The `admission_whitelist_patterns` section defines container image patterns permitted to be deployed without attestations. This includes patterns for Google-managed system images, the Argo CD registry, the External Secrets registry, and the Redis container used by Argo CD.

The `defaultAdmissionRule` section mandates attestation using the attestor we created. Therefore, any other images would require attestation to run on the cluster:

```
resource "google_binary_authorization_policy" "policy" {
  count = var.branch == "dev" ? 1 : 0
  admission_whitelist_patterns {
    name_pattern = "gcr.io/google_containers/*"...
    name_pattern = "gcr.io/google-containers/*"...
    name_pattern = "k8s.gcr.io/**"...
    name_pattern = "gke.gcr.io/**"...
    name_pattern = "gcr.io/stackdriver-agents/*"...
    name_pattern = "quay.io/argoproj/*"...
    name_pattern = "ghcr.io/dexidp/*"...
    name_pattern = "docker.io/redis[@:]*"...
    name_pattern = "ghcr.io/external-secrets/*"
  }
  global_policy_evaluation_mode = "ENABLE"
  default_admission_rule {
    evaluation_mode  = "REQUIRE_ATTESTATION"
    enforcement_mode = "ENFORCED_BLOCK_AND_AUDIT_LOG"
    require_attestations_by = [
      module.qa-attestor[0].attestor
    ]
  }
}
```

To enforce the binary authorization policy within a cluster, we must also enable binary authorization. To do so, we add the following block within the `cluster.tf` file:

```
resource "google_container_cluster" "main" {
...
  dynamic "binary_authorization" {
    for_each = var.branch == "prod" ? [1] : []
    content {
      evaluation_mode = "PROJECT_SINGLETON_POLICY_ENFORCE"
    }
  }
...
}
```

This dynamic block is created exclusively when the branch name is `prod`. The reason for this approach is our intention to deploy our code to the Dev environment without image attestation, conduct testing, and then attest the images if the tests succeed. Therefore, only the Prod cluster should disallow unattested images. To achieve this, we will include the following steps in the Dev CD workflow:

```
  binary-auth:
    name: Attest Images
    needs: [run-tests]
    uses: ./.github/workflows/attest-images.yml
    secrets: inherit
```

As you can see, this calls the `attest-images.yml` workflow. Let's look at that now:

```
...
    steps:
    - uses: actions/checkout@v2
    - id: gcloud-auth ...
    - name: Set up Cloud SDK ...
    - name: Install gcloud beta
      id: install-gcloud-beta
      run: gcloud components install beta
    - name: Attest Images
      run: |
        for image in $(cat ./images); do
          no_of_slash=$(echo $image | tr -cd '/' | wc -c)
          prefix=""
          if [ $no_of_slash -eq 1 ]; then
            prefix="docker.io/"
          fi
          if [ $no_of_slash -eq 0 ]; then
            prefix="docker.io/library/"
          fi
          image_to_attest=$image
          if [[ $image =~ "@" ]]; then
            echo "Image $image has DIGEST"
            image_to_attest="${prefix}${image}"
          else
            echo "All images should be in the SHA256 digest format"
            exit 1
          fi
          echo "Processing $image"
          attestation_present=$(gcloud beta container binauthz attestations list
--attestor-project="${{ secrets.PROJECT_ID }}" --attestor="${{ secrets.ATTESTOR_NAME }}"
--artifact-url="${image_to_attest}")
          if [ -z "${attestation_present// }" ]; then
            gcloud beta container binauthz attestations sign-and-create --artifact-
url="${image_to_attest}" --attestor="${{ secrets.ATTESTOR_NAME }}" --attestor-project="${{
secrets.PROJECT_ID }}" --keyversion-project="${{ secrets.PROJECT_ID }}" --keyversion-
location="${{ secrets.KMS_KEY_LOCATION }}" --keyversion-keyring="${{ secrets.KMS_KEYRING_
NAME }}" --keyversion-key="${{ secrets.KMS_KEY_NAME }}" --keyversion="${{ secrets.KMS_KEY_
VERSION }}"
```

```
          fi
        done
```

The YAML file performs several tasks, including the installation of `gcloud` and authentication with GCP. It also installs the `gcloud beta` CLI and, importantly, attests images.

To attest images, it searches the `blog-app.yaml` manifest for all images. For each image, it checks whether the image is in the `sha256` digest format. If yes, it proceeds to attest the image.

It's worth noting that the workflow verifies that images are specified using a `sha256` digest format rather than a tag in the image definition. This choice is crucial when working with binary authorization. Why? Because binary authorization requires deploying images with their `sha256` digest instead of a tag. This precaution is essential because, with tags, anyone can associate a different image with the same tag as the attested image and push it to the container registry. In contrast, a digest is a hash generated from a Docker image. Therefore, as long as the image's content remains unchanged, the digest remains the same. This prevents any attempts to bypass binary authorization controls.

The format for specifying images in this manner is as follows:

```
<repo_url>/<image_name>@sha256:<sha256-digest>
```

Therefore, before pushing the changes to the remote repository, let's replace the image tags with `sha256` digests. Use the following commands to do so:

```
$ grep -ir "image:" ./manifests/blog-app |\
 awk {'print $3'} | sort -t: -u -k1,1 > ./images
$ for image in $(cat ./images); do
  no_of_slash=$(echo $image | tr -cd '/' | wc -c)
  prefix=""
  if [ $no_of_slash -eq 1 ]; then
    prefix="docker.io/"
  fi
  if [ $no_of_slash -eq 0 ]; then
    prefix="docker.io/library/"
  fi
  image_to_attest=$image
  if [[ $image =~ "@" ]]; then
    echo "Image $image has DIGEST"
    image_to_attest="${prefix}${image}"
  else
    DIGEST=$(docker pull $image | grep Digest | awk {'print $2'})
    image_name=$(echo $image | awk -F ':' {'print $1'})
    image_to_attest="${prefix}${image_name}@${DIGEST}"
  fi
  escaped_image=$(printf '%s\n' "${image}" | sed -e 's/[]\/$*.^[]/\\&/g')
  escaped_image_to_attest=$(printf '%s\n' "${image_to_attest}" | \
 sed -e 's/[]\/$*.^[]/\\&/g')
  echo "Processing $image"
```

```
grep -rl $image ./manifests | \
xargs sed -i "s/${escaped_image}/${escaped_image_to_attest}/g"
done
```

To verify whether the changes were successful, run the following command:

```
$ cat manifests/blog-app/blog-app.yaml | grep "image:"
image: docker.io/library/mongo@sha256:2a1093b275d9bc...
image: docker.io/bharamicrosystems/mdo-posts@sha256:b5bc...
image: docker.io/bharamicrosystems/mdo-reviews@sha256:073..
image: docker.io/bharamicrosystems/mdo-ratings@sha256:271..
image: docker.io/bharamicrosystems/mdo-users@sha256:5f5a...
image: docker.io/bharamicrosystems/mdo-frontend@sha256:87..
```

As we can see, the images have been updated. Now, let's proceed to push the changes to the remote repository using the following commands:

```
$ cp ~/modern-devops/ch13/binaryauth/binaryauth.tf terraform/
$ cp ~/modern-devops/ch13/binaryauth/cluster.tf terraform/
$ cp ~/modern-devops/ch13/binaryauth/variables.tf terraform/
$ cp -r ~/modern-devops/ch13/binaryauth/.github .
$ git add --all
$ git commit -m "Enabled Binary Auth"
$ git push
```

Now, let's review the Dev CD workflow on GitHub Actions, where we should observe the following:

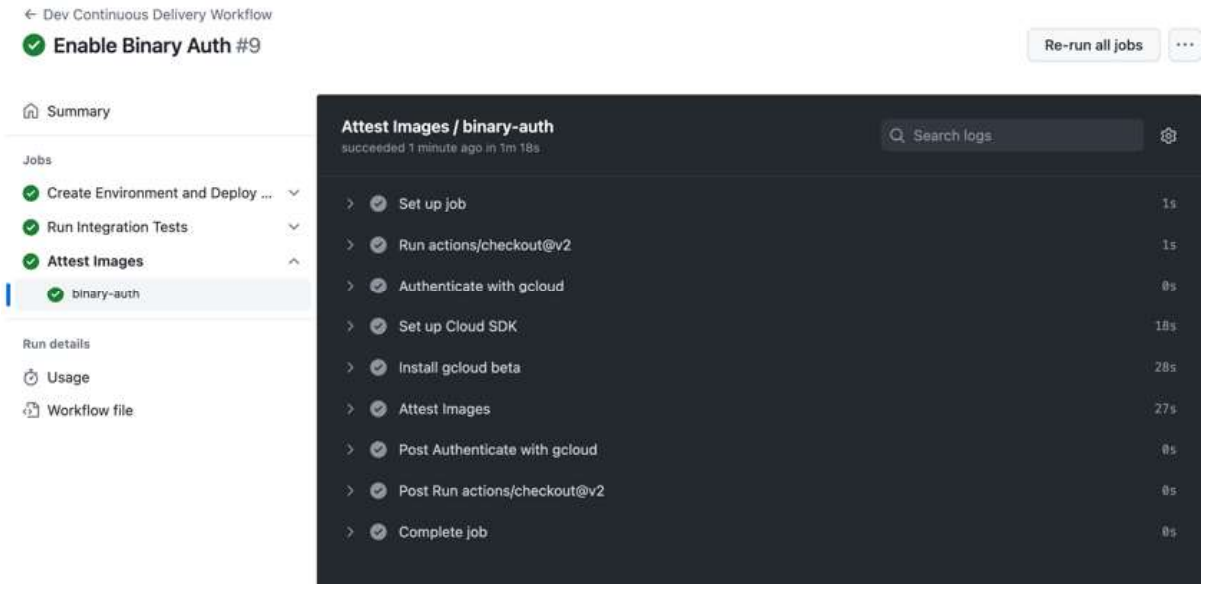

Figure 13.13 – Dev CD workflow – Attest Images

As is evident, the workflow has successfully configured binary authorization and attested our images. To verify, execute the following command:

```
$ gcloud beta container binauthz attestations list \
 --attestor-project="$PROJECT_ID" \
 --attestor="quality-assurance-attestor" | grep resourceUri
resourceUri: docker.io/bharamicrosystems/mdo-ratings@
sha256:271981faefafb86c2d30f7d3ce39cd8b977b7dd07...
resourceUri: docker.io/library/mongo@sha256:2a1093b275d9bc546135ec2e2...
resourceUri: docker.io/bharamicrosystems/mdo-posts@
sha256:b5bc1fc976a93a88cc312d24916bd1423dbb3efe25e...
resourceUri: docker.io/bharamicrosystems/mdo-frontend@
sha256:873526fe6de10e04c42566bbaa47b76c18f265fd...
resourceUri: docker.io/bharamicrosystems/mdo-users@
sha256:5f5aa595bc03c53b86dadf39c928eff4b3f05533239...
resourceUri: docker.io/bharamicrosystems/mdo-reviews@
sha256:07370e90859000ff809b1cd1fd2fc45a14c5ad46e...
```

As we can see, the attestations have been successfully created. Having deployed our application in the Dev environment, tested it, and attested all the images within, we can now proceed with deploying the code to the Prod environment. This involves merging our code with the `prod` branch, and we will implement pull request gating for this purpose.

