice in a coherent visualization that can help me solve the most critical problems expeditiously
Business driver desired: Reduction in the MTTD/MTTR the customer's visible issue, often leading to self-healing approaches.
Business outcome achieved: Improved customer experience leading to higher customer satisfaction score (CSAT)

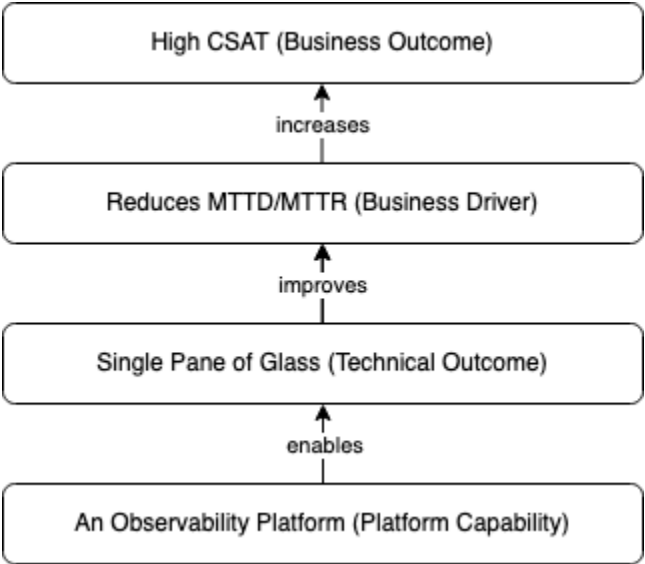


Figure 3.11 Illustration of solution to Exercise 3.4. The way to read this is from the bottom layer and each subsequent layer builds on top of the other.

Chapter 4 Solutions

Solutions to Exercise 4.1

```
package test

default deny := false

deny if code_coverage_lt_required
deny if multiple_replicas_not_defined
deny if container_resource_limits_not_defined
deny if container_resource_requests_not_defined

code_coverage_lt_required if {
    print("sha to check", input.request.object.metadata.sha)
    print("results returned from data source",
data.code_quality_attributes[input.request.object.metadata.sha].coverage)
    data.code_quality_attributes[input.request.object.metadata.sha].coverage < 80
}

multiple_replicas_not_defined if {
    print("replicas defined", input.request.object.spec.replicas)
    input.request.object.spec.replicas < 2
}

container_resource_limits_not_defined if {
    some container in input.request.object.spec.template.spec.containers
    print("container resource definition", container.resources)
    not container.resources.limits
}

container_resource_requests_not_defined if {
    some container in input.request.object.spec.template.spec.containers
    print("container resource definition", container.resources)
    not container.resources.requests
}
```

Solution to Exercise 4.2

```
#!/usr/bin/env bash
set -eo pipefail

# Script expects registry and image reference as parameters
registry="$1"
image="$2"

# Validate that the cosign generated key files are available in the current directly,
# and that the key passphrase is set in the local environment
if [ ! -f "cosign.key" ]; then
    echo "signing key not available; not able to sign image."
    exit 1
fi

if [ ! -f "cosign.pub" ]; then
    echo "verification key not available; not able to validate signing process."
    exit 1
fi

if [ ! "${COSIGN_PASSWORD}" ]; then
    echo "signing key passphrase is not available; not able to sign image."
    exit 1
fi
```

```

fi

# Validate image registry credentials and access
if [ ! "${DOCKER_LOGIN}" ]; then
    echo "registry access username is not set, will not be able to push image."
    exit 1
fi

if [ ! "${DOCKER_PASSWORD}" ]; then
    echo "registry access password is not set, will not be able to push image."
    exit 1
fi

echo "${DOCKER_PASSWORD}" | docker login -u "${DOCKER_LOGIN}" --password-stdin
"${registry}"

# get image manifest
docker image inspect --format='{{index .RepoDigests 0}}' "${registry}/${image}" >
manifestid
# sign image and store signature
cosign sign --key cosign.key $(cat manifestid) -y
# verify signature
cosign verify --key cosign.pub "${registry}/${image}"

```

Solution to Exercise 4.3

```

package test

default deny := false

deny if code_coverage_lt_required
deny if multiple_replicas_not_defined
deny if container_resource_limits_not_defined
deny if container_resource_requests_not_defined
deny if container_image_from_organization_repository_not_defined

code_coverage_lt_required if {
    print("sha to check", input.request.object.metadata.sha)
    print("results returned from data source",
    data.code_quality_attributes[input.request.object.metadata.sha].coverage)
    data.code_quality_attributes[input.request.object.metadata.sha].coverage < 80
}

multiple_replicas_not_defined if {
    print("replicas defined", input.request.object.spec.replicas)
    input.request.object.spec.replicas < 2
}

container_resource_limits_not_defined if {
    some container in input.request.object.spec.template.spec.containers
    print("container resource definition", container.resources)
    not container.resources.limits
}

container_resource_requests_not_defined if {
    some container in input.request.object.spec.template.spec.containers
    print("container resource definition", container.resources)
    not container.resources.requests
}

container_image_from_organization_repository_not_defined if {

```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://livebook.manning.com/#!/book/book-title/discussion>

```
    some container in input.request.object.spec.template.spec.containers
    print("container image definition", container.image)
    not regex.match(`^ghcr.io/epetech.*`, container.image)
}
```

Chapter 5 Solutions

Solution to Exercise 5.1: Set up an observability system for infrastructure_monitoring

Identify your observability goals: Think about what is needed for you to be sure the platform is operating as it should be. This might include things like:

- Uptime
- Network throughput
- Cluster control plane errors
- CPU/Memory/DiskIO usage

Also, think about how you will know your platform is being used and (hopefully) show growth over time. This might include tools other than your kubernetes cluster as well:

- Number of team namespaces
- Number of deployments or pods
- Ingress points or routes and services published to the cluster
- Number and frequency of tickets submitted to the platform team and time to resolution

Open-source components: For this example, we will use the Grafana stack (Prometheus, Loki, Tempo, Grafana), but there are a plethora of others with more being published over time.

Install on a cluster: The bash script in the companion repository ``exercise_solutions/chapter5/Exercise-5.1/install_k8s_with_observability.sh`` can be run on a local (Linux-based) computer to install a Kubernetes cluster using Kind, and then deploy Grafana LGT with Prometheus using the public docker image ``grafana/otel-lgtm``.

Explore captured metrics: Now without any extra configuration, this stack deploys with Prometheus capturing cluster metrics by default and now you can begin to explore them. In fact, this is the experience your platform customers should expect when deploying applications!

Sample Prometheus Queries: Open the Prometheus UX at `http://localhost:9090` and use the query bar to explore a few metrics

- Uptime of different cluster components can be seen with ``time() - process_start_time_seconds``
- Uptime of cluster nodes can be queried with ``node_time_seconds - node_boot_time_seconds``

(Optional) Trace Ingestion: While we have not yet instrumented applications, we should include support for Open Telemetry (OTEL) trace ingestion, as it will be very useful to all teams that adopt it.

If you look at the prometheus-values we set, notice the ``extra_scrape_configs`` settings. This enables prometheus to scrape some configs from a cluster deployed by Kind that it would normally get by default in a cluster like EKS, namely ``kube-apiserver``. However, prometheus is set up that any pod deployed by your usage could automatically be scraped for metrics by adding the following annotation to the pod spec:

annotations:

```
prometheus.io/scrape: "true"
prometheus.io/path: "/metrics"
prometheus.io/port: "8080"
```

How can we do the same things for OTEL metrics? For an open-source option, take a look at the Grafana Agent install to enable this kind of self-service!

Solution to Exercise 5.2: Correlating data with a demo application

Identify a demo application: For this solution, we can deploy the publicly available demo "OpenTelemetry Demo" published by the OpenTelemetry Project at <https://opentelemetry.io/docs/demo/kubernetes-deployment/>. The OpenTelemetry demo app is a microservices-based web application that will allow us to generate metrics, logs and traces that can be ingested by our observability stack.

Start the observability stack: If you followed the example solution to exercise 5.1 in the companion repository, ensure the local cluster you installed is running along with all services that were deployed as part of the Grafana LGT stack, Prometheus, and the optional Grafana Alloy install. This will allow you to pick up all the telemetry data being sent by the OTel demo once it is properly configured.

Send data from the demo application: First, we need to configure the otel demo application to ensure that the metrics, logs, and traces are being collected by our deployed agent. A script that can do this is in the Exercise 5.2 folder of the solutions companion repository.

Note that as a best practice deployed services should provide an endpoint for any custom generated metrics, and we can be sure they're sent to the right endpoint with pod annotations. In a true engineering platform, we could simply document this and expect the engineering team to follow the procedure. For more advanced automation we could even patch deployments with the information when they're deployed as part of the platform!

If you followed the example solution to exercise 5.1 to deploy the observability stack no changes to that are needed to pick up telemetry from the new application deployment. This is exactly what we want in a self-service platform!

A script that can deploy and configure the otel demo in a cluster is available at `install_and_configure_demo.sh` in the exercise 5.2 folder. This script will:

- Install the otel demo application on the cluster
- Configure the demo to emit metrics to the installed Prometheus instance
- Configure otel traces to be sent to Tempo
- Logs are automatically scraped from any namespace, so there's nothing needed

If you'd like to check out the application, you can port-forward the frontend and explore, or even generate simulated load.

First, expose the frontend on your local machine

```
kubect! --namespace demo port-forward svc/frontend-proxy 8080:8080
```

Now, you can see it in your browser at <http://localhost:8080>. If you would like to simulate load on the application, access it at <http://localhost:8080/loadgen>

Check and make sure that the data is showing up on an observability dashboard:

Now we can go into our sample Grafana installation and create a dashboard to verify that the application telemetry is being collected. First, either run some load through the application or explore the app and purchase some merchandise so we have good telemetry data. Now we can create a dashboard with 3 panels, one for a metric, log and traces.

- Open the Grafana Interface at <http://localhost:3000>
- Login with username: admin and password: Epetech
- Click on Dashboards -> Create Dashboard -> Add Visualization
 - Note that data sources should already be configured if you used the solution from Ex 5.1.

Metrics: First, we can add a Metrics panel from Prometheus. We can use this example to show an interesting business metric. What percentage of requests for TARGETED ads result in the display of a TARGETED ad? (HINT: if you explore the application and purchase a few things, or run the loadgen, it should not be 100%)

- Select the Prometheus datasource
- For your query, use the following:

```
100*(sum by (job)
(rate(app_ads_ad_requests_total{app_ads_ad_response_type="TARGETED"}[5m]))
/ sum by (job)
(rate(app_ads_ad_requests_total{app_ads_ad_request_type="TARGETED"}[5m])))
```

- Select Run Query and then Apply in the top right
- You should have your first Metrics Panel!

Logs: Now, we can explore logs from Loki. For example, we may want to see logs from the recommender to see if it correlates with targeted adds.

- Above the panel we created, click Add -> Visualization
- Change the Datasource to Loki and use the code entry
- Use the query `{namespace="demo", app="recommendation"}`
 - Note you could also use the Label filters to explore
- Run Query (You may need to switch the Panel to Table View)
- Apply to your dashboard and save
- We now have a panel showing logs from our demo app of cart activity!

Traces: Last, we can explore traces. Trace the GetAds requests to see why targeted ads are not returned if the ratio gets too high.

- Above the panel we created, click Add -> Visualization
- Change the Datasource to Tempo
- Use the following TraceQL to examine "Add to Cart" actions with the following query:

```
{resource.service.name="ad" && name="oteldemo.AdService/GetAds"}
```

- Ensure you're in Table view and click the "Refresh" button above the query.
 - Note: You can also use the "Search" query type to explore other traces
- Apply to your dashboard and save
- We now have a panel showing traces from our demo app of cart activity!

Solution to 5.3: Case Study: Evaluate the TCO of continuing to support your observability system locally

If we evaluate the options using the TCO, in most cases the answer will be option 2.

This can be justified by using a breakdown of what it will cost the company to support the chosen observability platform across all the factors we talked about in the chapter. We need to make some assumptions here on costs, but even with very average numbers you can see that the costs do not balance.

Option 1: Custom stack

Assumptions

Salaries

- Technical Product Manager Salary (with benefits): 200K/year
- Developer Salary (with Benefits): 250K/year (x2)

Cloud Hosting

- Kubernetes Cluster: 10k/month = 120k/year
- Storage for logs, metrics and traces: 10k/month - 120k/year
- Network, Communication, Security Infrastructure: 5k/month = 60k/year

Timelines

- Time to MVP: 2 months
- Cycle time for new features: 4 weeks

Adoption Roadmap

- 1 Key team at MVP
- Gradual onboarding with expectation for full adoption around 1 year depending on feature and support availability

SLOs

- 98% uptime (could be a stretch without additional ops support)
- New software version availability: Within 6 months of release

Year 1 Summary

- Cost: 750K
- Time to full adoption: 1 year
- Availability: 98%
- Latest OSS package Features: Within 6 months of release
- 10-12 custom feature releases per year

Option 2: SaaS Platform

Assumptions

License cost

- Priced based on data retention.
- Estimates show a license cost of \$500K per year (support included)

Timelines

- Configured and available for full company-wide adoption in 2 months
- New features released as available (often weekly)

SLAs

- 99.99% Uptime or a bill credit is issued

Year 1 Summary

- Cost: 500K
- Time to full adoption: 2 months
- Availability 99.99%
- Feature releases: Weekly

Comparison

While these numbers are fictitious, in our experience the general pattern will be shown to play out the same way in a real situation and often be more lopsided. Here we can see the following:

- Cost: Option 2 is 200k/year cheaper
- Time to value: Option 2 provides adoption across the company in 1/6th of the time. This means that implementation costs have to be absorbed by the company without any benefit for much less time
- Availability: The SaaS option is guaranteed to be available over a week more per year, plus a monetary reimbursement will be received in the event of the guarantee being exceeded.

Last is the value of developer experience in getting features that make the observability platform more useful on a regular basis which will enhance adoption. Without this, costs related to outages as a lack of observability available, or even teams acquiring their own tools will need to be accounted for!

Solution to Exercise 5.4: Use Prometheus data to create SLOs

In our business case we should first think about what business goals of the engineering platform are. Here is an example of what we could think about, you likely can provide many others.

Key Objectives:

- Adoption of the platform by users
- Increase in engineering efficiency
- Stability of releases to the platform

SLOs

- Platform Uptime: 99.95%
- Control plane latency: 95% of new configurations should be recognized within 2s
- Control Plane error rate: Less than 0.1% of properly formed configuration changes should report errors

SLO Justification

- Platform uptime: This will increase stability of the platform itself, minimizing organizational downtime
- Control Plane Latency: This provides fast feedback time to engineering teams, increasing efficiency of teams releasing new configurations
- Control Plane Error Rate: Minimizes false positives, increasing developer trust in the platform and the willingness to adopt

SLIs

- `etcd_server_has_leader` (indicates control plane network failure)
- `apiserver_request_duration_seconds` (control plane latency)
- `scheduler_e2e_scheduling_duration_seconds` (Scheduler latency)
- `kube_apiserver_workqueue_errors` (control plane errors)

Chapter 6 Solutions

Solutions to Exercise 6.1: Assess a platform developer tool according to the software selection criteria

Example answer:

https://github.com/effective-platform-engineering/companion-code/blob/main/chapter-6/6.1_assess_a_tool_in_one_of_the_categories_against_the_criteria/assessment-worksheet.md

Solutions to Exercise 6.2: Create and run a configuration test against a built-in AWS role

Expected result.

```
$ rspec test.rb --format documentation
```

```
iam_role 'AdminUsersRole'
  is expected to exist
  is expected to have iam policy "AdministratorAccess"
```

Finished in 0.54632 seconds (files took 5.48 seconds to load)

2 examples, 0 failures

```
$ rspec test_with_error.rb --format documentation
```

```
iam_role 'AdminUsersRole'
  is expected to exist
  is expected to have iam policy "AdministratorAccess_oops" (FAILED - 1)
```

Failures:

```
1) iam_role 'AdminUsersRole' is expected to have iam policy "AdministratorAccess_oops"
   Failure/Error: it { should have_iam_policy('AdministratorAccess_oops') }
     expected `iam_role 'AdminUsersRole'.has_iam_policy?("AdministratorAccess_oops")` to be
   truthy, got nil
     # ./test.rb:5:in `block (2 levels) in <top (required)>'
```

Finished in 0.54097 seconds (files took 6.41 seconds to load)

2 examples, 1 failure

Failed examples:

```
rspec ./test.rb:5 # iam_role 'AdminUsersRole' is expected to have iam policy
"AdministratorAccess_oops"
```

Solutions to Exercise 6.3: Perform static analysis of terraform code

Using the Terraform CLI, validate the syntax and canonical formatting of main.tf

Solution:

```
$ terraform init
$ terraform validate .
```

```
Error: expected cidr_block to contain a valid Value, got: 10.0.0.0 with err: invalid CIDR
address: 10.0.0.0
```

...

Correct the error and re-run the test.

Using the Terraform CLI, apply canonical formatting to main.tf

Solution:

```
$ terraform fmt
```

```
$ cat main.tf
```

```
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
  tags = {
    Name      = "main"
    Pipeline  = "https://github.com/my-org-name/my-repo-name"
  }
}

resource "aws_security_group_rule" "my_sg_rule" {
  type          = "ingress"
  from_port     = 0
  to_port       = 65535
  protocol      = "tcp"
  cidr_blocks   = ["0.0.0.0/0"]
  security_group_id = "sg-123456"
}
```

#A Notice how the file has been changed so that the assignment characters ("=") are now aligned vertically.

Using Tflint, check the code style and formatting for best practices.

Solution.

```
$ tflint
```

```
...
Warning: terraform "required_version" attribute is required (terraform_required_version)
```

```
...
Warning: Missing version constraint for provider "aws" in "required_providers"
```

```
...

Make the recommended changes until Tflint no longer returns warnings.
```

Finally, run a Trivy scan against main.tf

```
$ trivy config main.tf
```

```
...
CRITICAL: Security group rule allows ingress from public internet.
```

```
...
MEDIUM: VPC Flow Logs is not enabled for VPC
```

```
...
LOW: Security group rule does not have a description.
```

Correct the security issues in main.tf. For the MEDIUM alert, rather than adding a VPC Flow Log configuration, research the Trivy documentation for instructions on how to add an inline comment to ignore that particular security check.

Solutions to Exercise 6.4: Implement static analysis through commit hooks

Solution.

Contents of .pre-commit-config.yaml.

Repos:

- repo: https://github.com/antonbabenko/pre-commit-terraform #A
rev: v1.99.0
Hooks:
 - id: terraform_validate
 - id: terraform_fmt
 - id: terraform_tflint
 - id: terraform_trivy
- repo: local #B
Hooks:
 - id: git-secrets
name: git-secrets
entry: git-secrets
language: system
args: ["--scan"]
- repo: https://github.com/pre-commit/pre-commit-hooks
rev: v4.6.0
Hooks:
 - id: check-symlinks #C
 - id: check-merge-conflict
 - id: check-added-large-files
 - id: forbid-new-submodules
 - id: check-executables-have-shebangs #D
 - id: check-json
 - id: check-yaml
args: [--allow-multiple-documents]
 - id: trailing-whitespace
args: [--markdown-linebreak-ext=md]

#A This pre-commit plug has built-in commands for each Terraform tool in our previous examples.

#B Example of the commit-hook referencing a locally installed tool.

#C These four lines check for common git conventions

#D The remainder of the commands in this general list perform basic linting against the common file types frequently found alongside Terraform files in an infrastructure pipeline: YAML, JSON, bash scripts, and markdown.

Solutions to Exercise 6.5: Experiment with Self-Hosted Runners

Follow the instructions to authenticate using the CircleCI cli.

If you haven't already, you will need to define an organizational namespace for your circleci account.

```
$ circleci namespace create <name> --org-id <your-organization-id>
```

The <your-organization-id> is the GitHub organization name that you integrated with CircleCI. Circle recommends that you use the same name for the namespace, but you are free to use any unique name. An organization may only have a single namespace. The same namespace is used for both your organization's orbs as well as private runner resource classes.

For epetech, assuming our github organization is also epetech then we could create the resource namespace as:

```
$ circleci namespace create epetech --org-id epetech
```

Next, create a resource class to identify your self-hosted runner namespace with the following command.

```
$ circleci runner resource-class create epetech/experiment 'private runner experiment' --generate-token
```

api:

```
auth_token: f776a3e34*****
```







```
+-----+
| RESOURCE CLASS | DESCRIPTION |
+-----+-----+
| epetech/experiment | private runner experiment |
+-----+-----+
```

Store the auth_token in our secrets manager.

Now, let's run a local instance of Kubernetes using Kind and create a Kubernetes namespace for this exercise.

```
$ kind create cluster --name experiment
```

Creating cluster "experiment" ...

- ✓ Ensuring node image (kindest/node:v1.29.2) 
- ✓ Preparing nodes 
- ✓ Writing configuration 
- ✓ Starting control-plane 
- ✓ Installing CNI 
- ✓ Installing StorageClass 

Set kubectl context to "kind-experiment"

You can now use your cluster with:

```
kubectl cluster-info --context kind-experiment
```

Create a circleci namespace.

```
$ kubectl create namespace circleci
```

Add the circleci helm repository using the following commands.

```
$ helm repo add container-agent https://packagecloud.io/circleci/container-agent/helm
$ helm repo update
```

Create a values.yaml for this helm deployment with the following contents.

Agent:

```
resourceClasses:
  epetech/experiment:
    token: f776a3e34*****
```

Use the <namespace> you previously created and the token generated by creating the runner resource class above.

Deploy the runner agent using the following command.

```
$ helm install container-agent container-agent/container-agent -n circleci -f values.yaml
```

If everything worked, you should see something like the following on your Self-Hosted Runners tab in CircleCI.

Self-Hosted Runners

[Create Resource Class](#)

Concurrency Usage ⓘ

0 of 5 tasks used

Resource Class	Runners	Version	Active Tasks	Last Contact	State	
twdps/experiment	container-agent-b8c5c5fdd-b6gfg	3.0.19-5234-08c7340	No Tasks	3s	idle	+

Finally, create a git repo and add the following simple pipeline to test our private runner. Use the <namespace> and <resource-class> you defined above.

```
# .circleci/config.yml
```

```
version: 2.1
```

```
jobs:
```

```
  build:
```

```
    docker:
```

```
      - image: cimg/base:current
```

```
      resource_class: epetech/experiment
```

```
    steps:
```

```
      - checkout
```

```
      - run: echo "Hi I'm on Runners!"
```

```
workflows:
```

```
  build-workflow:
```

```
    jobs:
```

```
      - build
```

Connect this repo from the Projects tab in CircleCI and start the pipeline building. You will see the executor initialize and run on your local cluster, followed by the example pipeline project using the runner to perform the pipeline tasks. The CircleCI project window will look something like this:

The screenshot displays the CircleCI interface for a project named 'experiment-circleci-private-runners'. The breadcrumb trail is: Dashboard > Project > experiment-circleci-private-runners > main > build-workflow > build (3). The job 'build' is shown as 'Success' with a green checkmark. Below the job status, a summary bar shows: Duration / Finished (9s / 4s ago), Queued (0s), Executor (twdps/experiment), Branch (main), Commit, and Author (Nic Cheneweth). The 'STEPS' section is expanded, showing a list of steps: 'Task lifecycle' (14s), 'Preparing environment variables' (0s), 'Checkout code' (1s), and 'echo "Hi I'm on Runners!"' (0s). A 'Parallel runs' banner at the top of the steps section encourages using parallelism to run faster tests.

See the CircleCI documentation¹ for additional information. If you are attempting the exercises in the rest of this book without access to a cloud vendor, it is possible to use a private runner on a local Kind cluster to test many of the kubernetes service and extension deployments.

Solutions to Exercise 6.6: Bootstrap our nonproduction and

¹ <https://circleci.com/docs/container-runner-installation/>

production AWS accounts with the initial service accounts and pipeline role

Start with the role permission that will be needed by this pipeline to manage IAM Users, Groups, Policies, and Roles.

In good TDD form, create a folder called test and add an AwsSpec test called platform_iam_profiles_role_spec.rb. Test for the role and the permissions we want the role to provide.

```
require 'awspec'

describe iam_role('PlatformIamProfilesRole') do
  it { should exist }
  it { should have_iam_policy('PlatformIamProfilesRolePolicy') }
  it { should be_allowed_action('iam:*') } #A
end
```

#A Normally, don't use a wildcard to define permission details. Review the available IAM permissions² and decide which ones you think a pipeline will need. Include matching lines for each permission needed. Getting the permission list correct can take a few iterations. A detailed example is provided in the companion code for these exercises.

Use your own AWS credentials for the first account and confirm that the test will run. You should expect to see a failing test result as we have not yet applied this configuration.

```
$ rspec test/platform_iam_profiles_role_spec.rb --format documentation
iam_role 'PlatformIamProfilesRole'
  is expected to exist (FAILED - 1)
  is expected to have iam policy "PlatformIamProfilesRolePolicy" (FAILED - 2)
  ...
```

Failures:

```
1) iam_role 'PlatformIamProfilesRole' is expected to exist
   Failure/Error: it { should exist }
   expected iam_role 'PlatformIamProfilesRole' to exist
   ...
```

Finished in 1.3 seconds (files took 5.68 seconds to load)
5 examples, 5 failures

Create similar tests for our service accounts. Keep in mind, in a multiaccount setting, we won't run those tests against every account just the account where the services accounts are created. One method is to put this test into a subfolder in the test folder. By default rspec will not recursively go through tests in a folder.

```
$ cat test/state/platform_iam_service_accounts_spec.rb

require 'awspec'

describe iam_group('NonprodServiceAccountGroup') do
  it { should exist }
  it { should have_iam_user('NonprodServiceAccount') }
```

2

https://docs.aws.amazon.com/service-authorization/latest/reference/list_awsidentityandaccesamanagementiam.html

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://livebook.manning.com/#!/book/book-title/discussion>

```

end

describe iam_policy('NonprodServiceAccountGroup') do
  it { should exist }
end

describe iam_group('ProdServiceAccountGroup') do
  it { should exist }
  it { should have_iam_user('ProdServiceAccount') }
end

describe iam_policy('ProdServiceAccountGroup') do
  it { should exist }
end

```

With both of our tests happily failing, let's write the terraform code to create the first role.

```

# platform-iam-profiles-role.tf

module "PlatformIamProfilesRole" {
  Source = "terraform-aws-modules/iam/aws//modules/iam-assumable-role"
  version = "5.55.0" #A

  create_role = true

  role_name      = "PlatformIamProfilesRole"
  role_path      = "/PlatformRoles/"
  role_requires_mfa = false
  custom_role_policy_arns = [aws_iam_policy.PlatformIamProfilesRolePolicy.arn]
  number_of_custom_role_policy_arns = 1
  trusted_role_arns = ["arn:aws:iam::${var.state_account_id}:root"] #B
}

```

#A Pin the module to a specific version. While it is a good practice to pin modules to a particular commit SHA rather than a version tag as the module owner may move the commit to which a tag applies, keep in mind that some module registries, such the Hashicorp public terraform registry, do not yet support this practice.

#B We define the account where our service-account identities live as the trusted source. In this example we are trusting any IAM User who has been granted access to assume roles outside the state account. Since we are the owners of this account and are only creating service accounts that should be able to do so this can be an acceptable starting point for simplicity sake. It is also good practice to narrow this to a more strictly identifiable source of service account identities. This is an example of where placing all service account users within a spefeici path would make that easier.

In the same file, we also need to define the `aws_iam_policy` resource to be attached to the role.

```

resource "aws_iam_policy" "PlatformIamProfilesRolePolicy" {
  name = "PlatformIamProfilesRolePolicy"
  path = "/PlatformPolicies/" #A
  policy = jsonencode({
    "Version" : "2012-10-17"
    "Statement" : [
      {
        "Action" : [
          "iam:*" #B
        ]
        "Effect" : "Allow"
        "Resource" : "*"
      },

```

```

    ]
  })
}

```

#A It is nearly always useful to place the platform level role policies into a dedicated path. This makes it easier to distinguish our product policies.

#B Would not normally be a wildcard. Using this here for simplicity. See exercise code samples in github for a detailed example.

Next we need to add terraform resources for managing our service accounts. Put the service account resources in `service_accounts.tf`

```

module "NonprodServiceAccount" {
  source = "terraform-aws-modules/iam/aws//modules/iam-user"
  version = "5.55.0"

  create_user = var.is_state_account #A
  name        = "NonprodServiceAccount"
  path        = "/PlatformServiceAccounts/" #B
  create_iam_access_key = false #C
  create_iam_user_login_profile = false #D
  force_destroy = true
  password_reset_required = false
}

```

#A The service accounts should only be created in the state account. Use a terraform variable to indicate whether or not to create the identities based on the account being configured.

#B Store all such service accounts in a single path. This will make it easier to manage the service account credentials.

#C Do not use terraform to manage the access credentials. Though there are effective, secure means of letting terraform manage credentials, terraform is not well suited to automated credential rotation. Create and rotate these credentials as a separate pipeline step.

#D Service accounts should not have console access.

As is, the above non-prod service account has no permissions. Define an IAM Group that grants members the ability to assume roles in all non-production accounts.

```

module "NonprodServiceAccountGroup" {
  source = "terraform-aws-modules/iam/aws//modules/Iam-group-with-assumable-roles-policy"
  version = "5.37.1"

  count = var.is_state_account ? 1 : 0 #A
  name   = "NonprodServiceAccountGroup"
  path   = "/PlatformGroups/"
  assumable_roles = var.all_nonprod_account_roles #B

  # include the nonprod service account in this nonprod group
  group_users = [
    module.NonprodServiceAccount.iam_user_name
  ]
}

```

#A The service accounts should only be created in the state account.

#B This variable is an environment parameter that defines a list of all the non-production accounts of the engineering platform and will look something like this:

```

"all_nonprod_account_roles": [
  "arn:aws:iam::111111111111:role/PlatformRoles/*", # state account
  "arn:aws:iam::222222222222:role/PlatformRoles/*", # sandbox account
]

```



```
"arn:aws:iam::333333333333:role/PlatformRoles/*" # non-prod account
]
```

You will only need to list a single account if that is all you are using for exercise purposes.