Release gating with pull requests and deployment to production

The process of pull request gating is straightforward. At the end of the Dev CD workflow, we'll introduce a step to initiate a pull request to merge `dev` into the `prod` branch. Human approval is required to proceed with merging the pull request. This step highlights how various organizations may adopt different methods to verify and promote tested code. Some may opt for automated merging, while others may prioritize human-triggered actions. Once the code is successfully merged into the `prod` branch, it triggers the Prod CD workflow. This workflow creates the Prod environment and deploys our application. It also executes the same integration test we ran in the Dev environment to ensure the deployed application in Prod remains intact.

Here's the step we'll add to the Dev CD workflow:

```
  raise-pull-request:
    name: Raise Pull Request
    needs: [binary-auth]
    uses: ./.github/workflows/raise-pr.yml
    secrets: inherit
```

As we can see, this step invokes the `raise-pr.yml` file. Let's look at that:

```
...
    steps:
      - uses: actions/checkout@v2
      - name: Raise a Pull Request
        id: pull-request
        uses: repo-sync/pull-request@v2
        with:
          destination_branch: prod
          github_token: ${{ secrets.GH_TOKEN }}
```

This workflow does the following:

- Checks out the code from the repository
- Raises a pull request to merge with the `prod` branch using the `GH_TOKEN` secret

To enable the workflow's functionality, we need to define a GitHub token. This token allows the workflow to act on behalf of the current user when creating the pull request. Here are the steps:

1. Go to `https://github.com/settings/personal-access-tokens/new`.
2. Create a new token with **Repository** access for the `mdo-environments` repository, granting it the `read-write` pull request permission. This approach aligns with the principle of least privilege, offering more granular control.
3. Once the token is created, copy it.
4. Now, create a GitHub Actions secret named `GH_TOKEN` and paste the copied token as the value. You can do this by visiting `https://github.com/<your_github_user>/mdo-environments/settings/secrets/actions`.

Next, let's proceed to copy the workflow files using the following commands:

```
$ cd ~/mdo-environments/.github/workflows
$ cp ~/modern-devops/ch13/raise-pr/.github/workflows/dev-cd-workflow.yml .
$ cp ~/modern-devops/ch13/raise-pr/.github/workflows/raise-pr.yml .
```

We're ready to push this code to GitHub. Run the following commands to commit and push the changes to your GitHub repository:

```
$ git add --all
$ git commit -m "Added PR Gating"
$ git push
```

This should trigger a GitHub Actions workflow in your GitHub repository, and you should observe something similar to the following:

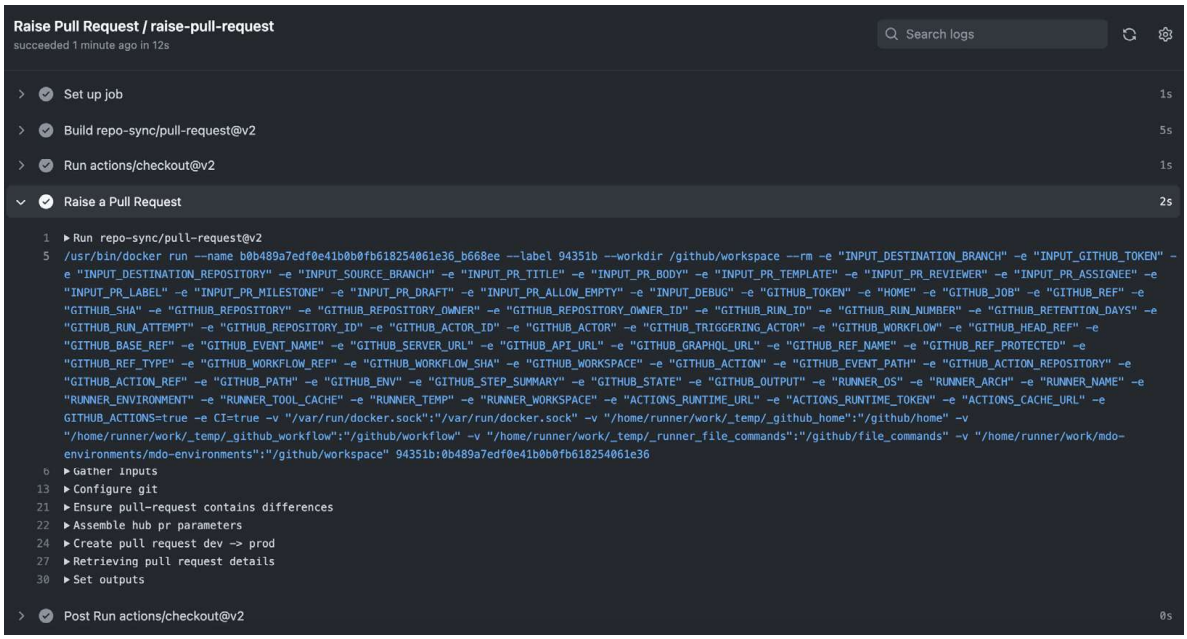

Figure 13.14 – Raising a pull request

GitHub has generated a pull request to merge the code into the `prod` branch, and the Dev CD workflow is running as anticipated. We can now review the pull request and merge the code into the `prod` branch.

Merging code and deploying to prod

As demonstrated in the previous section, the Dev CD workflow created our environment, deployed the application, tested it, and attested application images. It then automatically initiated a pull request to merge the code into the `prod` branch.

We've entered the **release gating phase,** where we require manual verification to determine whether the code is ready for merging into the `prod` branch.

Since we know the pull request has been created, let's proceed to inspect and approve it. To do so, go to `https://github.com/<your_github_user>/mdo-environments/pulls`, where you will find the pull request. Click on the pull request, and you will encounter the following:

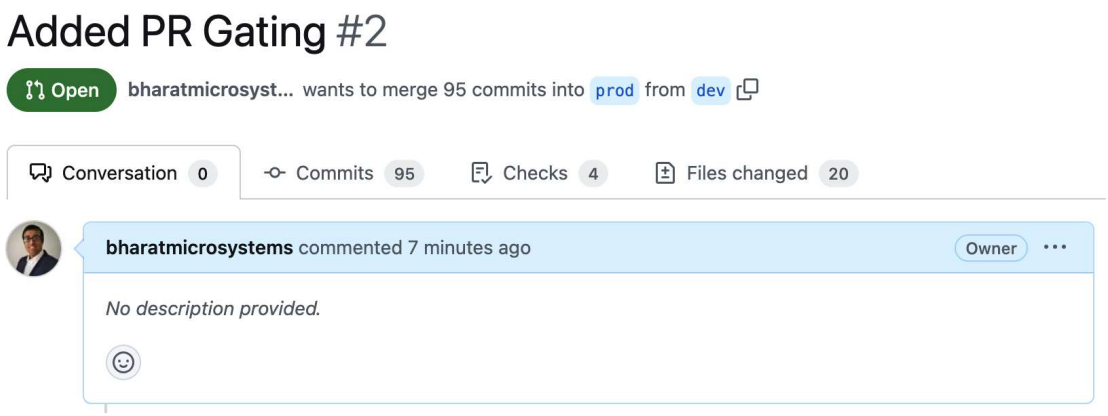

Figure 13.15 – Pull request

We see that the pull request is ready to merge. Click on **Merge pull request**, and you will see that the changes will reflect on the `prod` branch.

If you go to `https://github.com/<your_user>/mdo-environments/actions`, you'll find that the Prod CD workflow has been triggered. When you click on the workflow, you will see a workflow run like the following:

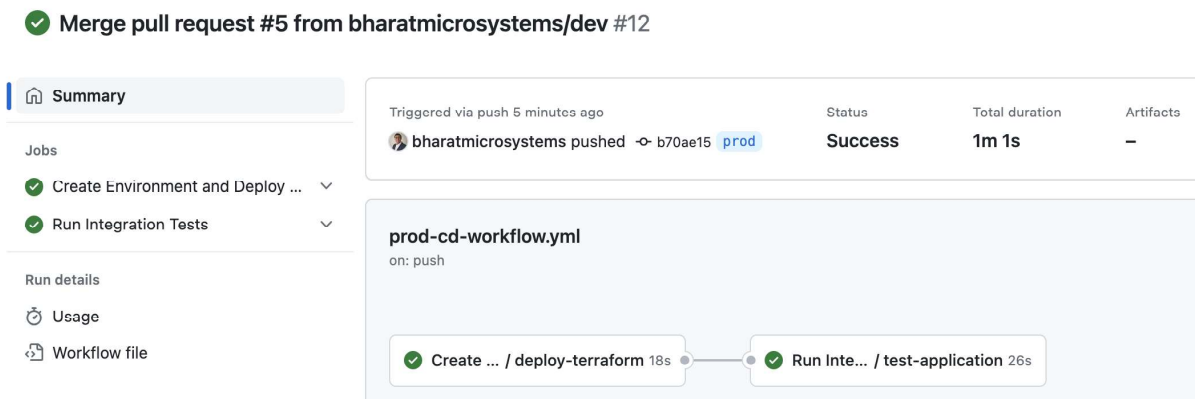

Figure 13.16 – Prod CD workflow

When we merged the pull request, it automatically triggered the Prod CD workflow as it would react to any new changes in the `prod` branch. The workflow did its job by building the Prod environment, deploying our application, and testing it. Note that binary authorization is enabled for this cluster.