Next, add the Production service account and group. These are identical to the nonprod definition except of course for the name and that it will be permitted to assume roles in the production accounts as well.

We also need a versions.tf file to pin terraform and the aws provider versions and to describe the location of our backend state store.

```
terraform {
  required_version = "~> 1.9"           #A
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 5.94"               #A
    }
  }

  backend "remote" {
    hostname     = "app.terraform.io"
    organization = "my_terraform_cloud_org"
    workspaces {
      prefix = "platform-iam-profiles-" #B
    }
  }
}

provider "aws" {
  region = var.aws_region
  # assume_role {                     #C
  #   role_arn = "arn:aws:iam::${var.aws_account_id}:role/${var.aws_assume_role}"
  #   session_name = "iam-profiles"
  # }

  default_tags {                      #D
    tags = {
      product = "Epetech.online engineering platform"
      pipeline = "platform-iam-profiles"
    }
  }
}
```

#A Pin terraform and provider versions.

#B Use prefix names that match the repo name so that workspaces are easier to find. This backend example uses "remote" which is a reference to the terraform cloud account we set up and are using to store our state files.

#C Normally, when this pipeline runs it will assume the platform-iam-profiles role, however when we first bootstrap this role we are not able to do so as the role doesn't yet exist. Comment out this section until after we have done the initial manual application.

#D Adopt a consistent tagging strategy. This is an organizational need more than a specifically engineering platform requirement. We will want to have a tagging strategies that lets us track and allocate costs. At minimum there should be a product identifier and a reference to the pipeline that orchestrates this code. For ease of moving between development tools, an effective choice is the repository name.

We need a variables.tf file to define each of the terraform variables used above. Use validations to help detect invalid configuration. Examples of variables validation:

```
variable "aws_region" {
  type = string
```

```

validation {
    #A
    condition = can(regex("[a-z][a-z]-[a-z]+-[1-9]", var.aws_region))
    error_message = "Invalid AWS Region name."
}
}

variable "aws_account_id" {
    type = string
    validation {
        #B
        condition = length(var.aws_account_id) == 12 && can(regex("^\\d{12}$",
var.aws_account_id))
        error_message = "Invalid AWS account ID"
    }
}

variable "aws_assume_role" { type = string }

variable "is_state_account" {
    description = "create STATE account configuration?"
    type        = bool
    default     = false
}

variable "state_account_id" {
    description = "arn principal root reference to state account id where all svc accounts are
defined"
    type        = string
    validation {
        #B
        condition = length(var.state_account_id) == 12 && can(regex("^\\d{12}$",
var.state_account_id))
        error_message = "Invalid AWS account ID"
    }
}

variable "all_nonprod_account_roles" {
    description = "arn reference to * roles for all non-production aws accounts;
arn:aws:iam::*****12345:role/*"
    type        = list(any)
}

variable "all_production_account_roles" {
    description = "arn reference to * roles for all production aws accounts;
arn:aws:iam::*****12345:role/*"
    type        = list(any)
}

```

#A Enforces a format of two letters, followed by a dash, followed by several letters, then a dash, and finally a number.

#B This account validator confirms 12 numbers.

Finally, we need the list of actual values that should populate the variables for each account. If we were using the four accounts described in the usual enterprise greenfield setting, git push will trigger a pipeline that deploys this config to the state account. Subsequently, tag the commit to apply this configuration to the sandbox, nonprod, and prod account in order. In this simplified architecture the combined nonprod account environment values will be:

```
$ cat environments/nonprod.auto.tfvars.json.tpl
```

```

{
  "aws_region": "us-east-1",
  "aws_account_id": "{{ op://my-vault/aws-2/aws-account-id }}", #A
  "aws_assume_role": "PlatformRoles/PlatformIamProfilesRole",
  "is_state_account": true,
  "state_account_id": "{{ op://my-vault/aws-2/aws-account-id }}", #B
}

```

```

    "all_nonprod_account_roles": [
      "arn:aws:iam::{{ op://my-vault/aws-2/aws-account-id }}:role/PlatformRoles/*"
    ],
    "all_production_account_roles": [
      "arn:aws:iam::{{ op://my-vault/aws-1/aws-account-id }}:role/PlatformRoles/*"
    ]
  }
}

$ cat environments/prod.auto.tfvars.json.tpl

{
  "aws_region": "us-east-2",
  "aws_account_id": "{{ op://my-vault/aws-1/aws-account-id }}", #A
  "aws_assume_role": "PlatformRoles/PlatformIamProfilesRole",
  "is_state_account": false,
  "state_account_id": "{{ op://my-vault/aws-2/aws-account-id }}",
  "all_nonprod_account_roles": [
    "arn:aws:iam::{{ op://my-vault/aws-2/aws-account-id }}:role/PlatformRoles/*"
  ],
  "all_production_account_roles": [
    "arn:aws:iam::{{ op://my-vault/aws-1/aws-account-id }}:role/PlatformRoles/*"
  ]
}

```

#A Account IDs are not generally considered secrets. But let's use this first pipeline to introduce an effective pipeline management approach for secrets in the tfvars file.

#B Since we are using the same account both as a state account as as the sandbox account, the values will be the same.

The two environment value files are in the environments folder and are in the form of templates into which we will need to inject the account numbers. We will use an orb command to perform this step in the pipeline but for this bootstrap step we need to do it manually. Note that the template filenames suggested above includes the .auto. directive which will cause terraform to automatically load the values without needing to use the -var-file flag. When this happens during a pipeline run, the step will write the file to the root folder and since this is taking place on the ephemeral runner no change is actually being made to the stored code. We can do this now manually for the account we are bootstrapping.

```
$ op inject -i environments/nonprod.auto.tfvars.json.tpl -o nonprod.auto.tfvars.json
```

This is a good point to define a .gitignore file to prevent manually testing of actions that are intended for the ephemeral runner from accidentally getting written into version control.

```

# .gitignore
.terraform*      #A
*.auto.tfvars.json #B

```

#A Exclude the local files terraform creates. When run in the pipeline, Terraform will also generate these files but that is an ephemeral location that will not preserve the files.

#B We discussed above a pattern where information inside an environment tfvars file could include injected, secure information. When doing that, an effective pattern is for the pipeline to pull the template from a folder and write the contents to the root location as an auto.tfvars. Now the parameters will be included in the terraform commands without needing to specifically include them. To prevent accidentally including the results from populating a template into the repository history, ignore these *.auto files.

Make sure you have the .terraformrc credential file setup on your workstation for access to the cloud state store. Using your personal AWS administrative credentials for the first account, work through the usual terraform init, plan, and apply stages at the command line until you have successfully created the two service accounts and added the initial

platform-iam-profiles-role to the non-prod aws account. Remove the nonprod tfvars file, generate the prod tfvar file from the template and repeat the process to add the platform-iam-profiles-role to the prod aws account using your personal prod account credentials. Using your personal credentials in a bootstrap results in the correct audit log contents.

Now that we are adding actual Terraform code, the trivy scans our commit hook triggers will start reporting warning messages. For example, trivy will warn that the service accounts we are creating do not have MFA configuration turned on and that various password policies related to console access are not met. These are basically false positives as service accounts cannot provide MFA information, nor are they granted console access. Add a .trivyignore file to turn off these sorts of warnings.

```
$ cat .trivyignore
# ignore MFA warning for services accounts
AVD-AWS-0123
# ignore password policy warnings.
# Service accounts are not allowed console access.
AVD-AWS-0063
AVD-AWS-0056
AVD-AWS-0059
```

```
.python-version      #C
.ruby-version
.venv
credentials          #D
```

#C Using pre-commit and awscli means both Python and Ruby configuration information is needed. You do not necessarily need to preserve this information.

#D Our example solution uses the python package iam-credential-rotation and writes the resulting new credentials to a file. In a pipeline, the file is ephemeral and secure and will not be preserved. But for local testing, if you use the python tool you will want to make sure to not accidentally store the results in the repository.

Solutions to Exercise 6.7: Create the CI and development test pipeline for IAM service accounts and roles

Start with the first step from our workflow diagram.

workflows:

deploy service accounts and roles to state account: #A

jobs:

```
- terraform/static-analysis:          #B
  name: static code analysis
  context: *context                    #C
  executor-image: *executor-image      #D
  workspace: nonprod                  #E
  tflint-scan: true                    #F
  tflint-provider: aws                 #G
  trivy-scan: true                     #H
  filters: *on-push-main               #I
```

#A The first work flow will be all the steps that happen when we push a code change. Since that will involve deploying service accounts and roles to the first account, we will name the

workflow with a description that hopefully makes that clear.

#B The first step in our workflow is to perform a static analysis of our terraform code. We are using the twdps/terraform@3.1.1 orb that is publicly available in the CircleCI orb registry. This orb contains a job called static-analysis

#C Add the team context we set up in the prerequisites section so that our pipeline will be able to successfully use the 1password orb to fetch secrets from our op vault. Since we will need to provide the context in many different jobs in this workflow, we defined the context setting in a global anchor that we can refer to wherever needed and yet change it by making a change in only one location.

#D The terraform orb uses a preconfigured opensource executor that comes with all the needed versions of tools. Though the orb will use the latest stable release by default, we will always want to pin to a specific executor so that the tool versions will align to our code and environments.

#E This configuration is an account-level config. Since we are only using two accounts in the exercising, we can refer to the test environment as simply nonprod and then prod for the prod account. We will name our workspaces to match.

#F The static-analysis job in the terraform orb will perform tfint scan by setting the parameter to true.

#G And since we are use AWS, we tell the tfint command in the orb to use the AWS provider plugin for tfint.

#H We also want to perform a Trivy scan and can accept the default scan parameters.

#I Use our git-push anchor to define the filter that will limit this step to running only when a change is pushed to main.

With this step defined, we can push these changes and then set up the pipeline in CircleCI.

Now add the plan step:

```
- terraform/plan:
  name: nonprod change plan
  context: *context
  executor-image: *executor-image
  workspace: nonprod
  before-plan: #J
  - set-environment:
    account: nonprod
  filters: *on-push-main #K
```

#K We do not need to make the plan step dependent on a successful completion of the static-analysys step, the jobs can run in parallel.

#J Now that we are running the Terraform plan we will need some credentials available. Specifically, we will need the NonprodServiceAccount AWS credentials which the pipeline will use in assuming the platform-iam-profiles Role. And we will need the credential for our Terraform cloud account where we are storing the terraform state files. In addition, we will need to tfvars values for the nonprod account.

Getting these credentials setup in the correct manner will require a few steps. Create a local command in the command section of our pipeline to manage setting this up.

commands:

```
set-environment: #L
  description: generate environment credentials
    and configuration from templates
parameters:
```

```

account:                                     #M
  description: account to be configured
  type: string
steps:
- op/env:                                   #N
  env-file: op.<< parameters.account >>.env
- op/tpl:                                   #O
  tpl-path: environments
  tpl-file: << parameters.account >>.auto.tfvars.json
- terraform/terraformrc                    #P

```

#L Use a description name for our setup command.

#M We will use this command both in this CI (git push) job as well as a tagged release. Define a parameter that we can pass to the set-environment command to support this usage.

#N First let's fetch the values we will need from our 1password vault using an op.env file.

```

# op.nonprod.env
export TFE_TOKEN={{ op://my-vault/svc-terraform-cloud/team-api-token }}

export AWS_ACCESS_KEY_ID={{
op://my-vault/aws-account-2/NonprodServiceAccount-aws-access-key-id }}
export AWS_SECRET_ACCESS_KEY={{
op://my-vault/aws-account-2/NonprodServiceAccount-aws-secret-access-key }}

```

Specifically, we will need to initialize the terraform backend and assume an IAM role so we will need the team terraform cloud api-token and the NonprodServiceAccount credentials that we created and saved earlier. We will have two op.env files. One for each account we will configure. We pass in the account name and use this to build the filename where the values are defined. The step uses a command from the 1password orb. Note the 1password value reference is made up of the vault, the item, and the field from the item. Use the details from the location where you stored these credentials. Adopt a naming convention that makes it easy to reference secrets from the vault.

#O We also need to take our tfvars template from the environments folder and inject the account_id into the file and then write a copy to the root folder that will automatically be picked up by terraform as our tfvars file for this plan. There is also a command in the 1password that performs this step. The command supports a number of parameters to support overriding the location and naming conventions of the file. This example assumes the templates are in the environment folder and that they are named to match the provided template name plus .tpl. Using json format for tfvars files makes supporting templates and other kinds of runtime analysis much easier.

#P Lastly, we want to use the TFE_TOKEN from our environment values that were loaded in the first step to write a ~/.terraformrc file in the correct format for accessing our terraform cloud state store. The terraform orb includes a command that will do this.

Push these changes and check the result plan output in the step logs in circleci.

Once the analysis and plan steps are running, we can add an approval step that will pause the pipeline while we evaluate the plan output. This approval step should be contingent on the successful completion of the prior steps.

```

- approve nonprod changes:
  type: approval
  requires:                               #Q
    - static code analysis
    - nonprod change plan
  filters: *on-push-main

```

#Q This approval step should be contingent on the successful completion of the prior steps. Use the requires: key to set this requirement.

Now add the apply step.

```
- terraform/apply:
  name: apply nonprod changes
  context: *context
  workspace: nonprod
  Before-apply:
    - set-environment:
        account: nonprod
  Requires:                                #R
    - approve nonprod changes
  filters: *on-push-main
```

#R At this point, the apply step parameters we supply the apply job from the terraform orb is nearly identical to our plan step. This step must be contingent on the approval step.

At this point we can push these changes and our pipeline will perform the first two steps, then pause waiting for us to review the plan. If we think the plan looks good, we can approve the next step and the changes will be applied to the account.

The next step in the workflow diagram is to perform the aws spec configuration tests we created earlier to confirm the service accounts and roles are configured as we expect.

We could create a complete new job, but our terraform orb supports adding custom steps after the apply step similar to how we could add custom steps prior to performing the apply step. Since AwsSpec is an rspec extension and rspec tests are run using a standard pattern, we should consider creating either an aws spec orb, or perhaps just adding an aws spec command to our terraform orb. We probably should to that before using the terraform orb internally with our platform customers should we ever provide them with terraform starterkits. But we can also create a local command. In either case, supplying this command to the apply job as an after-apply parameter is cleaner than creating a whole new job.

```
after-apply:
  - aws-integration-tests:                #S
    account: nonprod
```

#S What do we need to do in the integration test?

```
aws-integration-tests:
  parameters:
    account:
      description: for example solution this is either nonprod or prod
      type: string
  steps:
    - run:
        name: Aws spec tests of pipeline managed AWS resources
        command: bash scripts/run_aws_spec_integration_tests.sh << parameters.account
>>
```

In keeping with our dump-pipelines strategy, the actual logic of the test is better kept in a local bash script. We will pass our account-name environment value to the script. Here is a possible solution for an AwsSpec testing script.

```
# scripts/run_aws_integration_tests.sh
#!/usr/bin/env bash

source bash-functions.sh
set -eo pipefail
```

```

export environment=$1
export aws_account_id=$(jq -er .aws_account_id "$environment".auto.tfvars.json)
#T

export aws_assume_role=$(jq -er .aws_assume_role "$environment".auto.tfvars.json)
export AWS_DEFAULT_REGION=$(jq -er .aws_region "$environment".auto.tfvars.json)
awsAssumeRole "${aws_account_id}" "${aws_assume_role}"      #U

# test roles
rspec test/*.rb --format documentation                      #V

# if this is the state account then test the profiles
if [[ ${environment} == "nonprod" ]]; then                  #W
  rspec test/state/platform_iam_service_accounts_spec.rb --format
  Documentation
Fi

```

#T These export command pull the required setting from our tfvars file.

#U When we did the terraform-apply step, terraform used the assume_role information in the provider resource to assume the correct role. But we are not using terraform in this step and out ENV just has the service account credentials. Before we can run the tests, we must assume the correct role. This bash function performs that step. But where is this function and how can we call it locally?

Add the following line to the set-environment command:

- do/bash-functions

This is a command that is in the pipeline-events orb we included in our pipeline. When you call this command, the file bash-functions.sh is written to the PWD. Take a look at the source code for the orb to see what functions it provides and what each does.

#V Now we can run all the aws spec tests in the test folder. At this point there is only the test of the platform-iam-profiles role. But as we add more roles for future pipelines there will be more test files.

#W And finally this script checks to see if we are in the workspace that includes our AWS services accounts. If so then it runs those tests, which we placed in the state folder.

Push these changes and debug any issues. At this point we have everything in our first stage pipeline workflow running except for the IAM service account credentials rotation.

Solutions to Exercise 6.8: Add a credential rotation step to the first stage pipeline

Add the rotate-service-account-credentials command.

command:

```

rotate-service-account-credentials:
  parameters:
    account:
      description: use this account environment values
      type: string
  Steps:

```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://livebook.manning.com/#!/book/book-title/discussion>


```
- run:
  name: create or rotate all service account credentials
  command: bash scripts/rotate_svc_acct_credentials.sh << parameters.account >>
```

This command will call a local bash script, passing the name of the tfvars file to use to find the correct account information.

Now, create the bash script that will use iam-credential-rotation to fetch new credentials, removing the oldest, and then save the new credentials to 1password overwriting the prior values.

```
#!/usr/bin/env bash
source bash-functions.sh

set -eo pipefail

export environment=$1
export aws_account_id=$(jq -er .aws_account_id "$environment".auto.tfvars.json)
#A
export aws_assume_role=$(jq -er .aws_assume_role "$environment".auto.tfvars.json)
export AWS_DEFAULT_REGION=$(jq -er .aws_region "$environment".auto.tfvars.json)

awsAssumeRole "${aws_account_id}" "${aws_assume_role}" #B

# Rotate AWS IAM User access credentials. https://pypi.org/project/iam-credential-rotation/
echo "rotate service account credentials"
iam-credential-rotation PlatformServiceAccounts > machine_credentials.json #C

# Write new nonprod sa credentials to 1password
echo "write NonprodServiceAccount credentials"

NonprodServiceAccountCredentials=$(jq -er .NonprodServiceAccount
machine_credentials.json) #D
NonprodAccessKey=$(echo $NonprodServiceAccountCredentials | jq .AccessKeyId | sed
's/"//g' | tr -d \\\n)
NonprodSecret=$(echo $NonprodServiceAccountCredentials | jq .SecretAccessKey | sed
's/"//g' | tr -d \\\n)

write1passwordField "my-vault" "aws-account-2"
"NonprodServiceAccount-aws-access-key-id" "$NonprodAccessKey" #E
write1passwordField "my-vault" "aws-account-2"
"NonprodServiceAccount-aws-secret-access-key" "$NonprodSecret"

# Write new prod sa credentials to 1password vault #F
echo "write ProdServiceAccount credentials"
ProdServiceAccountCredentials=$(jq -er .ProdServiceAccount machine_credentials.json)
ProdAccessKey=$(echo $ProdServiceAccountCredentials | jq .AccessKeyId | sed 's/"//g' | tr
-d \\\n)
ProdSecret=$(echo $ProdServiceAccountCredentials | jq .SecretAccessKey | sed 's/"//g' | tr
-d \\\n)

write1passwordField "my-vault" "aws-account-2" "ProdServiceAccount-aws-access-key-id"
"$ProdAccessKey"
write1passwordField "my-vault" "aws-account-2"
"ProdServiceAccount-aws-secret-access-key" "$ProdSecret"
```

#A We load the aws account id, role to assume, and default region from the current tfvar file in use.

#B Then we use the same awsAssumeRole bash function we used in our aws spec test to fetch short-lived credentials based on the role we must assume to perform this task.

#C Use the iam-credential-rotation script to rotate both of service accounts, capturing the output into a local file.

#D The next three lines parse out the credential values for the NonprodServiceAccount from the file output of the rotation utility.

#E Now we will make use of the second bash function that the do/bash-functions orb command made available to us locally; write1passwordField. You can review the details of the command in the source code of the orb. But to summarize, the script will use the 1password cli that is preinstalled on the executor we are using, and our 1password credentials from our environment to update the contents of the location where we stored the service account credentials.

#F And finally, we repeat the instructions to save the new credentials for the ProdServiceAccount.

Solutions to Exercise 6.9: Create the release pipeline for IAM service accounts and roles

workflows:

...

deploy roles to prod:

```
jobs:
  - terraform/plan:                                #A
    name: prod change plan
    context: *context
    executor-image: *executor-image
    workspace: prod
    before-plan:
      - set-environment:
        account: prod                                #B
      filters: *on-tag-main
  - approve prod changes:
    type: approval
    requires:
      - prod change plan
    filters: *on-tag-main
  - terraform/apply:
    name: apply prod changes
    context: *context
    workspace: prod
    before-apply:
      - set-environment:
        account: prod
    After-apply:                                     #C
      - aws-integration-tests:
        account: prod
    requires:
      - approve prod changes
    filters: *on-tag-main
```

#A We are using the terraform orb in the same way as we did in the first stage, changing only the workspace name to match the correct tfvars and backend state name.

#B We are making this change in a different account so we need an account specific op.env

file. The TFE_TOKEN is the same, but since this is our Production account, we use the ProdServiceAccount credentials.

```
# op.prod.env
export TFE_TOKEN={{ op://my-vault/svc-terraform-cloud/team-api-token }}
export AWS_ACCESS_KEY_ID={{
op://my-vault/aws-account-2/ProdServiceAccount-aws-access-key-id }}
export AWS_SECRET_ACCESS_KEY={{
op://my-vault/aws-account-2/ProdServiceAccount-aws-secret-access-key }}
```

#C Notice, we do not want to rotate the service account credentials in the release pipeline. The service accounts only exist in one account.

See the example complete repository contents at
<https://github.com/effective-platform-engineering/companion-code>.

Chapter 7 Solutions

Solutions to Exercise 7.1: Create a release pipeline for hosted zone and zone delegation

First we need to define a new role in the platform-iam-profiles repository. Start by adding the associated test.

```
$ cat test/platform_hosted_zones_spec.rb

require 'awspec'

describe iam_role('PlatformHostedZonesRole') do
  it { should exist }
  it { should have_iam_policy('PlatformHostedZonesRolePolicy') }

  it { should be_allowed_action('route53:AcceptDomainTransferFromAnotherAwsAccount') }
  it { should be_allowed_action('route53:AssociateVPCWithHostedZone') }
  it { should be_allowed_action('route53:CancelDomainTransferToAnotherAwsAccount') }
  ...
end
```

#A Review the AWS list of Route53 and Route53Domain permission to decide which permission will be necessary to manage host zones and delegations. See a complete list of permission in the companion code example.

Then, add the actual role definition:

```
$ cat platform_hosted_zones_role.tf

# PlatformHostedZonesRole
#
# Used by: platform-hosted-zones pipeline

module "PlatformHostedZonesRole" {
  source      = "terraform-aws-modules/iam/aws//modules/iam-assumable-role"
  version     = "5.55.0"
  create_role = true

  role_name           = "PlatformHostedZonesRole"
  role_path           = "/PlatformRoles/"
  role_requires_mfa   = false
  custom_role_policy_arns = [aws_iam_policy.PlatformHostedZonesRolePolicy.arn]
  number_of_custom_role_policy_arns = 1

  trusted_role_arns = ["arn:aws:iam::${var.state_account_id}:root"] #A
}

# role permissions
resource "aws_iam_policy" "PlatformHostedZonesRolePolicy" {
  name = "PlatformHostedZonesRolePolicy"
  path = "/PlatformPolicies/"

  policy = jsonencode({
```

```

"Version" : "2012-10-17"
"Statement" : [
  {
    "Action" : [
      "route53:AcceptDomainTransferFromAnotherAwsAccount",
      "route53:AssociateVPCWithHostedZone",
      "route53:CancelDomainTransferToAnotherAwsAccount",
      . . . #B
    ]
    "Effect" : "Allow"
    "Resource" : "*"
  },
]
})
}

```

#A recall, in the four-account strategy, service account only exist in the state account and trust to assume is from that account. In our examples we will use only two account, and in this case what we are calling the nonprod account acts like the state account (use the nonprod account id here).

#B These are the matching permissions from the test.

With the role added, push these changes and release to both accounts via the platform-iam-profile pipeline.

Now, back to our platform-hosted-zones repo, define the tests for the zones to be created in the same account as the primary domain (account-1).

```
$ cat test/prod_account_spec.rb
```

```

require 'awspec'

describe route53_hosted_zone('epetech.io.') do
  it { should exist }
end

describe route53_hosted_zone('prod-i01-aws-us-east-2.epetech.io.') do
  it { should exist }
end

describe route53_hosted_zone('dev.epetech.io.') do
  it { should exist }
end

describe route53_hosted_zone('qa.epetech.io.') do
  it { should exist }
end

describe route53_hosted_zone('api.epetech.io.') do
  it { should exist }
end

```

Next, the the zones delegated to our sandbox account.

```
$ cat test/nonprod_account_spec.rb
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
<https://livebook.manning.com/#!/book/book-title/discussion>

```
require 'awspec'

describe route53_hosted_zone('sbx-i01-aws-us-east-1.epetech.io.') do
  it { should exist }
end

describe route53_hosted_zone('preview.epetech.io.') do
  it { should exist }
end
```

You will save pipeline development time later if you start enabling the pipeline as soon as you have written the automated tests of your configuration.

Add our initial pipeline configuration, defining the orbs we will use and the triggers for our workflows, to `.circleci.config.yml`.

```
---
version: 2.1

orbs:
  terraform: twdps/terraform@3.1.1
  op: twdps/onepassword@3.0.0
  do: twdps/pipeline-events@5.1.0

globals:
  - &context <my-team>
  - &executor-image twdps/circleci-infra-aws:alpine-2025.04

on-push-main: &on-push-main
  branches:
    only: /main/
  tags:
    ignore: /.*/

on-tag-main: &on-tag-main
  branches:
    ignore: /.*/
  tags:
    only: /.*/
```

Since we will apply all changes at once, we will not need different contexts for secrets so we can have a single `op.env` file with the following credentials:

```
export TFE_TOKEN={{ op://my-vault/svc-terraform-cloud/team-api-token }}
export AWS_ACCESS_KEY_ID={{
op://my-vault/aws-account-2/ProdServiceAccount-aws-access-key-id }}
export AWS_SECRET_ACCESS_KEY={{
op://my-vault/aws-account-2/ProdServiceAccount-aws-secret-access-key }}
export SLACK_BOT_TOKEN=op://my-vault/svc-slack/post-bot-token
export GH_TOKEN={{ op://my-vault/svc-github/access-token }}
```

Then add our typical `set-environment` command that injects these values, sets up our terraform cloud access, and so on.

commands:

```

set-environment:
  parameters:
    account:
      description: account description
      type: string
  steps:
    - op/env:
        env-file: op.env
    - op/tpl:
        tpl-path: environments
        tpl-file: << parameters.account >>.auto.tfvars.json
    - terraform/terraformrc
    - do/bash-functions

```

We will actually have only one tfvars file since we apply these changes in a single pass. But we do need the values and we will still inject account id information. Let's call our environment multiaccount to indicate more than one account is involved.

```
$ cat environments/multiaccount.auto.tfvars.json.tpl
```

```

{
  "nonprod_account_id": "{{ op://my-vault/aws-account-2/aws-account-id }}",
  "prod_account_id": "{{ op://my-vault/aws-account-1/aws-account-id }}",
  "assume_role": "PlatformRoles/PlatformHostedZonesRole"      #A
}

```

#A We refer to the pipeline role we created as our first action above.

The first thing our pipeline will do is run the integration tests, so add a command for that. Since we are applying configuration to two accounts at the same time, we can run both of the account-level tests. However, awscli tests will need to assume a role one account at a time so our test script will need to be called at least twice since we are using two accounts in our examples.

Commands:

...

```

integration-tests:
  parameters:
    account:
      description: account description
      type: string
  steps:
    - run:
        name: integration test nonprod account
        command: bash scripts/hosted_zone_test.sh << parameters.account >> nonprod
    - run:
        name: integration test prod account
        command: bash scripts/hosted_zone_test.sh << parameters.account >> prod

```

The hosted_zone_test.sh script is just like our other awscli test procedures. We assume the appropriate role and then run the test using the desired spec file.

```
$ cat scripts/hosted_zone_test.sh
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://livebook.manning.com/#!/book/book-title/discussion>

```
#!/usr/bin/env bash
set -eo pipefail

source bash-functions.sh

export environment=$1
export account=$2
export aws_account_id=$(jq -er ".$${account}"_account_id "$environment".auto.tfvars.json)
#A
export aws_assume_role=$(jq -er .assume_role "$environment".auto.tfvars.json)
export AWS_DEFAULT_REGION=us-east-1

awsAssumeRole "${aws_account_id}" "${aws_assume_role}"      #B

rspec "test/${account}_account_spec.rb"
```

#A Pull the account id and role information from the tfvars file.

#B Use the bash function added by the pipeline-events orb.

We know that our completed pipeline will have a scheduled job to runs these tests routinely. But as part of setting up our pipeline and debugging the flow of the test, let's first add that recurring job as the only job, and leave out the filters or conditions that would normally be used to adjust when the job runs. The will result in the first iteration of our pipeline simply performing the aws spec tests.

jobs:

```
recurring-integration-tests:
  description: |
    Recurring job (weekly) to run pipeline integration tests to detect aws configuration drift
  docker:
    - image: *executor-image
  parameters:
    account:
      description: account description
      type: string
  steps:
    - checkout
    - setup_remote_docker
    - set-environment:
        account: << parameters.account >>
    - integration-tests:
        account: << parameters.account >>
```

Workflows:

```
weekly integration test:
  Jobs:
    - recurring-integration-tests:
        name: AWS hosted zones integration test
        context: *context
        account: multiaccount
```

Push these changes, and start the project building in the CircleCI ui. Confirm that the integration test successfully test for the presence of the desired zones, though of course the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://livebook.manning.com/#!/book/book-title/discussion>

tests are failing at this point.