To confirm that binary authorization is functioning correctly, let's perform some checks to ensure unattested images cannot be deployed.

First, let's establish a connection to the `prod` cluster using the following command:

```
$ gcloud container clusters get-credentials \
mdo-cluster-prod --zone us-central1-a --project ${PROJECT_ID}
```

Let's attempt to deploy a pod to your cluster using an `nginx` image. Please use the following command:

```
$ kubectl run nginx --image=nginx
Error from server (VIOLATES_POLICY): admission webhook "imagepolicywebhook.image-policy.
k8s.io" denied the request: Image nginx denied by Binary Authorization default admission
rule. Image nginx denied by attestor projects/<PROJECT_ID>/attestors/quality-assurance-
attestor: Expected digest with sha256 scheme, but got tag or malformed digest
```

Now, as expected, the deployment failed, but there's something else to note if you examine the reason. The failure happened because we specified a tag instead of a `sha256` digest. Let's attempt to deploy the image again, but this time, with a digest.

To do so, let's retrieve the image digest and set it as a variable called `DIGEST` using the following command:

```
$ DIGEST=$(docker pull nginx | grep Digest | awk {'print $2'})
```

Now, let's redeploy the image using the digest with the following command:

```
$ kubectl run nginx --image=nginx@$DIGEST
Error from server (VIOLATES_POLICY): admission webhook "imagepolicywebhook.image-
policy.k8s.io" denied the request: Image nginx@sha256:6926dd8... denied by Binary
Authorization default admission rule. Image nginx@sha256:6926dd8... denied by attestor
projects/<PROJECT_ID>/attestors/quality-assurance-attestor: No attestations found that
were valid and signed by a key trusted by the attestor
```

This time, the deployment was denied for a valid reason, confirming that binary authorization functions correctly. This ensures the security of your Kubernetes cluster, preventing the deployment of unattested images and giving you complete control over your environment. With this in place, any issues that arise won't stem from deploying untested or vulnerable images.

We've covered a lot of ground in integrating security and QA into our CI/CD pipelines. Now, let's explore some best practices for securing modern DevOps pipelines.

Security and testing best practices for modern DevOps pipelines

Tooling is not the only thing that will help you in your DevSecOps journey. Here are some helpful tips that can help you address security risks and have a more secure culture within your organization.

Adopt a DevSecOps culture

Adopting a DevSecOps approach is critical in implementing modern DevOps. Therefore, it is vital to embed security within an organization's culture. You can achieve that by implementing effective communication and collaboration between the *development*, *operations*, and *security* teams. While most organizations have a security policy, it mustn't be followed just to comply with rules and regulations. Instead, employees should cross-skill and upskill themselves to adopt a DevSecOps approach and embed security early on during development. Security teams need to learn how to write code and work with APIs, while developers need to understand security and use automation to achieve this.

Establish access control

You have heard about the **Principle of Least Privilege** (**PoLP**) several times in this book. Well, that is what you need to implement for a better security posture, which means you should make all attempts to grant only the required privileges to people to do their job, and nothing more. Reduce the just-in-case syndrome by making the process of giving access easier so that people don't feel hindered, and as a result, they do not seek more privileges than they require.

Implement shift left

Shifting left means embedding security into software at the earlier stages of software development. This means security experts need to work closely with developers to enable them to build secure software right from the start. The security function should not be review-only but should actively work with developers and architects to develop a security-hardened design and code.

Manage security risks consistently

You should accept risks, which are inevitable, and should have a **Standard Operating Procedure** (**SOP**) should an attack occur. You should have straightforward and easy-to-understand policies and practices from a security standpoint in all aspects of software development and infrastructure management, such as **configuration management**, **access controls**, **vulnerability testing**, **code review**, and **firewalls**.

Implement vulnerability scanning

Open source software today is snowballing, and most software implementations rely on ready-made open source frameworks, software libraries, and third-party software that don't come with a guarantee or liability of any kind. While the open source ecosystem is building the technological world like never before, it does have its own share of vulnerabilities, which you don't want to insert within your software through no fault of your own. Vulnerability scanning is crucial, as scans can discover any third-party dependency with vulnerabilities and alert you at the initial stage.

Automate security

Security should not hinder the speed of your DevOps teams; therefore, to keep up with the fast pace of DevOps, you should look at embedding security within your CI/CD processes. You can do code analysis, vulnerability scanning, configuration management, and infrastructure scanning with policy as code and binary authorization to allow only tested and secure software to be deployed. Automation helps identify potential vulnerabilities early on in the software development life cycle, thereby bringing down the cost of software development and rework.

Similarly, QA is the backbone of software delivery, and modern DevSecOps heavily emphasizes automating it. Here are some tips you can follow to implement a modern testing approach.

Test automation within your CI/CD pipelines

Automating testing across the board is key. This means encompassing a wide spectrum, from unit and integration testing to functional, security, and performance testing. The goal is to seamlessly embed these tests within your CI/CD pipeline, ensuring a constant stream of validation. In this journey, creating isolated and reproducible test environments becomes crucial to thwart any interference among tests. Here, methods such as containerization and virtualization are valuable tools for environment isolation.

Manage your test data effectively

Test data management is another pivotal aspect. It's imperative to handle your test data effectively, not only ensuring its consistency but also safeguarding data privacy. Leveraging data generation tools can be a game-changer in this regard, allowing you to create relevant datasets for your testing needs. Moreover, when dealing with sensitive information, the consideration of data anonymization is prudent. This ensures that you maintain the highest standards of data protection while still benefiting from comprehensive testing procedures.

Test all aspects of your application

CI is all about keeping the development process flowing smoothly. This involves frequently merging code and running tests automatically, ensuring the code base remains stable. When tests fail, immediate attention is crucial to rectify the issues promptly.

End-to-end testing is your compass to ensure the entire application workflow functions as expected. Automation frameworks play a pivotal role in replicating real user interactions, making it possible to assess your application thoroughly.

Load testing is an essential part of the process, as it evaluates how your application performs under varying loads, providing insights into its robustness and capacity. Additionally, scalability testing ensures that the system is well-equipped to handle growth, an important factor for the long-term health of your application.

Implement chaos engineering

Incorporating chaos engineering practices is a proactive strategy to uncover and address potential system weaknesses. By conducting controlled experiments, you can gauge the resilience of your system and better prepare it for unexpected challenges. These experiments involve intentionally introducing chaos into your environment to observe how your system responds. This not only helps you identify weaknesses but also provides valuable insights into how to make your system more robust and reliable.

Monitor and observe your application when it is being tested

Setting up robust monitoring and observability tools is crucial for gaining deep insights into your system's performance and behavior. These tools allow you to collect essential metrics, logs, and traces, providing a comprehensive view of your application's health and performance.

Effective testing in production

Implementing feature flags and canary releases is a prudent strategy for testing new functionality in a real production environment while minimizing risks. Feature flags allow you to enable or disable certain features at runtime, giving you control over their activation. Canary releases involve rolling out new features to a small subset of users, allowing you to monitor their impact before a full-scale release.

By utilizing feature flags, you can introduce new features to a limited audience without affecting the entire user base. This controlled approach lets you observe user interactions, collect feedback, and assess the feature's performance in a real-world scenario. Simultaneously, canary releases enable you to deploy these features to a small, representative group of users, allowing you to monitor their behavior, collect performance metrics, and identify potential issues.

Crucially, continuous monitoring is essential during this process. By closely observing the impact of the new functionality, you can quickly detect any issues that may arise. If problems occur, you have the flexibility to roll back the changes by simply turning off the feature flags or reverting to the previous version. This iterative and cautious approach minimizes the impact of potential problems, ensuring a smoother user experience and maintaining the stability of your production environment.

Documentation and knowledge sharing

Documenting testing procedures, test cases, and best practices is essential for ensuring consistency and reliability within the development and testing processes. Comprehensive documentation serves as a reference for team members, providing clear guidelines on how to conduct tests, the expected outcomes, and the best practices to follow. This documentation acts as a valuable resource for both new and existing team members, fostering a shared understanding of the testing procedures.

Encouraging knowledge sharing among team members further enhances the collective expertise of the team. By promoting open communication and sharing experiences, team members can learn from one another, gain insights into different testing scenarios, and discover innovative solutions to

common challenges. This collaborative environment promotes continuous learning and ensures that the team stays updated on the latest developments and techniques in the field of software testing.

By adhering to these best practices, teams can enhance the security and reliability of their CI/CD pipelines. Properly documented procedures and test cases enable consistent testing, reducing the likelihood of introducing errors into the code base. Knowledge sharing ensures that the team benefits from the collective wisdom and experiences of its members, leading to more informed decision-making and efficient problem-solving.

In addition, managing security risks effectively becomes possible through well-documented testing procedures and disseminating best practices. Teams can identify potential security vulnerabilities early in the development process, enabling them to address these issues before they escalate into significant threats. Regular knowledge-sharing sessions can also include discussions about security best practices, ensuring that team members are aware of the latest security threats and countermeasures.

Ultimately, these best practices contribute to a robust testing and development culture. They empower teams to deliver software faster and with confidence, knowing that their CI/CD pipelines are secure, reliable, and capable of handling the challenges of modern software development.