Now let's add our zone delegation files. In addition to the domain data resource file and preview.epetech.io delegation examples from the chapter section, we need to add zones for each of the zones defined in table 6.1. The preview subdomain is a cross-account delegation, here is an example of delegation with the same account.

```
$ cat zone_dev_epetech_io.tf
```

```
# define a provider in the account where this subdomain will be managed
provider "aws" {
  alias = "subdomain_dev_epetech_io"
  region = "us-east-2"
  assume_role {
    role_arn = "arn:aws:iam::${var.prod_account_id}:role/${var.assume_role}" #A
    session_name = "platform-hosted-zones" #B
  }
}

# create a route53 hosted zone for the subdomain
module "subdomain_dev_epetech_io" {
  source = "terraform-aws-modules/route53/aws//modules/zones"
  version = "3.1.0"
  create = true

  providers = {
    aws = aws.subdomain_dev_epetech_io #C
  }

  zones = {
    "dev.${local.domain_epetech_io}" = { #D
      tags = {
        cluster = "prod-i01-aws-us-east-2" #E
      }
    }
  }
}

# Create a zone delegation in the top level domain for this subdomain
module "subdomain_zone_delegation_dev_epetech_io" {
  source = "terraform-aws-modules/route53/aws//modules/records"
  version = "3.1.0"
  create = true

  providers = {
    aws = aws.domain_epetech_io #F
  }

  private_zone = false #G
  zone_name = local.domain_epetech_io
  records = [
    {
      name = "dev"
      type = "NS"
      ttl = 172800
    }
  ]
}
```

```

    zone_id      = data.aws_route53_zone.zone_id_epetech_io.id
    allow_overwrite = true
    records      =
module.subdomain_dev_epetech_io.route53_zone_name_servers["dev.${local.domain_epete
ch_io}"]
  }
]

depends_on = [module.subdomain_dev_epetech_io]
}

```

#A The provider definition is where we reference the role we just deployed via the profiles pipeline to assume with permissions to make these changes.

#B We can also add a session name to the sts assume role action for audit log purposes.

#C The provider referenced here is the provider where we want to create the hosted zone.

#D This zone is a subdomain of the primary domain.

#E We can add tags to the DNS records.

#F Now we use the provider for the account where the primary domain is registered. In the case of the dev. subdomain this is the same account. In the preview. subomain example it was a different account.

#G In these examples we are using only public domain records.

In addition to the preview and dev subdomains, create files for the following delegations:

```

zone_sbx_i01_aws_us_east_1_epetech_io.tf
zone_qa_epetech_io.tf
zone_api_epetech_io.tf
zone_prod_i01_aws_us_east_2_epetech_io.tf

```

Follow the same pattern and modify for each needed zone and delegation.

With the resource files created, let's begin iterating on the pipeline. Add the scheduling condition to the weekly test job so it will not run with every change:

```

weekly integration test:
  when:
    equal: [ scheduled_pipeline, << pipeline.trigger_source >> ]
  jobs:
    . . .

```

Now, add a workflow for the git-push stage that performs the terraform static analysis and plan jobs.

workflows:

hosted-zones change plan:

```

Jobs:
- terraform/static-analysis:
  name: static code analysis
  context: *context
  executor-image: *executor-image
  workspace: state #A
  tflint-scan: true
  tflint-provider: aws
  trivy-scan: true
  before-static-analysis:

```

```

- op/env:                                #B
  env-file: op.env
  filters: *on-push-main

- terraform/plan:
  name: hosted-zones change plan
  context: *context
  executor-image: *executor-image
  workspace: state
  before-plan:
    - set-environment:
      account: multiaccount            #C
  filters: *on-push-main

```

#A We will only have a single state file for this pipeline. It wouldn't make sense to name it for any one account, so let's just call it state.

#B The static-analysis step only needs our op.env, not the rest of the set-environment command.

#C We still pass an account parameter but we will only pass one, which we named multiaccount.

Push these pipeline changes and debug any issues that the static-analysis reveals and review the resulting change plan.

If everything checks out, add the release stage to our pipeline.

hosted-zones release:

```

jobs:
- terraform/apply:
  name: hosted-zones release
  context: *context
  workspace: state
  before-apply:
    - set-environment:
      account: multiaccount
  after-apply:
    - integration-tests:
      account: multiaccount
    - do/slack-bot:
      channel: engineering platform events
      message: Release platform-hosted-zones
      include-link: true
      include-tag: true
  filters: *on-tag-main

- do/gh-release:
  name: generate release notes
  context: *context
  notes-from-file: release.md
  include-commit-msg: true
  before-release:
    - op/env
  requires:
    - hosted-zones release
  filters: *on-tag-main

```

Push these changes, and then tag the successful commit to make the changes in our accounts.

Before we leave this pipeline, let's complete the scheduling of the recurring tests but add the associated conditions and jobs.

workflows:

```
hosted-zones change plan:
  when:
    not:
      equal: [ scheduled_pipeline, << pipeline.trigger_source >> ]
  jobs:
    . . .
```

schedule weekly rotation and integration test:

```
Jobs:
  - do/schedule-pipeline:
      name: weekly integration test
      context: *context
      scheduled-pipeline-name: weekly-integration-test
      scheduled-pipeline-description: |
        Weekly, automated run of aws hosted zone integration tests.
      hours-of-day: "[1]"
      days-of-week: "[\SUN\]"
      Before-schedule:
        - op/env:
            env-file: op.env
      filters: *on-tag-main
```

See the example complete repository contents at: (link to accompanying code)

Solutions to Exercise 7.2: Create a release pipeline for a role-based network

A new pipeline usually means a new Role defined in our pipeline_iam_profiles repo. Like in the previous pipeline exercise, add the test for the role.

```
$ cat test/platform_vpc_role_spec.rb
```

```
require 'awspec'

describe iam_role('PlatformVPCRole') do
  it { should exist }
  it { should have_iam_policy('PlatformVPCRolePolicy') }

  it { should be_allowed_action('ec2:Accept*') }
  it { should be_allowed_action('ec2:AdvertiseByoipCidr') }
  it { should be_allowed_action('ec2:AllocateAddress') }
  . . .
  #A
end
```

#A As with previous example, we will need to determine the actual permissions needed for our vpc pipeline. See complete example in code samples.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
<https://livebook.manning.com/#!/book/book-title/discussion>

Add the role definition along with the other roles already in the repo.

```
$ cat platform_vpc_role.tf
```

```
module "PlatformVPCRole" {
  source    = "terraform-aws-modules/iam/aws//modules/iam-assumable-role"
  version   = "5.55.0"
  create_role = true

  role_name          = "PlatformVPCRole"
  role_path          = "/PlatformRoles/"
  role_requires_mfa  = false

  custom_role_policy_arns    = [aws_iam_policy.PlatformVPCRolePolicy.arn]
  number_of_custom_role_policy_arns = 1
  trusted_role_arns = ["arn:aws:iam::${var.state_account_id}:root"]
}

# role permissions
resource "aws_iam_policy" "PlatformVPCRolePolicy" {
  name = "PlatformVPCRolePolicy"
  path = "/PlatformPolicies/"

  policy = jsonencode({
    "Version" : "2012-10-17"
    "Statement" : [
      {
        "Action" : [
          "ec2:Accept*",
          "ec2:AdvertiseByoipCidr",
          "ec2:AllocateAddress",
          "ec2:AllocateIpamPoolCidr"
          ...
        ]
        "Effect" : "Allow"
        "Resource" : "*"
      },
    ]
  })
}
```

Push these changes and deploy the new role to both accounts.

Back to our vpc pipeline. Since in our example, we can assume we have fully reserved the cidr ranges of our networks within the enterprise allocation system, the depth of our tests could be limited to the following.

```
# frozen_string_literal: true
require 'awspec'
require 'json'

tfvars = JSON.parse(File.read('./' + ENV['WORKSPACE'] + '.auto.tfvars.json'))

describe vpc(tfvars['cluster_name'] + '-vpc') do
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://livebook.manning.com/#!/book/book-title/discussion>

```

it { should exist }
it { should be_available }
its(:cidr_block) { should eq tfvars['vpc_cidr'] }
end

```

If it is possible that someone outside the team managing these networks could make changes to the networks, then these tests should be expanded to include other parts of the configuration such as subnets, routing tables, and nat gateways.

Like in our prior examples, having first created our tests, now is a good time to do the initial pipeline setup. Use the same starting lines from our prior terraform exercises.

We can also reuse the set-environment command from our previous pipeline. Although, beginning with this VPC pipeline, we are now switching from account-specific configuration to cluster-specific configuration so rename the parameter passed to set-environment to cluster.

Next, create the integration-test command.

commands:

...

integration-tests:

```

description: run awscli aws configuration tests
parameters:
  cluster:
    description: tf workspace name          #A
    type: string
  steps:
    - run:
      name: run awscli aws configuration tests
      Environment:
        WORKSPACE: <<parameters.cluster>>    #B
      command: bash scripts/run_awscli_integration_tests.sh << parameters.cluster >>

```

#A This is our cluster name which we also use to identify the terraform workspace.

#B Though we can pass the terraform workspace information as a regular parameter in the jobs provided by our terraform orb, notice how in the integration test we reference the cluster name via the WORKSPACE environment variable in order to fetch environment specific values.

The bash script for our integration test will be very much like our other pipelines.

```

#!/usr/bin/env bash
set -eo pipefail

source bash-functions.sh

export cluster_name=$1
export aws_account_id=$(jq -er .aws_account_id "$cluster_name".auto.tfvars.json)
export aws_assume_role=$(jq -er .aws_assume_role "$cluster_name".auto.tfvars.json)
export AWS_DEFAULT_REGION=$(jq -er .aws_region "$cluster_name".auto.tfvars.json)

awsAssumeRole "${aws_account_id}" "${aws_assume_role}"

```

```
rspec test/platform_vpc_spec.rb --format documentation
```

Add the integration test job.

jobs:

```
recurring-integration-tests:
  description: |
    Recurring job (weekly) to run pipeline integration tests to detect aws configuration drift
  docker:
    - image: *executor-image
  environment:
    TF_WORKSPACE: << parameters.cluster >>
  parameters:
    cluster:
      description: cluster configuration
      type: string
  steps:
    - checkout
    - setup_remote_docker
    - set-environment:
        cluster: << parameters.cluster >>
    - integration-tests:
        cluster: << parameters.cluster >>
```

Add a workflow job for the recurring integration test, but without the scheduled triggers so we can validate our tests now.

weekly integration test:

```
Jobs:
  - recurring-integration-tests:
      name: sbx-i01-aws-us-east-1 integration test
      context: *context
      cluster: sbx-i01-aws-us-east-1
  - recurring-integration-tests: #A
      name: prod-i01-aws-us-east-2 integration test
      context: *context
      cluster: prod-i01-aws-us-east-2
```

#A in the scheduled job we can run the jobs concurrently, though they are separate jobs with separate credentials needed.

The contents of the .env files for this pipeline are the same as for the platform-iam-profiles pipeline.

```
op.sbx-i01-aws-us-east-1.env  #A
op.prod-i01-aws-us-east-2.env #B
```

#A Same contents as op.nonprod.env from the profiles repo.

#B Same contents as op.prod.env from the profiles repo.

Push these changes and confirm that our test run correctly and fail. Once done, we can add in the workflow conditions to the recurring test.

weekly integration test:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
<https://livebook.manning.com/#!/book/book-title/discussion>

```

when:
  equal: [ scheduled_pipeline, << pipeline.trigger_source >> ]
jobs:

...

```

Now we can add our VPC configuration. We really only need a single resource definition using the terraform aws provider vpc module.

```
$ cat main.tf
```

```

module "vpc" {
  source = "terraform-aws-modules/vpc/aws"
  version = "5.21.0"
  name = "${var.cluster_name}-vpc"
  cidr = var.vpc_cidr
  azs = var.vpc_azs

  # private, node subnet
  private_subnets = var.vpc_private_subnets
  private_subnet_suffix = "private-subnet"
  private_subnet_tags = {
    "kubernetes.io/cluster/${var.cluster_name}" = "shared"
    "Tier" = "node" #A
    "karpenter.sh/discovery" = "${var.cluster_name}-vpc"
  }

  # public ingress subnet
  public_subnets = var.vpc_public_subnets
  public_subnet_suffix = "public-subnet"
  public_subnet_tags = {
    "kubernetes.io/cluster/${var.cluster_name}" = "shared"
    "kubernetes.io/role/elb" = "1" #B
    "Tier" = "public" #A
  }

  # intra, non-outbound route subnet
  intra_subnets = var.vpc_intra_subnets
  intra_subnet_suffix = "intra-subnet"
  intra_subnet_tags = {
    "kubernetes.io/cluster/${var.cluster_name}" = "shared"
    "Tier" = "intra" #A
  }

  # dedicated cluster database subnet
  database_subnets = var.vpc_database_subnets
  database_subnet_suffix = "database-subnet"
  database_subnet_tags = {
    "kubernetes.io/cluster/${var.cluster_name}" = "shared"
    "Tier" = "database" #A
  }

  create_database_subnet_group = true #C
  map_public_ip_on_launch = false
  enable_dns_hostnames = true

```



```

enable_dns_support    = true
enable_nat_gateway    = true                                #D
single_nat_gateway    = true                                #E
}

```

#A Apply a tag to identify the subnet so we can easily find it with terraform data resources in later pipelines.

#B Identify the Load Balancer network with this tag. Our service mesh will use this tag to decide where to place classic or network load balancers.

#C RDS and potentially other resources make use of a database subnet group. Where those kind of resources are provided by the Platform, network specific instances for the entire cluster will use the subnet group we define here.

#D This natgw support provided by this terraform module is for the public subnet.

#E We will only provision one, rather than one-per AZ.

The tfvar files for our vpcs contain the detailed values outlined in the chart included in the exercise description. And of course include the new role we created for this pipeline.

```
$ cat environments/sbx-i01-aws-us-east-1.auto.tfvars.json.tpl
```

```

{
  "cluster_name": "sbx-i01-aws-us-east-1",
  "aws_account_id": "{{ op://my-vault/aws-account-2/aws-account-id }}",
  "aws_assume_role": "PlatformRoles/PlatformVPCRole",
  "aws_region": "us-east-1",
  "vpc_cidr": "10.80.0.0/16",
  "vpc_azs": [
    "us-east-1a",
    "us-east-1b",
    "us-east-1c"
  ],
  "vpc_private_subnets": [
    "10.80.0.0/18",
    "10.80.64.0/18",
    "10.80.128.0/18"
  ],
  "vpc_intra_subnets": [
    "10.80.192.0/20",
    "10.80.208.0/20",
    "10.80.224.0/20"
  ],
  "vpc_database_subnets": [
    "10.80.240.0/23",
    "10.80.242.0/23",
    "10.80.244.0/23"
  ],
  "vpc_public_subnets": [
    "10.80.246.0/23",
    "10.80.248.0/23",
    "10.80.250.0/23"
  ]
}

```

```
]
}
```

```
$ cat environments/prod-i01-aws-us-east-2.auto.tfvars.json.tpl
```

```
{
  "cluster_name": "prod-i01-aws-us-east-2",
  "aws_account_id": "{{ op://my-vault/aws-account-1/aws-account-id }}",
  "aws_assume_role": "PlatformRoles/PlatformVPCRole",
  "aws_region": "us-east-2",
  "vpc_cidr": "10.90.0.0/16",
  "vpc_azs": [
    "Us-east-2a",
    "Us-east-2b",
    "Us-east-2c"
  ],
  "vpc_private_subnets": [
    "10.90.0.0/18",
    "10.90.64.0/18",
    "10.90.128.0/18"
  ],
  "vpc_intra_subnets": [
    "10.90.192.0/20",
    "10.90.208.0/20",
    "10.90.224.0/20"
  ],
  "vpc_database_subnets": [
    "10.90.240.0/23",
    "10.90.242.0/23",
    "10.90.244.0/23"
  ],
  "vpc_public_subnets": [
    "10.90.246.0/23",
    "10.90.248.0/23",
    "10.90.250.0/23"
  ]
}
```

In the variables.tf file there are some additional validations we can use for availability zones and cidr blocks.

```
variable "vpc_cidr" {
  type = string
  validation {
    condition = can(cidrhost(var.vpc_cidr, 32))
    error_message = "Invalid IPv4 CIDR"
  }
}
```

```
variable "vpc_azs" {
  description = "list of subnet AZs"
  type = list(string)
```

```
  validation {
    condition = alltrue([
```

```

    for v in var.vpc_azs : can(regex("[a-z][a-z]-[a-z]+-[1-9][a-c]", v))
  ])
  error_message = "Invalid VPC AZ name"
}

validation {
  condition    = length(var.vpc_azs) == 3
  error_message = "length of list(string) not equal to 3 "
}
}

variable "vpc_private_subnets" {
  description = "private node group subnet"
  type        = list(string)

  validation {
    condition = alltrue([
      for v in var.vpc_private_subnets : can(cidrhost(v, 32))
    ])
    error_message = "Invalid IPv4 CIDR"
  }

  validation {
    condition    = length(var.vpc_private_subnets) == 3
    error_message = "length of list(string) not equal to 3 "
  }
}

```

Add similar validations for the other cidr variables. The versions.tf file will follow the same pattern we have established in our other terraform pipelines.

With the resources defined, let's add the first stage (actions triggered by git-push) to our pipeline.

workflows:

```

deploy sbx-i01-aws-us-east-1 vpc:
  when:
    not:
      equal: [ scheduled_pipeline, << pipeline.trigger_source >> ]
  jobs:
    - terraform/static-analysis:
        name: static code analysis
        context: *context
        executor-image: *executor-image
        workspace: sbx-i01-aws-us-east-1          #A
        tflint-scan: true
        tflint-provider: aws
        trivy-scan: true
        before-static-analysis:
          - op/env:
              env-file: op.sbx-i01-aws-us-east-1.env
        filters: *on-push-main

    - terraform/plan:

```

```

name: sbx-i01-aws-us-east-1 change plan
context: *context
executor-image: *executor-image
workspace: sbx-i01-aws-us-east-1
before-plan:
  - set-environment:
      cluster: sbx-i01-aws-us-east-1
filters: *on-push-main

- approve sbx-i01-aws-us-east-1 changes:
  type: approval
  requires:
    - static code analysis
    - sbx-i01-aws-us-east-1 change plan
  filters: *on-push-main

- terraform/apply:
  name: apply sbx-i01-aws-us-east-1 changes
  context: *context
  workspace: sbx-i01-aws-us-east-1
  before-apply:
    - set-environment:
        cluster: sbx-i01-aws-us-east-1
  after-apply:
    - integration-tests:
        cluster: sbx-i01-aws-us-east-1
  requires:
    - approve sbx-i01-aws-us-east-1 changes
  filters: *on-push-main

```

In this exercise, we can use the following .trivyignore directives for what amount to false-positives from capabilities in the vpc module we aren't using.

```

AVD-AWS-0017
AVD-AWS-0057
AVD-AWS-0101
AVD-AWS-0102
AVD-AWS-0105
AVD-AWS-0178

```

Push these changes and confirm the tests validate that the sandbox vpc is successfully provisioned.

Now we can add the release workflow.

```

deploy prod-i01-aws-us-east-2 vpc:
  jobs:
    - terraform/plan:
        name: prod-i01-aws-us-east-2 change plan
        context: *context
        executor-image: *executor-image
        workspace: prod-i01-aws-us-east-2
        checkov-scan: true
        before-plan:
          - set-environment:
              cluster: prod-i01-aws-us-east-2

```

```

    cluster: prod-i01-aws-us-east-2
    filters: *on-tag-main

- approve prod-i01-aws-us-east-2 changes:
  type: approval
  requires:
    - prod-i01-aws-us-east-2 change plan
  filters: *on-tag-main

- terraform/apply:
  name: apply prod-i01-aws-us-east-2 changes
  context: *context
  workspace: prod-i01-aws-us-east-2
  before-apply:
    - set-environment:
        cluster: prod-i01-aws-us-east-2
  after-apply:
    - aws-integration-tests:
        cluster: prod-i01-aws-us-east-2
    - do/slack-bot:
        channel: lab-events
        message: Release aws-platform-vpc
        include-link: true
        include-tag: true
  requires:
    - approve prod-i01-aws-us-east-2 changes
  filters: *on-tag-main

- do/gh-release:
  name: generate release notes
  context: *context
  notes-from-file: release.md
  include-commit-msg: true
  before-release:
    - op/env:
        env-file: op.prod-i01-aws-us-east-2.env
  requires:
    - apply prod-i01-aws-us-east-2 changes
  filters: *on-tag-main

```

Push these changes and tag the commit to release to prod.

Finally, let's add the scheduled integration test workflow, and push the changes.

```

schedule weekly integration test:
  jobs:
    - do/schedule-pipeline:
        name: weekly integration test
        context: *context
        scheduled-pipeline-name: weekly-integration-test
        scheduled-pipeline-description: |
          Weekly, automated run of platform vpc integration tests
        hours-of-day: "[1]"
        days-of-week: "[\\"SUN\\"]"
        Before-schedule:

```

```
- op/env:
  env-file: op.prod-i01-aws-us-east-2.env
filters: *on-tag-main
```

Solutions to Exercise 7.3: Create a build and release pipeline for the control plane base

We start by adding a role for this pipeline to the pipeline_iam_profiles repo. Like in the previous pipeline exercise, add the test for the role.

```
$ cat test/platform_control_plan_base_role_spec.rb
```

```
require 'awspec'

describe iam_role('PlatformControlPlaneBaseRole') do
  it { should exist }
  it { should have_iam_policy('PlatformControlPlaneBaseRolePolicy') }

  it { should be_allowed_action('route53:AssociateVPCWithHostedZone') }
  it { should be_allowed_action('route53:ChangeResourceRecordSets') }
  it { should be_allowed_action('route53:ChangeTagsForResource') }
  . . . #A
end
```

#A As with the previous example, we will need to determine the actual permissions needed for our vpc pipeline. See complete example in code samples.

Add the role definition along with the other roles already in the repo.

```
$ cat platform_control_plane_base_role.tf
module "PlatformControlPlaneBaseRole" {
  source      = "terraform-aws-modules/iam/aws//modules/iam-assumable-role"
  version     = "5.55.0"
  create_role = true

  role_name           = "PlatformControlPlaneBaseRole"
  role_path           = "/Roles/"
  role_requires_mfa   = false
  custom_role_policy_arns = [
    aws_iam_policy.PlatformControlPlaneBaseRolePolicy.arn
  ]
  number_of_custom_role_policy_arns = 1

  trusted_role_arns = ["arn:aws:iam::${var.state_account_id}:root"]
}

# role permissions
resource "aws_iam_policy" "PlatformControlPlaneBaseRolePolicy" {
  name = "PlatformControlPlaneBaseRolePolicy"
  path = "/Policies/"

  policy = jsonencode({
    "Version" : "2012-10-17"
    "Statement" : [
      {
```

```

    "Action" : [
      "route53:AssociateVPCWithHostedZone",
      "route53:ChangeResourceRecordSets",
      "route53:ChangeTagsForResource",
      ...
    ]
    "Effect" : "Allow"
    "Resource" : "*"
  },
]
})
}

```

Push these changes and deploy the new role to both accounts.

Add the basic AWS resource tests from Chapter 7.

Start with the same, initial `.circleci/config.yml` pipeline shell:

```

---
version: 2.1

orbs:
  terraform: twdps/terraform@3.1.1
  op: twdps/onepassword@3.0.0
  do: twdps/pipeline-events@5.1.0

globals:
  - &context <my-team>
  - &executor-image twdps/circleci-infra-aws:alpine-2025.04

on-push-main: &on-push-main
  branches:
    only: /main/
  tags:
    ignore: /.*/

on-tag-main: &on-tag-main
  branches:
    ignore: /.*/
  tags:
    only: /.*/

```

Next, add the workflow section with the Terraform static analysis and plan jobs.

```

workflows:

  deploy sbx-i01-aws-us-east-1 control plane base:
    When: #A
    not:
      equal: [ scheduled_pipeline, << pipeline.trigger_source >> ]
    jobs:
      - terraform/static-analysis:
          name: static code analysis
          context: *context
          executor-image: *executor-image
          trivy-scan: true
          before-static-analysis:

```

```

- set-environment:
  cluster: sbx-i01-aws-us-east-1
  filters: *on-push-main

- terraform/plan:
  name: plan sbx-i01-aws-us-east-1 changes
  context: *context
  executor-image: *executor-image
  workspace: sbx-i01-aws-us-east-1
  tfc-workspace: aws-control-plane-base-sbx-i01-aws-us-east-1
  tfc-organization: epetech
  before-plan:
    - set-environment:
      cluster: sbx-i01-aws-us-east-1
      filters: *on-push-main

```

#A Like our prior pipelines, we will add a nightly scheduled job to run functional tests and watch for configuration drift, so we need to set this workflow to only run when triggered by git changes.

These first two jobs both use the set-environment command. Let's add this a commands section before the workflow section:

commands:

```

set-environment:
  description: generate environment credentials and configuration from templates
  parameters:
    cluster:
      description: cluster and tf workspace name
      type: string
  steps:
    - op/env:
        env-file: op.<< parameters.cluster >>.env      #A
    - op/tpl:
        tpl-path: environments
        tpl-file: << parameters.cluster >>.auto.tfvars.json  #B
    - terraform/terraformrc                                #C
    - do/bash-functions                                    #D

```

#A Uses the specified env file, populating with values from our secret store.

#B Generate the auto-included tfvars file for the specified environment, also populating with values from our secret store.

#C Uses the ENV value from our .env file to create the Terraform cloud credentials file in the expected location. Remember, everything on this runner is ephemeral and secure.

#D Write the shared bash functions available in the pipelin_events orb so we have access to them in our local bash scripts.

From the listings in chapter 7 up through the testing strategy section, we should have the following files in our aws-control-plane-base repo already (we referenced the tfvars file already):

```

environments/sbx-i01-aws-us-east-1.auto.tfvars.json.tpl
environments/prod-i01-aws-us-east-2.auto.tfvars.json.tpl

```

```

scripts/generate_kubeconfig.sh

```



```

scripts/base_configuration.sh
scripts/ebs_storage_class_tests.sh
scripts/efs_storage_class_tests.sh

test/control_plane_base.rb
test/baseline/cluster_addons.bats
test/baseline/baseline_resources.bats
test/ebs/dynamic-volume/dynamic-volume-test-pod.yaml
test/ebs/dynamic-volume/initial-pvc-test.bats
test/ebs/dynamic-volume/expanded-pvc-test.bats
test/efs/multi-write/multi-write-test-pods.yaml
test/efs/multi-write/multi-write-test.bats

tpl/karpenter_values.tpl
tpl/generate_kubeconfig.sh
tpl/base_configuration.sh
tpl/default-node-class.yaml
tpl/default-amd-node-pool.yaml
tpl/default-arm-node-pool.yaml

data.tf
main.tf
karpenter_deploy.tf
efs_csi_storage.tf

```

We can run the pipeline now and get feedback from the static analysis job and review the plan.

Let's add an approval step and the subsequent apply job to sandbox workflow.

- approve sbx-i01-aws-us-east-1 changes:
 - type: approval
 - requires:
 - static code analysis
 - plan sbx-i01-aws-us-east-1 changes
 - filters: *on-push-main
- terraform/apply:
 - name: apply sbx-i01-aws-us-east-1 changes
 - context: *context
 - executor-image: *executor-image
 - workspace: sbx-i01-aws-us-east-1
 - before-apply:
 - set-environment:
 - cluster: sbx-i01-aws-us-east-1
 - after-terraform-init:
 - replace-management-node-group-nodes:
 - cluster: sbx-i01-aws-us-east-1
 - after-apply:
 - base-configuration-validation:
 - cluster: sbx-i01-aws-us-east-1
 - requires:
 - approve sbx-i01-aws-us-east-1 changes
 - filters: *on-push-main

The apply job will call two additional commands in addition to set-environment. Once the Terraform apply job is finished we need to apply our base Kubernetes resources and then run our complete test suite. Let's add the base-configuration-validation command.

commands:

...

run-integration-tests:

description: run cluster configuration and functional tests

parameters:

cluster:

description: cluster and tf workspace name

type: string

steps:

- run:

name: run aws configuration checks

environment:

CLUSTER: << parameters.cluster >>

command: bash scripts/aws_integration_tests.sh << parameters.cluster >>

- run:

name: run cluster service smoke tests

command: bash scripts/eks_addons_tests.sh << parameters.cluster >>

- run:

name: run karpenter node pool functional tests

command: bash scripts/node_pool_tests.sh

- run:

name: run ebs storage class functional tests

command: bash scripts/ebs_storage_class_tests.sh << parameters.cluster >>

- run:

name: run efs storage class functional tests

command: bash scripts/efs_storage_class_tests.sh << parameters.cluster >>

base-configuration-validation:

parameters:

cluster:

description: cluster and tf workspace name

type: string

default: ""

steps:

- run:

name: generate kubeconfig

command: bash scripts/generate_kubeconfig.sh <<parameters.cluster>>

- run:

name: store cluster identifiers and apply baseline configuration

command: bash scripts/base_configuration.sh <<parameters.cluster>>

- op/env:

env-file: op.<< parameters.cluster >>.env

- run-integration-tests:

cluster: << parameters.cluster >>

We also want to incorporate the node upgrade process for our management node pool.

Upgrading the node pool is as simple as running the pipeline. Since our management node pool is an AWS managed node pool we just need to mark the pool a tainted for the node replacement and update to happen. The Apply job runs this taint after the Terraform init but before the apply step.

commands:

...

replace-management-node-group-nodes:

parameters:

cluster:

description: cluster and tf workspace name

type: string

default: ""

steps:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://livebook.manning.com/#!/book/book-title/discussion>