Summary

This chapter has covered CI/CD pipeline security and testing, and we have understood various tools, techniques, and best practices surrounding it. We looked at a secure CI/CD workflow for reference. We then understood, using hands-on exercises, the aspects that made it secure, such as secret management, container vulnerability scanning, and binary authorization.

Using the skills learned in this chapter, you can now appropriately secure your CI/CD pipelines and make your application more secure.

In the next chapter, we will explore the operational elements along with key performance indicators for running our application in production.

Questions

1. Which of these is the recommended place for storing secrets?

   A. Private Git repository

   B. Public Git repository

   C. Docker image

   D. Secret management system

2. Which one of the following is an open source secret management system?

   A. Secret Manager

   B. HashiCorp Vault

   C. Anchore Grype

3. Is it a good practice to download a secret within your CD pipeline's filesystem?
4. Which base image is generally considered more secure and consists of the fewest vulnerabilities?

   A. Alpine

   B. Slim

   C. Buster

   D. Default

5. Which of the following answers are true about binary authorization? (Choose two)

   A. It scans your images for vulnerabilities.

   B. It allows only attested images to be deployed.

   C. It prevents people from bypassing your CI/CD pipeline.

Answers

1. D
2. B
3. No
4. A
5. B and C

14

Understanding Key Performance Indicators (KPIs) for Your Production Service

In the previous chapters, we looked at the core concepts of modern DevOps – **Continuous Integration** (**CI**) and **Continuous Deployment/Delivery** (**CD**). We also looked at various tools and techniques that can help us enable a mature and secure DevOps channel across our organization. In this rather theory-focused chapter, we'll try to understand some **key performance indicators** (**KPIs**) for operating our application in production.

In this chapter, we're going to cover the following main topics:

- Understanding the importance of reliability
- SLOs, SLAs, and SLIs
- Error budgets
- Recovery Time Objective (RPO) and Recovery Point Objective (RTO)
- Running distributed applications in production

So, let's get started!

Understanding the importance of reliability

Developing software is one thing, and running it in production is another. The reason behind such a disparity is that most development teams cannot simulate production conditions in non-production environments. Therefore, many bugs are uncovered when the software is already running in production. Most issues encountered are non-functional – for example, the services could not scale properly with additional traffic, the amount of resources assigned to the application was suboptimal, thereby crashing the site, and many more. These issues need to be managed to make the software more reliable.

To understand the importance of software reliability, let's look at an example retail banking application. Software reliability is critically important for several reasons:

- **User satisfaction**: Reliable software ensures a positive user experience. Users expect software to work as intended, and when it doesn't, it can lead to frustration, loss of trust, and a poor reputation for the software or the organization behind it. For a bank's retail customer, it might mean customers cannot do essential transactions and, therefore, may face hassles in payments and receipts, leading to a loss in user satisfaction.
- **Business reputation**: Software failures can tarnish a company's reputation and brand image. For our bank, if the issues are frequent, customers will look for other options, resulting in considerable churn and loss of business.
- **Financial impact**: Software failures can be costly. They can result in lost sales, customer support expenses, and even legal liabilities in cases where software failures cause harm or financial losses to users. This becomes especially critical for banking applications as customers' money is involved. If transactions don't happen in time, it can result in a loss of customer business, which will hurt the bank in the long run.
- **Competitive advantage**: Reliable software can provide a competitive edge. Users are more likely to choose and stick with a bank with robust online banking software that consistently meets their needs and expectations.
- **Productivity and efficiency**: Within organizations, reliable software is essential for maintaining productivity. Imagine the pain that the customer support and front office staff would have in such a disruption! You would also need more resources to manage these issues, which can disrupt operations, leading to wasted time and resources.
- **Security**: Reliable software is often more secure. Attackers can exploit vulnerabilities and bugs in unreliable software. In the case of a bank, security is of prime importance because any breach can result in direct financial impact and loss. Ensuring reliability is a fundamental part of cybersecurity.
- **Compliance**: In some industries, especially banking, there are regulatory requirements related to software reliability. Failing to meet these requirements can result in legal and financial penalties.
- **Customer trust**: Trust is a critical factor in software usage, especially in the case of a banking application. Users must trust that their money and data will be handled securely and that the software will perform as expected. Software reliability is a key factor in building and maintaining this trust.
- **Maintainability**: Reliable software is typically easier to maintain. When software is unreliable, fixing bugs and updating becomes more challenging, potentially leading to a downward spiral of increasing unreliability.

- **Scaling and growth**: As software usage grows, reliability becomes even more critical. Software that works well for a small user base may struggle to meet the demands of a larger user base without proper reliability measures in place.

In summary, software reliability is not just a technical concern; it has wide-reaching implications for user satisfaction, business success, and even legal and financial aspects. Therefore, investing in ensuring the reliability of software is a prudent and strategic decision for organizations.

Historically, running and managing software in production was the job of the Ops team, and most organizations still use it. The Ops team comprises a bunch of **system administrators** (**SysAdmins**) who must deal with the day-to-day issues of running the software in production. They implement scaling and fault tolerance with software, patch and upgrade software, work on support tickets, and keep the systems running so the software application functions well.

We've all experienced the divide between Dev and Ops teams, each with its own goals, rules, and priorities. Often, they found themselves at odds because what benefited Dev (software changes and rapid releases) created challenges for Ops (stability and reliability).

However, the emergence of DevOps has changed this dynamic. In the words of Andrew Shafer and Patrick Debois, DevOps is a culture and practice in software engineering aimed at bridging the gap between software development and operations.

Looking at DevOps from an Ops perspective, Google came up with **site reliability engineering** (**SRE**) as an approach that embodies DevOps principles. It encourages shared ownership, the use of common tools and practices, and a commitment to learning from failures to prevent recurring issues. The primary objective is to develop and maintain a dependable application without sacrificing the speed of delivery – a balance that was once thought contradictory (that is, *create better software faster*).

The idea of SRE is a novel thought about what would happen if we allowed software engineers to run the production environment. So, Google devised the following approach for running its Ops team.

For Google, an ideal candidate for joining the SRE team should exhibit two key characteristics:

- Firstly, they quickly become disinterested in manual tasks and seek opportunities to automate them
- Secondly, they possess the requisite skills to develop software solutions, even when faced with complex challenges

Additionally, SREs should share an academic and intellectual background with the broader development organization. Essentially, SRE work, traditionally within the purview of operations teams, is carried out by engineers with strong software expertise. This strategy hinges on the natural inclination and capability of these engineers to design and implement automation solutions, thus reducing reliance on manual labor.

By design, SRE teams maintain a strong engineering focus. Without continuous engineering efforts, the operational workload escalates, necessitating an expansion of the team to manage the increasing demands. In contrast, a conventional operations-centric group scales in direct proportion to the growth of the service. If the services they support thrive, operational demands surge with increased traffic, compelling the hiring of additional personnel to perform repetitive tasks.

To avert this scenario, the team responsible for service management must incorporate coding into their responsibilities; otherwise, they risk becoming overwhelmed.

Accordingly, Google establishes a 50% upper limit on the aggregate "Ops" work allocated to all SREs, encompassing activities such as handling tickets, on-call duties, and manual tasks. This constraint guarantees that SRE teams allocate a substantial portion of their schedules to enhancing the stability and functionality of the service. While this limit serves as an upper bound, the ideal outcome is that, over time, SREs carry minimal operational loads and primarily engage in development endeavors as the service evolves to a self-sustaining state. Google's objective is to create systems that are not merely automated but inherently self-regulating. However, practical considerations such as scaling and introducing new features continually challenge SREs.

SREs are meticulous in their approach, relying on measurable metrics to track progress toward specific goals. For instance, stating that a website is *running slowly* is vague and unhelpful in an engineering context. However, declaring that the 95th percentile of response time has exceeded the **service-level objective** (**SLO**) by 10% provides precise information. SREs also focus on reducing repetitive tasks, known as **toil**, by automating them to prevent burnout. Now, let's look at some of the key SRE performance indicators.

Understanding SLIs, SLOs, and SLAs

In the realm of site reliability, three crucial parameters guide SREs: the **indicators of availability – service-level indicators** (**SLIs**), the **definition of availability** –SLOs, and the **consequences of unavailability – service-level agreements** (**SLAs**). Let's start by exploring SLIs in detail.

SLIs

SLIs serve as quantifiable reliability metrics. Google defines them as "*carefully defined quantitative measures of some aspect of the level of service provided.*" Common examples include request latency, failure rate, and data throughput. SLIs are specific to user journeys, which are sequences of actions users perform to achieve specific goals. For instance, a user journey for our sample Blog App might involve creating a new blog post.

Google, the original advocate of SRE, has identified four golden signals that apply to most user journeys:

- **Latency**: This measures the time it takes for your service to respond to user requests
- **Errors**: This indicates the percentage of failed requests, highlighting issues in service reliability

- **Traffic**: Traffic represents the demand directed toward your service, reflecting its usage
- **Saturation**: Saturation assesses how fully your infrastructure components are utilized

One recommended approach by Google to calculate SLIs is by determining the ratio of good events to valid events:

```
SLI = (Good Events * 100) / Valid Events
```

A perfect SLI score of 100 implies everything functions correctly, while a score of 0 signifies widespread issues.

A valuable SLI should align closely with the user experience. For example, a lower SLI value should correspond to decreased customer satisfaction. If this alignment is absent, the SLI may not provide meaningful insights or be worth measuring.