```

- run:
  name: taint eks-managed-nodegroup for complete refresh to latest AWS managed
EKS-optimized ami
  command: |
    export TAINT=$(jq -er .auto_refresh_management_node_group <<
parameters.cluster >>.auto.tfvars.json)
    if [[ "$TAINT" == "true" ]]; then
      terraform taint
"module.eks.module.eks_managed_node_group[\"management-arm-rkt-mng\"].aws_eks_no
de_group.this[0]"
    fi

```

Naturally, this won't work until the node pool exists so for the first run or two, comment out the command. After both the sbx and prod clusters have been provisioned then uncomment the commands for the taint and upgrade to start occurring.

At this point, we have the full git-push-triggered portion of our pipeline and can provision and fully test the cluster and baseline configuration functionality.

Once this stage of the pipeline is running smoothly, we can add the release workflow.

workflows:

...

release prod-i01-aws-us-east-2 control plane base:

jobs:

- terraform/plan:
 - name: plan prod-i01-aws-us-east-2 changes
 - context: *context
 - executor-image: *executor-image
 - workspace: prod-i01-aws-us-east-2
 - tfc-workspace: aws-control-plane-base-prod-i01-aws-us-east-2
 - tfc-organization: epetech
 - before-plan:
 - set-environment:
 - cluster: prod-i01-aws-us-east-2
 - filters: *on-tag-main
- approve prod-i01-aws-us-east-2 changes:
 - type: approval
 - requires:
 - plan prod-i01-aws-us-east-2 changes
 - filters: *on-tag-main
- terraform/apply:
 - name: apply prod-i01-aws-us-east-2 changes
 - context: *context
 - executor-image: *executor-image
 - workspace: prod-i01-aws-us-east-2
 - before-apply:
 - set-environment:
 - cluster: prod-i01-aws-us-east-2
 - after-terraform-init:
 - replace-management-node-group-nodes:
 - cluster: prod-i01-aws-us-east-2
 - after-apply:
 - base-configuration-validation:
 - cluster: prod-i01-aws-us-east-2
 - do/slack-bot:
 - channel: platform-events
 - message: Release aws-control-plane-base

```

      include-link: true
      include-tag: true
    requires:
      - approve prod-i01-aws-us-east-2 changes
    filters: *on-tag-main

- do/gh-release:
  name: generate release notes
  context: *context
  notes-from-file: release.md
  include-commit-msg: true
  before-release:
    - op/env:
        env-file: op.prod-i01-aws-us-east-2.env
  requires:
    - apply prod-i01-aws-us-east-2 changes
  filters: *on-tag-main

```

With this in place, once a change is successfully running in sandbox, we can release the changes to prod with the appropriate semantic git tag.

To finish this off, we just need to add the scheduled nightly job to run our test suite. Let's add the `schedule` job to our release workflow, along with the stand alone workflow that will actually run on our schedule.

```

- do/schedule-pipeline:
  name: schedule nightly integration tests
  context: *context
  scheduled-pipeline-name: nightly control plane base integration test
  scheduled-pipeline-description: |
    Automatically triggers nightly run of integration-test job
  hours-of-day: "[1]"
  days-of-week: "[\SUN\]"
  before-schedule:
    - op/env:
        env-file: op.prod-i01-aws-us-east-2.env
  filters: *on-tag-main

```

```

run nightly integration test:
  when:
    equal: [ scheduled_pipeline, << pipeline.trigger_source >> ]
  jobs:
    - integration-tests:
        name: nightly sbx-i01-aws-us-east-1 integration test
        cluster: sbx-i01-aws-us-east-1
        context: *context
    - integration-tests:
        name: nightly prod-i01-aws-us-east-2 integration test
        cluster: prod-i01-aws-us-east-2
        context: *context

```

And we need to add the nightly job definition just below our commands section.

jobs:

```

integration-tests:
  docker:
    - image: *executor-image
  parameters:
    cluster:
      description: cluster and tf workspace name

```

```
  type: string
steps:
- checkout
- set-environment:
  cluster: << parameters.cluster >>
- kube/op-config: #A
  op-value: my-vault/<< parameters.cluster>>/kubeconfig-base64
- run-integration-tests:
  cluster: << parameters.cluster >>
```

#A The primary pipeline jobs generate and save the kubeconfig file, but our nightly job can just use this saved value. Here we are using a command from the kube-ops orb that is defined to take a reference to where we store this value and use it to setup the kubeconfig file in the expected location.

Chapter 8 Solutions

Solution to Exercise 8.2 Run Trivy scan on metrics-server chart and create a values.yaml to correct the findings

(insert here a copy of the results of a trivy scan of the community metrics-server chart)

Customizations to add to the metrics-server deployment:

```
image:
  pullPolicy: Always

securityContext:
  allowPrivilegeEscalation: false
  readOnlyRootFilesystem: true
  runAsNonRoot: true
  runAsUser: 65534
  runAsGroup: 65534
  seccompProfile:
    type: RuntimeDefault

updateStrategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 1

podDisruptionBudget:
  enabled: true
  maxUnavailable: 1

resources:
  requests:
    cpu: 100m
    memory: 200Mi
  limits:
    cpu: 200m
    memory: 250Mi
```

Solution to Exercise 8.3: Create the set-environment command for our control-plane-services pipeline.

commands:

```
set-environment:
  description: generate environment credentials
  parameters:
    cluster:
      description: cluster and tf workspace name
      type: string
  steps:
    - op/env:
        env-file: op.<< parameters.cluster >>.env
    - kube/op-config:
        op-value: platform/<< parameters.cluster>>/kubeconfig-base64
    - do/bash-functions
```

Exercise 8.4 Add the necessary steps to the control plane services deployment job for our pipeline

commands:

...

```
run-integration-tests:
  steps:
    - run:
        name: run control plane services integration tests
        command: bash scripts/services_integration_test.sh
```

jobs:

```
deploy control plane services:
  docker:
    - image: *executor-image
  parameters:
    cluster:
      description: cluster name
      type: string
  steps:
    - checkout
    - set-environment:
        cluster: << parameters.cluster >>
    - run:
        name: install metrics-server
        command: bash scripts/install_metrics_server.sh << parameters.cluster >>
    - run:
        name: install kube-state-metrics
        command: bash scripts/install_kube_state_metrics.sh << parameters.cluster >>
    - run:
        name: install kubernetes-event-exporter
        command: bash scripts/install_kubernetes_event_exporter.sh << parameters.cluster >>
  >>
  - run-integration-tests
```

The metrics-server install script was provided in chapter 8. Here are the additional example install scripts:

```
# install_kube_state_metrics.sh
#!/usr/bin/env bash
set -eo pipefail
source bash-functions.sh
```

```
cluster_name=$1
CHART_VERSION=$(jq -er .kube_state_metrics_chart_version
environments/$cluster_name.json)
echo "kube-state-metrics chart version $CHART_VERSION"
```

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo update
```

```
# perform trivy scan of chart with install configuration
trivyScan "prometheus-community/kube-state-metrics" "kube-state-metrics"
"$CHART_VERSION" "kube-state-metrics-values/$cluster_name-values.yaml"
```

```
helm upgrade --install kube-state-metrics prometheus-community/kube-state-metrics \
  --version $CHART_VERSION \
  --namespace kube-system \
  --values kube-state-metrics-values/$cluster_name-values.yaml
```

```
# install_kubernetes_event_exporter.sh
#!/usr/bin/env bash
set -eo pipefail
source bash-functions.sh
```

```
cluster_name=$1
CHART_VERSION=$(jq -er .kubernetes_event_exporter_chart_version
environments/$cluster_name.json)
echo "kubernetes-event-exporter chart version $CHART_VERSION"
```

```
# perform trivy scan of chart with install configuration
trivyScan "oci://registry-1.docker.io/bitnamicharts/kubernetes-event-exporter"
"kubernetes-event-exporter" "$CHART_VERSION"
"kubernetes-event-exporter-values/$cluster_name-values.yaml"
```

```
helm upgrade --install event-exporter
oci://registry-1.docker.io/bitnamicharts/kubernetes-event-exporter \
  --version $CHART_VERSION \
  --namespace kube-system \
  --values kubernetes-event-exporter-values/$cluster_name-values.yaml
```

Examples for the Helm values.yaml used in this scripts. These are for the sbx environment. For these exercises you could use the same values in prod.

```
# metrics-server-values/sbx-i01-aws-us-east-1-values.yaml
image:
  pullPolicy: Always
```

```
securityContext:
  allowPrivilegeEscalation: false
  readOnlyRootFilesystem: true
  runAsNonRoot: true
  runAsUser: 65534
  runAsGroup: 65534
  seccompProfile:
    type: RuntimeDefault
```

```
updateStrategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 1
```

```
podDisruptionBudget:
  enabled: true
  maxUnavailable: 1
```

```
service:
  labels:
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "metrics-server"
```

```
resources:
  requests:
    cpu: 100m
```



```

    memory: 200Mi
  limits:
    cpu: 200m
    memory: 250Mi

nodeSelector:
  nodegroup: management-arm-rkt-mng

tolerations:
- key: "dedicated"
  operator: "Equal"
  value: "management"
  effect: "NoSchedule"

# kube-state-metrics-values/sbx-i01-aws-us-east-1-values.yaml
image:
  pullPolicy: Always

customLabels:
  kubernetes.io/cluster-service: "true"
  kubernetes.io/name: "kube-state-metrics"

containerSecurityContext:
  runAsGroup: 65534
  runAsUser: 65534

# Available collectors for kube-state-metrics.
# By default, all available resources are enabled, comment out to disable.
collectors:
- certificatesigningrequests
- configmaps
- cronjobs
- daemonsets
- deployments
- endpoints
- horizontalpodautoscalers
- ingresses
- jobs
- leases
- limitranges
- mutatingwebhookconfigurations
- namespaces
- networkpolicies
- nodes
- persistentvolumeclaims
- persistentvolumes
- poddisruptionbudgets
- pods
- replicaset
- replicationcontrollers
- resourcequotas
# - secrets
- services
- statefulsets
- storageclasses
- validatingwebhookconfigurations
- volumeattachments

nodeSelector:
  nodegroup: management-arm-rkt-mng

```

```

tolerations:
  - key: "dedicated"
    operator: "Equal"
    value: "management"
    effect: "NoSchedule"

resources:
  limits:
    cpu: 100m
    memory: 64Mi
  requests:
    cpu: 10m
    memory: 32Mi

# kubernetes-event-exporter-values/sbx-i01-aws-us-east-1-values.yaml
nameOverride: event-exporter

image:
  pullPolicy: Always

rbac:
  ## @param rbac.create Create the RBAC roles for API accessibility
  ##
  create: true
  ## @param rbac.rules [array] List of rules for the cluster role
  ##
  rules:
    - apiGroups: ["*"]
      resources:
        - certificatesigningrequests
        - configmaps
        - cronjobs
        - daemonsets
        - deployments
        - endpoints
        - events
        - horizontalpodautoscalers
        - ingresses
        - jobs
        - leases
        - limitranges
        - mutatingwebhookconfigurations
        - namespaces
        - networkpolicies
        - nodes
        - nodeclaims
        - persistentvolumeclaims
        - persistentvolumes
        - poddisruptionbudgets
        - pods
        - replicaset
        - replicationcontrollers
        - resourcequotas
        - services
        - statefulsets
        - storageclasses
        - validatingwebhookconfigurations
        - volumeattachments
      verbs: ["get", "watch", "list"]

pdb:

```

```

create: true
minAvailable: "0"

containerSecurityContext:
  enabled: true
  runAsUser: 65534
  runAsGroup: 65534
  runAsNonRoot: true
  privileged: false
  readOnlyRootFilesystem: true
  allowPrivilegeEscalation: false
  capabilities:
    drop: ["ALL"]
  seccompProfile:
    type: "RuntimeDefault"

resourcesPreset: "nano"

nodeSelector:
  nodegroup: management-arm-rkt-mng

tolerations:
- key: "dedicated"
  operator: "Equal"
  value: "management"
  effect: "NoSchedule"

config:
  metricsNamePrefix: "event_exporter_"

```

Solution to Exercise 8.5: Add the release stage to our control-plane-base pipeline

Add the release and nightly test workflows to the aws-control-plane-base pipeline.

Jobs:

...

```

integration-tests:
  docker:
    - image: *executor-image
  parameters:
    cluster:
      description: cluster name
      type: string
  steps:
    - checkout
    - set-environment:
        cluster: << parameters.cluster >>
    - run-integration-tests

```

workflows:

...

```

release prod-i01-aws-us-east-2 control plane base:
  jobs:
    - deploy control plane services:
        name: deploy prod-i01-aws-us-east-2 control plane services
        context: *context
        cluster: prod-i01-aws-us-east-2
        filters: *on-tag-main

```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://livebook.manning.com/#!/book/book-title/discussion>

```

- do/gh-release:
  name: generate release notes
  context: *context
  notes-from-file: release.md
  include-commit-msg: true
  before-release:
    - op/env:
        env-file: op.prod-i01-aws-us-east-2.env
  after-release:
    - do/slack-bot:
        channel: lab-events
        message: Release aws-control-plane-services
        include-link: true
        include-tag: true
  requires:
    - deploy prod-i01-aws-us-east-2 control plane services
  filters: *on-tag-main

- do/schedule-pipeline:
  name: schedule nightly integration tests
  context: *context
  scheduled-pipeline-name: nightly control plane services integration tests
  scheduled-pipeline-description: |
    Automatically triggers nightly run of control plane services integration tests
  hours-of-day: "[1]"
  days-of-week: "[\SUN\]"
  before-schedule:
    - op/env:
        env-file: op.prod-i01-aws-us-east-2.env
  filters: *on-tag-main

run nightly integration tests:
when:
  equal: [ scheduled_pipeline, << pipeline.trigger_source >> ]
jobs:
  - integration-tests:
      name: nightly sbx-i01-aws-us-east-1 integration tests
      context: *context
      cluster: sbx-i01-aws-us-east-1
  - integration-tests:
      name: nightly prod-i01-aws-us-east-2 integration tests
      context: *context
      cluster: prod-i01-aws-us-east-2

```

Solution to Exercise 8.6: Complete the definition of our pipeline trigger filters in these anchors.

```

on-push-main: &on-push-main
branches:
  only: /main/
tags:
  ignore: /.*/

on-tag-main: &on-tag-main
branches:
  ignore: /.*/
tags:
  only: /.*/

```

Exercise 8.7 Complete the steps we will need for the set-environment command in our extensions pipeline.

commands:

```
set-environment:
  description: generate environment credentials and configuration from templates
  parameters:
    cluster:
      description: cluster and tf workspace name
      type: string
  steps:
    - op/env:
        env-file: op.<< parameters.cluster >>.env
    - op/tpl:
        tpl-path: environments
        tpl-file: << parameters.cluster >>.auto.tfvars.json
    - kube/op-config:
        op-value: platform/<< parameters.cluster >>/kubeconfig-base64
    - terraform/terraformrc
    - do/bash-functions
```

Exercise 8.8: Create AwsSepc test to confirm roles

```
$ cat test/service_account_iam_roles.rb
require 'awspec'
require 'json'

tfvars = JSON.parse(File.read('./' + ENV['WORKSPACE'] + '.auto.tfvars.json'))

describe iam_role(tfvars['cluster_name'] + '-cert-manager-sa') do
  it { should exist }
end

describe iam_role(tfvars['cluster_name'] + '-external-dns-sa') do
  it { should exist }
end
```