Let's look at the following figure to understand this better:

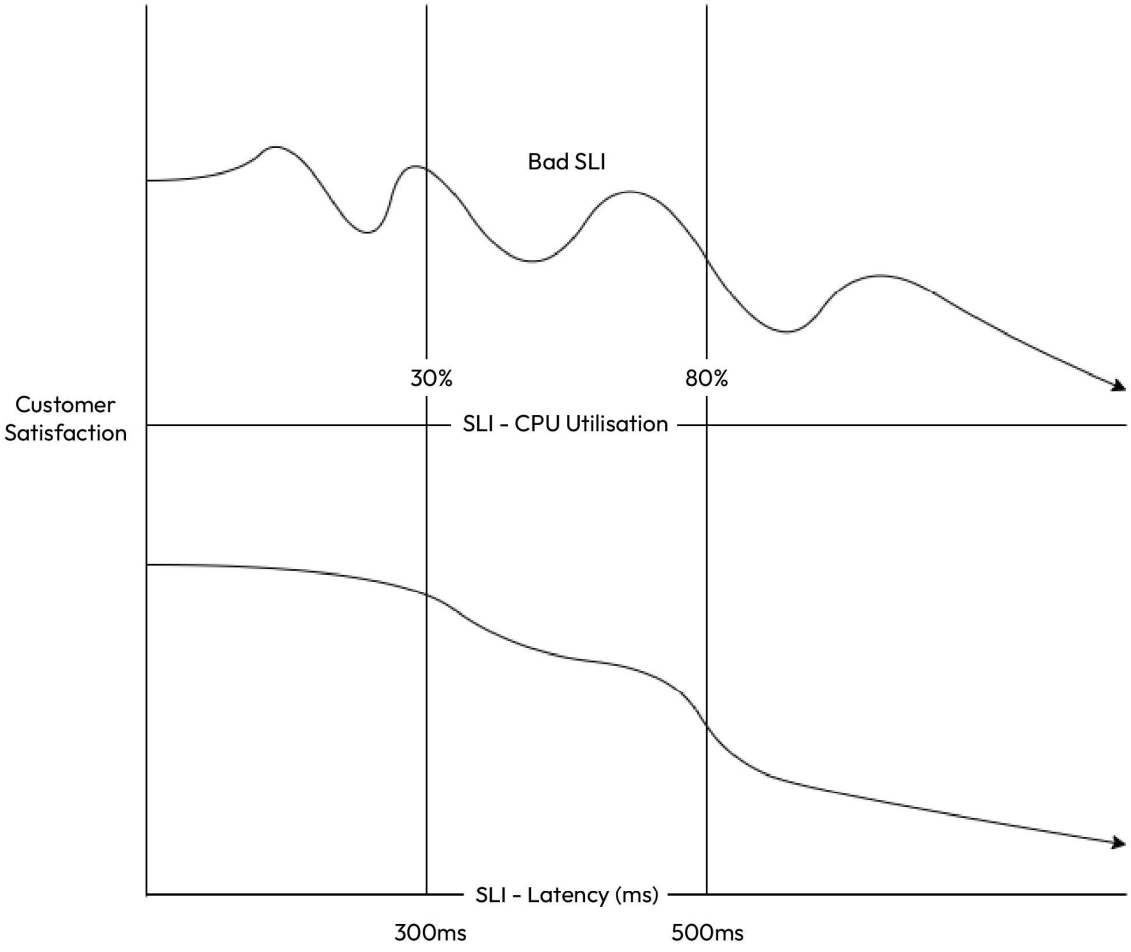

Figure 14.1 – Good versus bad SLI

As we can see, the CPU Utilisation SLI does not reflect customer satisfaction in any way; that is, there is no direct correlation between increasing CPU Utilisation and decreased customer satisfaction except after it crosses the 80% threshold. In contrast, the Latency SLI directly correlates with customer satisfaction, which reduces with increasing latency and significantly after the 300ms and 500ms levels. Therefore, it is a good idea to use Latency as an SLI over CPU Utilization.

It's also advisable to limit the number of SLIs to a manageable quantity. Too many SLIs can lead to team confusion and trigger numerous false alarms. It's best to focus on four or five metrics directly linked to customer satisfaction. For instance, instead of monitoring CPU and memory usage, prioritize metrics such as request latency and error rate.

Furthermore, prioritizing user journeys is essential, giving higher importance to journeys that significantly impact customers and lower importance to those with less of a customer impact. For example, ensuring a seamless create and update post experience in our Blog App is more critical than the reviews and ratings service. SLIs alone do not make much sense as they are just measurable indicators. We need to set objectives for SLIs. So, let's look at SLOs.

SLOs

Google's definition of SLOs states that they "*establish a target level for the reliability of your service.*" They specify the percentage of compliance with SLIs required to consider your site reliable. SLOs are formulated by combining one or more SLIs.

For instance, if you have an SLI that mandates *request latency to remain below 500ms within the last 15 minutes with a 95th percentile measurement*, an SLO would necessitate *the SLI to be met 99% of the time for a 99% SLO*.

While every organization aims for 100% reliability, setting a 100% SLO is not a practical goal. A system with a 100% SLO tends to be costly, technically complex, and often unnecessary for most applications to be deemed acceptable by their users.

In the realm of software services and systems, the pursuit of 100% availability is generally misguided because users cannot feel any practical distinction between a system that is 100% available and one that is 99.999% available. Multiple intermediary systems exist between the user and the service, such as their personal computer, home Wi-Fi, **Internet Service Provider** (**ISP**), and the power grid, and these collectively exhibit availability far lower than 99.999%. Consequently, the negligible difference between 99.999% and 100% availability becomes indistinguishable amidst the background noise of other sources of unavailability. Thus, investing substantial effort to attain that last 0.001% availability yields no noticeable benefit to the end user.

In light of this understanding, a question arises: if 100% is an inappropriate reliability target, what constitutes the right reliability target for a system? Interestingly, this is not a technical inquiry but rather a product-related one, necessitating consideration of the following factors:

- **User satisfaction**: Determining the level of availability that aligns with user contentment, considering their typical usage patterns and expectations

- **Alternatives**: Evaluating the availability of alternatives available to dissatisfied users, should they seek alternatives due to dissatisfaction with the product's current level of availability
- **User behavior**: Examining how users' utilization of the product varies at different availability levels, recognizing that user behavior may change in response to fluctuations in availability

Moreover, a completely reliable application leaves no room for the introduction of new features, as any new addition has the potential to disrupt the existing service. Therefore, some margin for error must be built into your SLO.

SLOs represent internal objectives that require consensus among the team and internal stakeholders, including developers, product managers, SREs, and CTOs. They necessitate the commitment of the entire organization. Not meeting an SLO does not carry explicit or implicit penalties.

For example, a customer cannot claim damages if an SLO is not met, but it may lead to dissatisfaction within organizational leadership. This does not imply that failing to meet an SLO should be consequence-free. Falling short of an SLO typically results in fewer changes and reduced feature development, potentially indicating a decline in quality and increased emphasis on the development and testing functions.

SLOs should be realistic, with the team actively working to meet them. They should align with the customer experience, ensuring that if the service complies with the SLO, customers do not perceive any service quality issues. If performance falls below the defined SLOs, it may affect the customer experience, but not to the extent that customers raise support tickets.

Some organizations implement two types of SLOs: **achievable** and **aspirational**. The achievable SLO represents a target the entire team should reach, while the aspirational SLO sets a higher goal and is part of an ongoing improvement process.

SLAs

According to Google, SLAs are "*formal or implicit agreements with your users that outline the repercussions of meeting (or failing to meet) the contained SLOs.*"

These agreements are of a more structured nature and represent business-level commitments made to customers, specifying the actions that will be taken if the organization fails to fulfill the SLA. SLAs can be either explicit or implicit. An explicit SLA entails well-defined consequences, often in terms of service credits, in case the expected reliability is not achieved. Implicit SLAs are evaluated in terms of potential damage to the organization's reputation and the likelihood of customers switching to alternatives.

SLAs are typically established at a level that is sufficient to prevent customers from seeking alternatives, and consequently, they tend to have lower thresholds compared to SLOs. For instance, when considering the request latency SLI, the SLO might be defined at a *300ms* SLI value, while the SLA could be set at a *500ms* SLI value. This distinction arises from the fact that SLOs are internal targets related to reliability, whereas SLAs are external commitments. By striving to meet the SLO, the team automatically satisfies the SLA, providing an added layer of protection for the organization in case of unexpected failures.

To understand the correlation between SLIs, SLOs, and SLAs, let's look at the following figure:

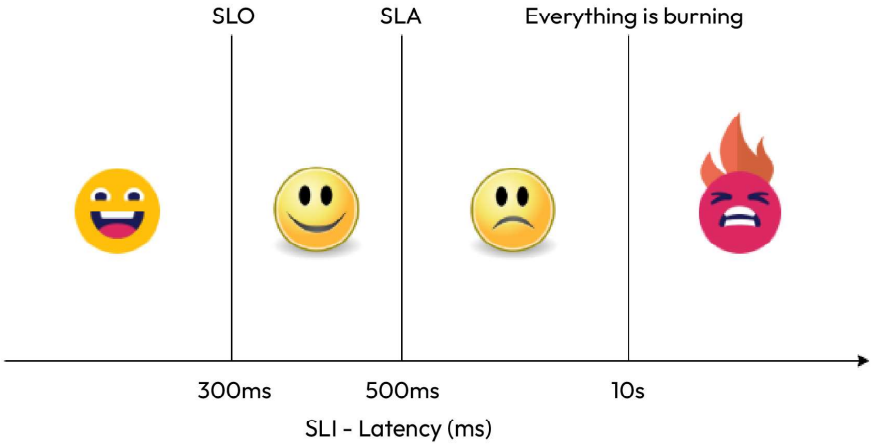

Figure 14.2 – SLIs, SLOs, and SLAs

This figure shows how customer experience changes with the level of latency. If we keep the latency SLO at *300ms* and meet it, everything is good! Anything between *300ms* to *500ms* and the customer starts experiencing some degradation in performance, but that is not enough for them to lose their cool and start raising support tickets. Therefore, keeping the SLA at *500ms* is a good strategy. As soon we cross the *500ms* threshold, unhappiness sinks in, and the customer starts raising support tickets for service slowness. If things cross the *10s* mark, then it is a cause of worry for your Ops team, and *Everything is burning* at this stage. However, as we know, the wording of SLOs is slightly different from what we imagine here. When we say that we have an SLO for *300ms* latency, it does not mean anything. A realistic SLO for an SLI mandating *request latency to remain below 300ms within the last 15 minutes with a 95th percentile measurement* would be to meet *the SLI x% of the time*. What should that x be? Should it be *99%*, or should it be *95%*? How do we decide this number? Well, for that, we'll have to look at **error budgets**.

Error budgets

As defined by Liz Fong-Jones and Seth Vargo, error budgets represent "*a quantitative measure shared between product and SRE teams to balance innovation and stability.*"

In simpler terms, an error budget quantifies the level of risk that can be taken to introduce new features, conduct service maintenance, perform routine enhancements, manage network and infrastructure disruptions, and respond to unforeseen situations. Typically, the monitoring system measures the uptime of your service, while SLOs establish the target you aim to achieve. The error budget is the difference between these two metrics and represents the time available to deploy new releases, provided it falls within the error budget limits.

This is precisely why a *100%* SLO is not usually set initially. Error budgets serve the crucial purpose of helping teams strike a balance between innovation and reliability. The rationale behind error

budgets lies in the SRE perspective that failures are a natural and expected part of system operations. Consequently, whenever a new change is introduced into production, there is an inherent risk of disrupting the service. Therefore, a higher error budget allows for introducing more features:

```
Error Budget = 100% – SLO
```

For instance, if your SLO is *99%*, your error budget would be *1%*. If you calculate this over a month, assuming *30 days/month* and *24 hours/day*, you will have a *7.2-hour* error budget to allocate for maintenance or other activities. For a *99.9%* SLO, the error budget would be *43.2 minutes* per month, and for a *99.99%* SLO, it would be *4.32 minutes* monthly. You can refer to the following figure for more details:

Figure 14.3 – Error budgets versus SLOs

These periods represent actual downtime, but if your services have redundancy, high availability measures, and disaster recovery plans in place, you can potentially extend these durations because the service remains operational while you patch or address issues with one server.

Now, whether you want to keep on adding *9s* within your SLO or aim for a lower number would depend on your end users, business criticality, and availability requirements. A higher SLO is more costly and requires more resources than a lower SLO. However, sometimes, just architecting your

Disaster recovery, RTO, and RPO

Disaster recovery is a comprehensive strategy that's designed to ensure an organization's resilience in the face of unexpected, disruptive events, such as natural disasters, cyberattacks, or system failures. It involves the planning, policies, procedures, and technologies necessary to quickly and effectively restore critical IT systems, data, and operations to a functional state. A well-implemented disaster recovery plan enables businesses to minimize downtime, data loss, and financial impact, helping them maintain business continuity, protect their reputation, and swiftly recover from adversities, ultimately safeguarding their long-term success.

Every organization incorporates disaster recovery to varying degrees. Some opt for periodic backups or snapshots, while others invest in creating failover replicas of their production environment. Although failover replicas offer increased resilience, they come at the expense of doubling infrastructure costs. The choice of disaster recovery mechanism that an organization adopts hinges on two crucial KPIs – **Recovery Time Objective** (**RTO**) and **Recovery Point Objective** (**RPO**).

RTO and RPO are crucial metrics in disaster recovery and business continuity planning. RTO represents the maximum acceptable downtime for a system or application, specifying the time within which it should be restored after a disruption. It quantifies the acceptable duration of service unavailability and drives the urgency of recovery efforts.

On the other hand, RPO defines the maximum tolerable data loss in the event of a disaster. It signifies the point in time to which data must be recovered to ensure business continuity. Achieving a low RPO means that data loss is minimized, often by frequent data backups and replication. The following figure explains RTO and RPO beautifully:

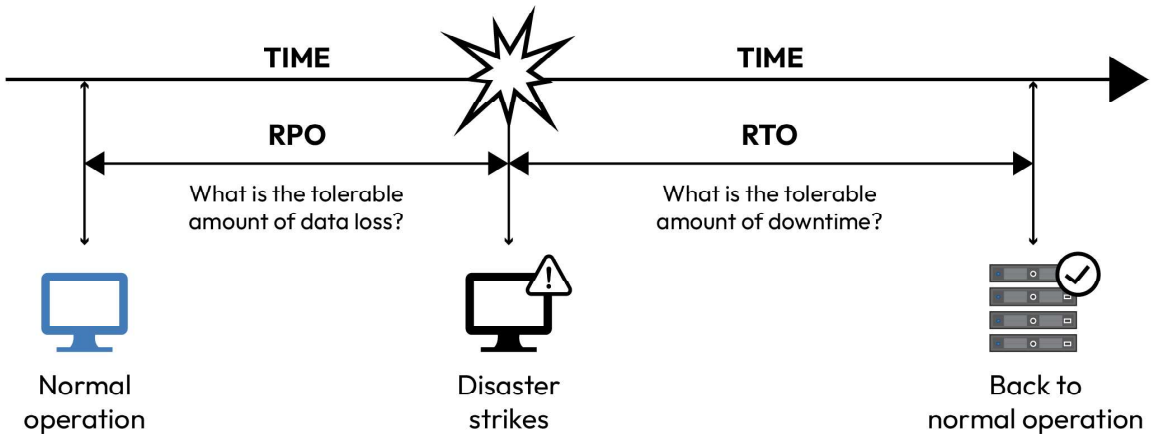

Figure 14.4 – RTO and RPO

A shorter RTO and RPO demand a more robust disaster recovery plan, which, in turn, results in higher costs for both infrastructure and human resources. Therefore, balancing RTO and RPO is essential to ensure a resilient IT infrastructure. Organizations must align their recovery strategies with these

objectives to minimize downtime and data loss, thereby safeguarding business operations and data integrity during unforeseen disruptions.

Running distributed applications in production

So far, we've been discussing KPIs for running an application in production, taking inspiration from SRE principles. Now, let's understand how we will put these thoughts in a single place to run a distributed application in production.

A **distributed application** or a **microservice** is inherently different from a monolith. While managing a monolith revolves around ensuring all operational aspects of one application, the complexity increases manyfold with microservices. Therefore, we should take a different approach to it.

From the perspective of SRE, running a distributed application in production entails focusing on ensuring the application's *reliability*, *scalability*, and *performance*. Here's how SREs approach this task:

- **SLOs**: SREs begin by defining clear SLOs that outline the desired level of reliability for the distributed application. SLOs specify the acceptable levels of *latency*, *error rates*, and *availability*. These SLOs are crucial in guiding the team's efforts and in determining whether the system is meeting its reliability goals.
- **SLIs**: SREs establish SLIs, which are quantifiable metrics that are used to measure the reliability of the application. These metrics could include response times, error rates, and other performance indicators. SLIs provide a concrete way to assess whether the application meets its SLOs.
- **Error budgets**: Error budgets are a key concept in SRE. They represent the permissible amount of downtime or errors that can occur before the SLOs are violated. SREs use error budgets to strike a balance between reliability and innovation. If the error budget is exhausted, it may necessitate a focus on stability and reliability over introducing new features.
- **Monitoring and alerting**: SREs implement robust monitoring and alerting systems to continuously track the application's performance and health. They set up alerts based on SLIs and SLOs, enabling them to respond proactively to incidents or deviations from desired performance levels. In the realm of distributed applications, using a service mesh such as **Istio** or **Linkerd** can help. They help you visualize parts of your application through a single pane of glass and allow you to monitor your application and alert on it with ease.
- **Capacity planning**: SREs ensure that the infrastructure supporting the distributed application can handle the expected load and traffic. They perform capacity planning exercises to scale resources as needed, preventing performance bottlenecks during traffic spikes. With modern public cloud platforms, automating scalability with traffic is all the more easy to implement, especially with distributed applications.
- **Automated remediation**: Automation is a cornerstone of SRE practices. SREs develop automated systems for incident response and remediation. This includes *auto-scaling*, *self-healing mechanisms*, and *automated rollback procedures* to minimize downtime.

- **Chaos engineering**: SREs often employ chaos engineering practices to introduce controlled failures into the system deliberately. This helps identify weaknesses and vulnerabilities in the distributed application, allowing for proactive mitigation of potential issues. Some of the most popular chaos engineering tools are Chaos Monkey, Gremlin, Chaos Toolkit, Chaos Blade, Pumba, ToxiProxy, and Chaos Mesh.
- **On-call and incident management**: SREs maintain on-call rotations to ensure 24/7 coverage. They follow well-defined incident management processes to resolve issues quickly and learn from incidents to prevent recurrence. Most SRE development backlogs come from this process as they learn from failures and, therefore, automate repeatable tasks.
- **Continuous improvement**: SRE is a culture of continuous improvement. SRE teams regularly conduct **post-incident reviews** (**PIRs**) and **root cause analyses** (**RCAs**) to identify areas for enhancement. Lessons learned from incidents are used to refine SLOs and improve the overall reliability of the application.
- **Documentation and knowledge sharing**: SREs document *best practices*, *runbooks*, and *operational procedures*. They emphasize knowledge sharing across teams to ensure that expertise is not siloed and that all team members can effectively manage and troubleshoot the distributed application. They also aim to automate the runbooks to ensure that manual processes are kept at a minimum.

In summary, SRE's approach to running a distributed application in production focuses on *reliability*, *automation*, and *continuous improvement*. It sets clear goals, establishes metrics for measurement, and employs proactive monitoring and incident management practices to deliver a highly available and performant service to end users.

Summary

This chapter covered SRE and the KPIs for running our service in production. We started by understanding software reliability and examined how to manage an application in production using SRE. We discussed the three crucial parameters that guide SREs: SLI, SLO, and SLA. We also explored error budgets and their importance in introducing changes within the system. Then, we looked at software disaster recovery, RPO, and RTO and how they define how complex or costly our disaster recovery measures will be. Finally, we looked at how DevOps or SRE will use these concepts to manage a distributed application in production.

In the next chapter, we will put what we've learned to practical use and explore how to manage all these aspects using a service mesh called Istio.