Exercise 8.9 Create deployment scripts and values files for cert-manager and external-dns deployments.

```
$ cat cert-manager-values/sbx-i01-aws-us-east-1-values.yaml
global:
  commonLabels:
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "cert-manager"
crds:
  enabled: true
  keep: true
replicaCount: 1
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 1
podDisruptionBudget:
  enabled: true
  minAvailable: 1
image:
  pullPolicy: Always
resources:
  requests:
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
<https://livebook.manning.com/#!/book/book-title/discussion>

```

    cpu: 100m
    memory: 100Mi
  limits:
    cpu: 150m
    memory: 150Mi
  nodeSelector:
    nodegroup: management-arm-rkt-mng
  tolerations:
    - key: "dedicated"
      operator: "Equal"
      value: "management"
      effect: "NoSchedule"
  webhook:
    replicaCount: 1
    strategy:
      type: RollingUpdate
      rollingUpdate:
        maxSurge: 0
        maxUnavailable: 1
  resources:
    requests:
      cpu: 100m
      memory: 100Mi
    limits:
      cpu: 150m
      memory: 150Mi
  podDisruptionBudget:
    enabled: true
    minAvailable: 1
  nodeSelector:
    nodegroup: management-arm-rkt-mng
  tolerations:
    - key: "dedicated"
      operator: "Equal"
      value: "management"
      effect: "NoSchedule"
  image:
    pullPolicy: Always
  cainjector:
    enabled: true
    replicaCount: 1
    strategy:
      type: RollingUpdate
      rollingUpdate:
        maxSurge: 0
        maxUnavailable: 1
  podDisruptionBudget:
    enabled: true
    minAvailable: 1
  resources:
    requests:
      cpu: 100m
      memory: 100Mi
    limits:
      cpu: 150m
      memory: 150Mi
  nodeSelector:
    nodegroup: management-arm-rkt-mng
  tolerations:
    - key: "dedicated"
      operator: "Equal"
      value: "management"

```

```

    effect: "NoSchedule"
  image:
    pullPolicy: Always
  acmesolver:
    image:
      pullPolicy: Always
  startupapicheck:
    enabled: true
  resources:
    requests:
      cpu: 100m
      memory: 100Mi
    limits:
      cpu: 150m
      memory: 150Mi
  nodeSelector:
    nodegroup: management-arm-rkt-mng
  tolerations:
    - key: "dedicated"
      operator: "Equal"
      value: "management"
      effect: "NoSchedule"
  image:
    pullPolicy: Always

```

```

$ cat scripts/install_cert_manager.sh
#!/usr/bin/env bash
set -eo pipefail
source bash-functions.sh

```

```

cluster_name=$1
chart_version=$(jq -er .cert_manager_chart_version $cluster_name.auto.tfvars.json)
AWS_ACCOUNT_ID=$(jq -er .aws_account_id $cluster_name.auto.tfvars.json)

```

```

echo "cert-manager chart version $chart_version"
helm repo add jetstack https://charts.jetstack.io --force-update
helm repo update

```

```

# perform trivy scan of chart with install configuration
trivyScan "jetstack/cert-manager" "cert-manager" "v$chart_version"
"cert-manager-values/$cluster_name-values.yaml"

```

```

helm upgrade --install cert-manager jetstack/cert-manager \
  --version v$chart_version \
  --namespace cert-manager --create-namespace \
  --set
serviceAccount.annotations."eks\.amazonaws\.com/role-arn"=arn:aws:iam::${AWS_ACCOUNT_ID}:role/PlatformRoles/${cluster_name}-cert-manager-sa \
  --values cert-manager-values/$cluster_name-values.yaml
sleep 180

```

```

$ cat external-dns-values/sbx-i01-aws-us-east-1-values.yaml
image:
  repository: registry.k8s.io/external-dns/external-dns
  pullPolicy: Always
serviceAccount:
  create: true
rbac:
  create: true
deploymentStrategy:
  type: Recreate

```

```

podSecurityContext:
  runAsNonRoot: true
  fsGroup: 65534
  seccompProfile:
    type: RuntimeDefault
securityContext:
  privileged: false
  allowPrivilegeEscalation: false
  readOnlyRootFilesystem: true
  runAsNonRoot: true
  runAsUser: 65532
  capabilities:
    drop: ["ALL"]
livenessProbe:
  httpGet:
    path: /healthz
    port: http
  initialDelaySeconds: 10
  periodSeconds: 10
  timeoutSeconds: 5
  failureThreshold: 2
  successThreshold: 1
readinessProbe:
  httpGet:
    path: /healthz
    port: http
  initialDelaySeconds: 5
  periodSeconds: 10
  timeoutSeconds: 5
  failureThreshold: 6
  successThreshold: 1
resources:
  requests:
    cpu: 100m
    memory: 128Mi
  limits:
    cpu: 1000m
    memory: 1024Mi
nodeSelector:
nodegroup: management-arm-rkt-mng
tolerations:
- key: "dedicated"
  operator: "Equal"
  value: "management"
  effect: "NoSchedule"
serviceMonitor:
  enabled: false
logLevel: info
logFormat: json
interval: 1m
policy: sync
sources:
- service
- ingress
- istio-gateway
- istio-virtualservice
extraArgs:
- --aws-zone-type=public
registry: txt
txtOwnerId:
txtPrefix:
txtSuffix:

```



```

excludeDomains: []
provider:
  name: aws
  webhook:
    image:
      pullPolicy: Always
  livenessProbe:
    httpGet:
      path: /healthz
      port: http-webhook
    initialDelaySeconds: 10
    periodSeconds: 10
    timeoutSeconds: 5
    failureThreshold: 2
    successThreshold: 1
  readinessProbe:
    httpGet:
      path: /healthz
      port: http-webhook
    initialDelaySeconds: 5
    periodSeconds: 10
    timeoutSeconds: 5
    failureThreshold: 6
    successThreshold: 1
  serviceMonitor:
    interval:
    scheme:
    tlsConfig: {}
    bearerTokenFile:
    scrapeTimeout:
    metricRelabelings: []
    relabelings: []

```

```

$ cat scripts/install_external-dns.sh
#!/usr/bin/env bash
set -eo pipefail
source bash-functions.sh

```

```

cluster_name=$1
chart_version=$(jq -er .external_dns_chart_version $cluster_name.auto.tfvars.json)
cluster_domains=$(jq -er .cluster_domains $cluster_name.auto.tfvars.json)
AWS_ACCOUNT_ID=$(jq -er .aws_account_id $cluster_name.auto.tfvars.json)

```

```

echo "external-dns chart version $chart_version"

```

```

# add domains to external-dns domainFilter
declare -a domains=( $(echo $cluster_domains | jq -r '.[ ]') )
cat <<EOF > cluster-domains-values.yaml
domainFilters:
EOF

```

```

for domain in "${domains[@]}";
do
  echo "  - $domain" >> cluster-domains-values.yaml
done

```

```

helm repo add external-dns https://kubernetes-sigs.github.io/external-dns/
helm repo update

```

```

# perform trivy scan of chart with install configuration
trivyScan "external-dns/external-dns" "external-dns" "$chart_version"

```

```
"external-dns-values/$cluster_name-values.yaml"
```

```
helm upgrade --install external-dns external-dns/external-dns \
  --version v$chart_version \
  --namespace istio-system \
  --set
serviceAccount.annotations."eks\.amazonaws\.com/role-arn"=arn:aws:iam::${AWS_ACCOUNT_ID}:role/PlatformRoles/${cluster_name}-external-dns-sa \
  --set txtOwnerId=${cluster_name}-epetech \
  --values cluster-domains-values.yaml \
  --values external-dns-values/$cluster_name-values.yaml
```

Exercise 8.10 Create state tests using Bats to confirm the extensions report a healthy status.

```
$ cat test/services-state-tests.bat
#!/usr/bin/env bats
```

```
@test "istiod pod status is Running" {
  run bash -c "kubectl get pods -n istio-system -o wide | grep 'istiod'"
  [[ "${output}" =~ "Running" ]]
}

@test "istio-ingressgateway pod status is Running" {
  run bash -c "kubectl get pods -n istio-system -o wide | grep 'ingressgateway'"
  [[ "${output}" =~ "Running" ]]
}

@test "istio-cni-node pod status is Running" {
  run bash -c "kubectl get pods -n istio-system -o wide | grep 'istio-cni-node'"
  [[ "${output}" =~ "Running" ]]
}

@test "cert-manager pod status is Running" {
  run bash -c "kubectl get pods -n cert-manager -o wide | grep 'cert-manager'"
  [[ "${output}" =~ "Running" ]]
}

@test "cert-manager-cainjector pod status is Running" {
  run bash -c "kubectl get pods -n cert-manager -o wide | grep 'cert-manager-cainjector'"
  [[ "${output}" =~ "Running" ]]
}

@test "cert-manager-webhook pod status is Running" {
  run bash -c "kubectl get pods -n cert-manager -o wide | grep 'cert-manager-webhook'"
  [[ "${output}" =~ "Running" ]]
}

@test "external-dns pod status is Running" {
  run bash -c "kubectl get pods -n istio-system -o wide | grep 'external-dns'"
  [[ "${output}" =~ "Running" ]]
}
```

Exercise 8.11 Create a script that uses the Httpbin deployment to test our extensions.

```

$ cat scripts/extensions_functional_tests.sh

#!/usr/bin/env bash
set -eo pipefail

export cluster_name=$1

bash scripts/toggle_httpbin.sh on $cluster_name

echo "test successful ingress, certs, dns management"
jsonResponse=$(curl -X GET "https://httpbin.$cluster_name.epetech.io/json" -H "accept:
application/json")
echo "response $jsonResponse"
if [[ ! $jsonResponse =~ "slideshow" ]]; then
    echo "httpbin not responding"
    bash scripts/toggle_httpbin.sh off $cluster_name
    exit 1
fi

bash scripts/toggle_httpbin.sh off $cluster_name

Listing 8.17 should be in our test/httpbin folder. We can use a script to toggle it on or off.

$ cat scripts/toggle_httpbin.sh
#!/usr/bin/env bash
set -eo pipefail

export toggle=$1
export cluster_name=$2

node_count () {          #A
    nodes=$(kubectl get nodes -l kubernetes.io/arch=amd64 | tail -n +2 | wc -l | xargs)
    echo "current node count $nodes"
}

echo "toggle $toggle httpbin test instance on $cluster_name"

cat <<EOF > test/httpbin/virtual-service.yaml    #B
---
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: httpbin
  namespace: default-mtls
spec:
  hosts:
  - "httpbin.$cluster_name.epetech.io"
  gateways:
  - istio-system/$cluster_name-epetech-io-gateway
  http:
  - route:
    - destination:
        host: httpbin.default-mtls.svc.cluster.local
        port:
            number: 80
EOF

if [[ $toggle == "on" ]]; then          #C
    echo "deploy httpbin to default-mtls"
    node_count

    kubectl apply -f test/httpbin --recursive
    sleep 180

```

```

    node_count
fi

if [[ $toggle == "off" ]]; then          #D
    echo "delete httpbin from default-mtls"

    kubectl delete -f test/httpbin --recursive
fi

```

#A Let's add a bit of debugging information. This function will display the current node count for the Karpenter managed default AMD node pool.

#B Our VirtualService definition will use specific cluster information to reference our default gateway. Since we want to use this test for all the clusters we manage, we should generate the contents of these resource at runtime using a heredoc template.

#C We will pass "on" to deploy the app. Adding some sleep time for a new node to be provisioned will avoid a race condition for our tests.

#D We can pass "off" to remove the deployment before exiting our script.

During initial creation of our platform there won't be anything running on our Karpenter managed nodes when we first deploy our extensions. Showing the count before and after the deployment helps us catch the circumstances where a node is needed but fails to provision. Later on there will be nodes, but as Karpenter will optimize the pool, it isn't unusual for the *next* deployment to cause a new node to be provisioned so this can still be useful.

Solution for confirming TLSv1.3.

```

output=$(curl -Iiv --tlsv1.1 "https://httpbin.$cluster_name.epetech.io" 2>&1)
echo "testing TLSv1.3 uplevel with 1.1 communication attempt"
if [[ ! $output =~ "SSL connection using TLSv1.3" ]]; then
    echo "TLSv1.3 not enforced"
    bash scripts/toggle_httpbin.sh off $cluster_name
    exit 1
fi

```

Hint: For making the toggle-off function call DRY, could the bash trap statement be used to delete the deployment regardless of success or failure of the tests?

Exercise 8.12 Add the release workflow to the aws-control-plane-extensions pipeline.

jobs:

...

integration tests:

```

docker:
  - image: *executor-image
parameters:
  cluster:
    description: cluster name
    type: string
steps:
  - checkout
  - set-environment:
    cluster: << parameters.cluster >>
  - run-integration-tests:

```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://livebook.manning.com/#!/book/book-title/discussion>

```
cluster: << parameters.cluster >>
```

workflows:

deploy sbx-i01-aws-us-east-1 control plane extensions:

```
when:
  not:
    equal: [ scheduled_pipeline, << pipeline.trigger_source >> ]
jobs:
  ...
```

release prod-i01-aws-us-east-2 control plane extensions:

```
jobs:
  - terraform/apply:
      name: apply prod-i01-aws-us-east-2 service account roles
      context: *context
      workspace: prod-i01-aws-us-east-2
      before-apply:
        - set-environment:
            cluster: prod-i01-aws-us-east-2
      after-apply:
        - validate-service-account-roles:
            cluster: prod-i01-aws-us-east-2
      filters: *on-tag-main

  - deploy control plane extensions:
      name: deploy prod-i01-aws-us-east-2 extensions
      context: *context
      cluster: prod-i01-aws-us-east-2
      requires:
        - apply prod-i01-aws-us-east-2 service account roles
      filters: *on-tag-main

  - do/gh-release:
      name: generate release notes
      context: *context
      notes-from-file: release.md
      include-commit-msg: true
      before-release:
        - op/env:
            env-file: op.prod-i01-aws-us-east-2.env
      after-release:
        - do/slack-bot:
            channel: platform-events
            message: Release aws-control-plane-extensions
            include-link: true
            include-tag: true
      requires:
        - deploy prod-i01-aws-us-east-2 extensions
      filters: *on-tag-main

  - do/schedule-pipeline:
      name: schedule nightly integration tests
      context: *context
      scheduled-pipeline-name: nightly control plane extensions integration tests
      scheduled-pipeline-description: |
        Automatically triggers nightly run of control plane extensions integration tests
      hours-of-day: "[1]"
      days-of-week: "[\"SUN\"]"
      before-schedule:
        - op/env:
            env-file: op.prod-i01-aws-us-east-2.env
```

```
filters: *on-tag-main

run nightly integration tests:
  when:
    equal: [ scheduled_pipeline, << pipeline.trigger_source >> ]
  jobs:
    - integration tests:
        name: nightly sbx-i01-aws-us-east-1 integration tests
        context: *context
        cluster: sbx-i01-aws-us-east-1
    - integration tests:
        name: nightly prod-i01-aws-us-east-2 integration tests
        context: *context
        cluster: prod-i01-aws-us-east-2
```