Questions

Answer the following questions to test your knowledge of this chapter:

1. Which of the following is a good example of an SLI?

   A. The response time should not exceed 300ms.

   B. The 95th percentile of response time in a 15-minute window should not exceed 300ms.

   C. 99% of all requests should respond within 300ms.

   D. The number of failures should not exceed 1%.

2. A mature organization should have a 100% SLO. (True/False)
3. SLOs are not tied to any customer-initiated punitive action. (True/False)
4. Which of the following should you consider while deciding on an SLO? (Choose three)

   A. User satisfaction

   B. Alternatives

   C. User behavior

   D. System capacity

5. SLAs are generally kept to a stricter SLI value than SLOs. (True/False)
6. Which of the following should you consider while defining SLIs?

   A. CPU, memory, and disk utilization

   B. Latency, errors, traffic, and saturation

   C. Utilization, capacity, and scale

7. An error budget of 1% provides how much scope for downtime per month?

   A. 72 hours

   B. 43.2 minutes

   C. 7.2 hours

   D. 4.32 minutes

8. An SRE is a software developer doing Ops. (True/False)
9. What minimum percent of the time should an SRE allocate to development work?

   A. 30%

   B. 40%

   C. 50%

   D. 60%

Answers

Here are the answers to this chapter's questions:

1. B
2. False
3. True
4. A, B, C
5. False
6. B
7. C
8. True
9. C

15

Implementing Traffic Management, Security, and Observability with Istio

In the previous chapter, we covered **site reliability engineering** (**SRE**) and how it has helped manage production environments using DevOps practices. In this chapter, we'll dive deep into a service mesh technology called Istio, which will help us implement SRE practices and manage our application better in production.

In this chapter, we're going to cover the following main topics:

- Revisiting the Blog App
- Introduction to service mesh
- Introduction to Istio
- Understanding the Istio architecture
- Installing Istio
- Using Istio Ingress to allow traffic
- Securing your microservices using Istio
- Managing traffic with Istio
- Observing traffic and alerting with Istio

Technical requirements

For this chapter, we will spin up a cloud-based Kubernetes cluster, **Google Kubernetes Engine** (**GKE**), for the exercises. At the time of writing, **Google Cloud Platform** (**GCP**) provides a free $300 trial for 90 days, so you can go ahead and sign up for one at `https://console.cloud.google.com/`.

You will also need to clone the following GitHub repository for some of the exercises: `https://github.com/PacktPublishing/Modern-DevOps-Practices-2e`.

You can use the Cloud Shell offering available from GCP to follow this chapter. Go to Cloud Shell and start a new session. Run the following command to clone the repository into your home directory to access the required resources:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
modern-devops
```

We also need to set the project ID and enable a few GCP APIs we will use in this chapter. To do so, run the following commands:

```
$ PROJECT_ID=<YOUR_PROJECT_ID>
$ gcloud services enable iam.googleapis.com \
 container.googleapis.com \
 binaryauthorization.googleapis.com \
 containeranalysis.googleapis.com \
 secretmanager.googleapis.com \
 cloudresourcemanager.googleapis.com \
 cloudkms.googleapis.com
```

If you haven't followed the previous chapters and want to start quickly with this, you can follow the next part, *Setting up the baseline*, though I highly recommend that you go through the last few chapters to get a flow. If you have been following the hands-on exercises in the previous chapters, feel free to skip this part.

Setting up the baseline

To ensure continuity with the previous chapters, let's start by creating a service account for Terraform so that we can interact with our GCP project:

```
$ gcloud iam service-accounts create terraform \
--description="Service Account for terraform" --display-name="Terraform"
$ gcloud projects add-iam-policy-binding $PROJECT_ID \
--member="serviceAccount:terraform@$PROJECT_ID.iam.gserviceaccount.com" \
--role="roles/editor"
$ gcloud iam service-accounts keys create key-file \
--iam-account=terraform@$PROJECT_ID.iam.gserviceaccount.com
```

You will see that a file called `key-file` has been created within your working directory. Now, create a new repository called `mdo-environments` with a `README.md` file on GitHub, rename the `main`

branch to `prod`, and create a new branch called `dev` using GitHub. Navigate to `https://github.com/<your_github_user>/mdo-environments/settings/secrets/actions/new` and create a secret named `GCP_CREDENTIALS`. For the value, print the `key-file` file, copy its contents, and paste it into the **values** field of the GitHub secret.

Next, create another secret, `PROJECT_ID`, and specify your GCP project ID within the **values** field.

Next, we need to create a **GCS bucket** for Terraform to use as a remote backend. To do this, run the following commands:

```
$ gsutil mb gs://tf-state-mdo-terraform-${PROJECT_ID}
```

The next thing we need to do is set up our Secrets Manager. Let's create a secret called `external-secrets`, where we will pass the MongoDB credentials in the JSON format. To do so, run the following command:

```
$ echo -ne '{"MONGO_INITDB_ROOT_USERNAME": "root", \
"MONGO_INITDB_ROOT_PASSWORD": "itsasecret"}' | \
gcloud secrets create external-secrets --locations=us-central1 \
--replication-policy=user-managed --data-file=-
Created version [1] of the secret [external-secrets].
```

We need to create the `Secret` resource, which will interact with GCP to fetch the stored secret. First, we need to create a GCP service account to interact with Secrets Manager using the following commands:

```
$ cd ~
$ gcloud iam service-accounts create external-secrets
```

As we're following the principle of least privilege, we will add the following role binding to provide access only to the `external-secrets` secret, as follows:

```
$ gcloud secrets add-iam-policy-binding external-secrets \
--member "serviceAccount:external-secrets@$PROJECT_ID.iam.gserviceaccount.com" \
--role "roles/secretmanager.secretAccessor"
```

Now, let's generate the service account key file using the following command:

```
$ gcloud iam service-accounts keys create key.json \
--iam-account=external-secrets@$PROJECT_ID.iam.gserviceaccount.com
```

Next, copy the contents of the `key.json` file into a new GitHub Actions secret called `GCP_SM_CREDENTIALS`.

We also need to create the following GitHub Actions secrets for binary authorization to work:

```
ATTESTOR_NAME=quality-assurance-attestor
KMS_KEY_LOCATION=us-central1
KMS_KEYRING_NAME=qa-attestor-keyring
KMS_KEY_NAME=quality-assurance-attestor-key
KMS_KEY_VERSION=1
```

As the workflow automatically raises pull requests at the end, we need to define a GitHub token. This token allows the workflow to act on behalf of the current user when creating the pull request. Here are the steps:

1. Go to `https://github.com/settings/personal-access-tokens/new`.
2. Create a new token with "Repository" access for the `mdo-environments` repository, granting it `read-write` pull request permissions. This approach aligns with the principle of least privilege, offering more granular control.
3. Once the token is created, copy it.
4. Now, create a GitHub Actions secret named `GH_TOKEN` and paste the copied token as the value.

Now that all the prerequisites have been met, we can clone our repository and copy the baseline code. Run the following commands to do this:

```
$ cd ~ && git clone git@github.com:<your_github_user>/mdo-environments.git
$ cd mdo-environments/
$ git checkout dev
$ cp -r ~/modern-devops/ch15/baseline/* .
$ cp -r ~/modern-devops/ch15/baseline/.github .
```

As we're now on the baseline, let's proceed further and understand the sample Blog App that we will deploy and manage in this chapter.

Revisiting the Blog App

Since we discussed the Blog App previously, let's look at the services and their interactions again:

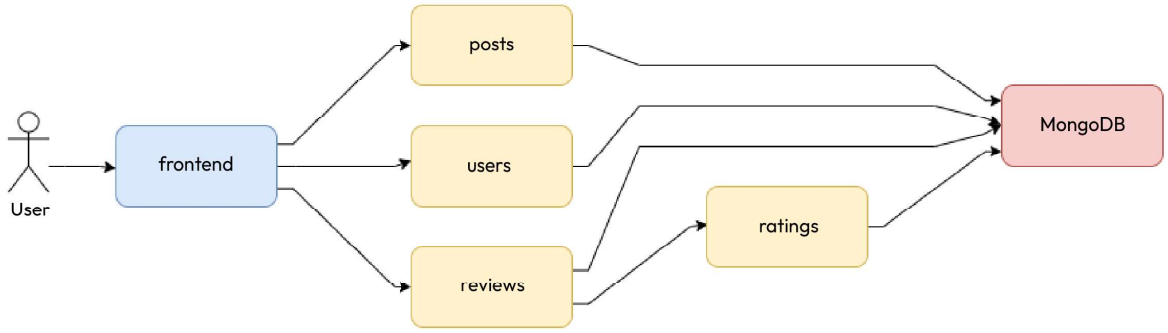

Figure 15.1 – The Blog App and its services and interactions

So far, we've created CI and CD pipelines for building, testing, and pushing our Blog App microservices containers using GitHub Actions, deploying them using Argo CD in a GKE cluster.

As you may recall, we created the following resources for the application to run seamlessly:

- **MongoDB**: We deployed an auth-enabled MongoDB database with root credentials. The credentials were injected via environment variables sourced from a Kubernetes **Secret** resource. To persist our database data, we created a **PersistentVolume** mounted to the container, which we provisioned dynamically using a **PersistentVolumeClaim**. As the container is stateful, we used a **StatefulSet** to manage it and, therefore, a headless **Service** to expose the database.
- **Posts, reviews, ratings, and users**: The *posts*, *reviews*, *ratings*, and *users* microservices interacted with MongoDB through the root credentials that were injected via environment variables sourced from the same **Secret** as MongoDB. We deployed them using their respective **Deployment** resources and exposed all of them via individual **ClusterIP Services**.
- **Frontend**: The *frontend* microservice does not need to interact with MongoDB, so there was no interaction with the Secret resource. We also deployed this service using a **Deployment** resource. As we wanted to expose the service on the internet, we created a **LoadBalancer Service** for it.

We can summarize them with the following diagram:

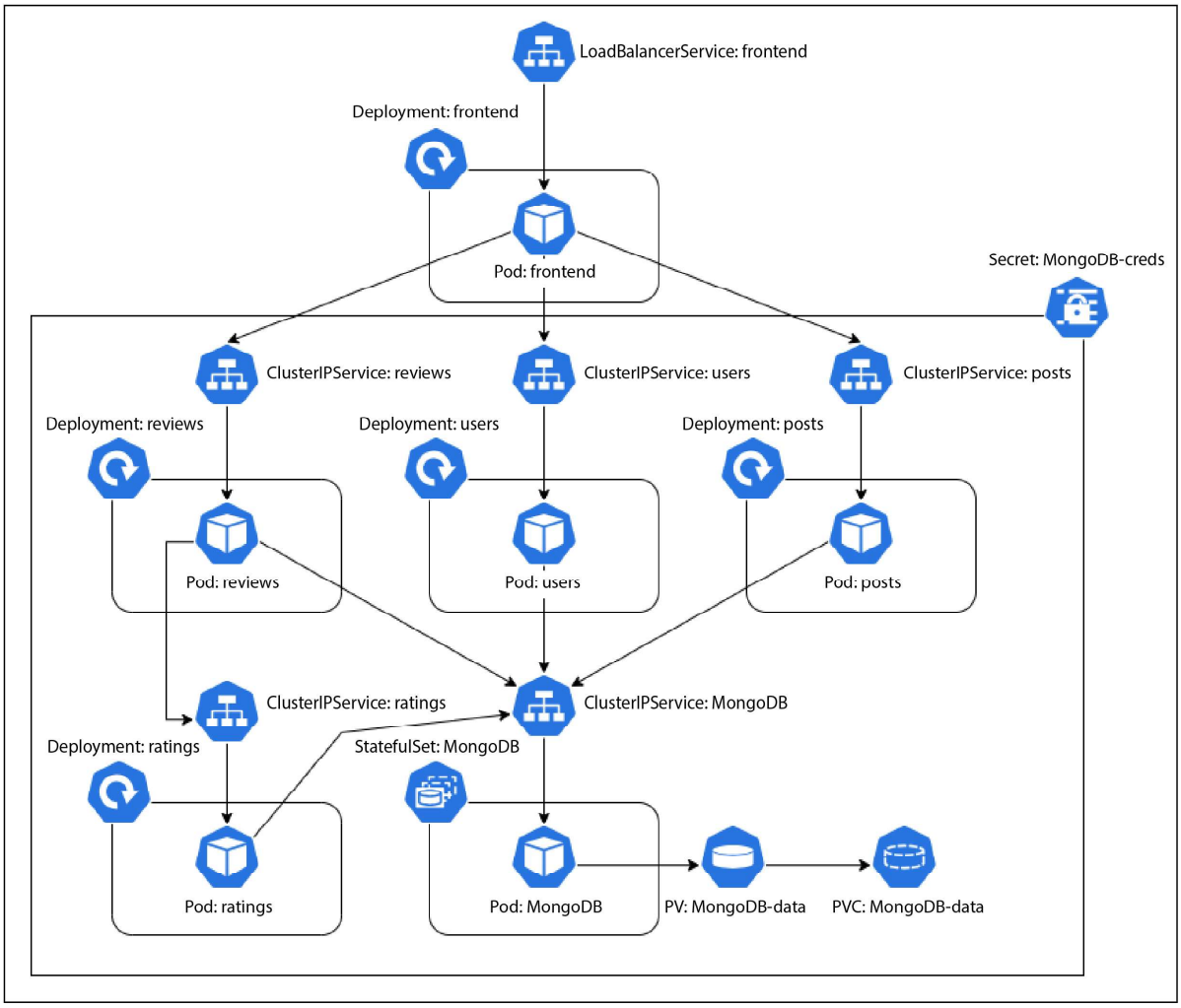

Figure 15.2 – Blog App – Kubernetes resources and interactions

Now that we understand the application, let's understand what a service mesh is and how it is beneficial in this use case.

Introduction to service mesh

Imagine being in a bustling city with a complex network of roads and highways. You're driving your car from one side of the city to the other. In this scenario, you deal with the following entities:

- **Your car**: Your car represents an individual service or application in a computer system. It has a specific purpose, just like a microservice or application in a software architecture.

- **Roads and highways**: The roads and highways are like the network connections and communication pathways between different services in your application. Services need to interact and communicate with each other to perform various functions, just as vehicles need roads to get from one place to another.
- **Traffic lights and signs**: Traffic lights, signs, and road rules help manage traffic flow, ensuring that vehicles (services) can safely and efficiently navigate the city. These are like the rules, protocols, and tools in a service mesh regulating communication and data exchange between services.
- **Traffic control center**: Think of the traffic control center as the service mesh. It's a centralized system that monitors and manages traffic flow across the city. Similarly, a service mesh is a centralized infrastructure that oversees and facilitates communication between services, ensuring they can communicate reliably and securely.
- **Traffic monitoring and optimization**: The traffic control center ensures safe travel and can optimize traffic flow. It can reroute vehicles to avoid congestion or accidents. In the context of a service mesh, it can optimize the flow of data and requests between services, ensuring efficient and resilient communication.
- **Safety and reliability**: In the city, the traffic control center helps prevent accidents and ensures everyone reaches their destinations safely. Similarly, a service mesh enhances the safety and reliability of your computer system by providing features such as load balancing, security, and fault tolerance.

So, just as the traffic control center makes your journey in a complex city more manageable and secure, a service mesh in a computer system simplifies and secures the communication between different services, ensuring that data and requests can flow smoothly, reliably, and safely.

Containers and the orchestration platforms that manage them, such as Kubernetes, have streamlined how we handle microservices. The introduction of container technology played a pivotal role in popularizing this concept by allowing for the execution and scalability of individual application components as self-contained entities, each with an isolated runtime environment.

While adopting a microservices architecture offers advantages such as accelerated development, enhanced system robustness, simplified testing, and the ability to scale different aspects of an application independently, it isn't without its challenges. Managing microservices can be a complex endeavor. Instead of dealing with a single, monolithic application, you now have multiple dynamic components, each catering to specific functionalities.

In the context of extensive applications, it's not uncommon to see hundreds of microservices interacting with each other, which can quickly become overwhelming. The primary concerns that may be raised by your security and operations teams are as follows:

- Ensuring secure communication between microservices. You need to secure numerous smaller services rather than securing a single monolithic application.

- Isolating a problematic microservice in case of an issue.
- Testing deployments with a limited percentage of traffic before a full release to establish trust.
- Consolidating application logs that are now distributed across multiple sources.
- Monitoring the health of the services becomes more intricate, with many components constituting the application.

While Kubernetes effectively addresses some management issues, it primarily serves as a container orchestration platform and excels in that role. However, it doesn't inherently solve all the complexities of a microservices architecture as they require specific solutions. Kubernetes does not inherently provide robust service management capabilities.

By default, communication between Kubernetes containers lacks security measures, and enforcing TLS between pods involves managing an extensive number of TLS certificates. Identity and access management between pods is also not applied out of the box.

While tools such as Kubernetes Network Policy can be employed to implement a firewall between pods, they function at a Layer 3 level rather than Layer 7, which is what modern firewalls operate at. This means you can identify the source of traffic but cannot inspect the data packets to make metadata-driven decisions, such as routing based on an HTTP header.

Although Kubernetes offers methods for deploying pods and conducting A/B testing and canary deployments, these processes often involve scaling container replicas. For example, deploying a new microservice version with just 10% of traffic directed to it requires at least 10 containers: 9 for the old version and 1 for the new version. Kubernetes distributes traffic evenly among pods without intelligent traffic splitting.

Each Kubernetes container within a pod maintains separate logging, necessitating a custom solution for capturing and consolidating logs.

While the Kubernetes dashboard provides features such as monitoring pods and checking their health, it does not offer insights into how components interact, the traffic distribution among pods, or the container chains that constitute the application. The inability to trace traffic flow through Kubernetes pods means you cannot pinpoint where in the chain a request encountered a failure.

To address these challenges comprehensively, a service mesh technology such as Istio can be of extreme help. This can effectively tackle the intricacies of managing microservices in Kubernetes and offer solutions for secure communication, intelligent traffic management, monitoring, and more. Let's understand what the Istio service mesh is through a brief introduction.

Introduction to Istio

Istio is a service mesh technology designed to streamline service connectivity, security, governance, and monitoring.

In the context of a microservices application, each microservice operates independently using containers, resulting in a complex web of interactions. This is where a service mesh comes into play, simplifying the discovery, management, and control of these interactions, often accomplished through a sidecar proxy. Allow me to break it down for you step by step.

Imagine a standard Kubernetes application comprising a frontend and a backend pod. Kubernetes offers built-in service discovery between pods using Kubernetes services and CoreDNS. Consequently, you can direct traffic using the service name from one pod to another. However, you won't have significant control over these interactions and runtime traffic management.

Istio steps in by injecting a sidecar container into your pod, which acts as a proxy. Your containers communicate with other containers via this proxy. This architecture allows all requests to flow through the proxy, enabling you to exert control over the traffic and collect data for further analysis. Moreover, Istio provides the means to encrypt communication between pods and enforce identity and access management through a unified control plane.

Due to this architecture, Istio boasts a range of core functionalities that enhance the traffic management, security, and observability of your microservices environment.

Traffic management

Istio effectively manages traffic by harnessing the power of the sidecar proxy, often referred to as the envoy proxy, alongside **ingress** and **egress gateways**. With these components, Istio empowers you to shape traffic and define service interaction rules. This includes implementing features such as **timeouts**, **retries**, **circuit breakers**, and much more, all through configurations within the control plane.

These capabilities open the door to intelligent practices such as **A/B testing**, **canary deployments**, and **staged rollouts** with **traffic division based on percentages**. You can seamlessly execute gradual releases, transitioning from an existing version (**Blue**) to a new one (**Green**), all with user-friendly controls.

Moreover, Istio allows you to conduct operational tests in a live production environment, offering **live traffic mirroring** to test instances. This enables you to gather real-time insights and identify potential production issues before they impact your application. Additionally, you can route requests to different language-specific microservices versions based on **geolocation or user profiles**, among other possibilities.

Security

Istio takes security seriously by securing your microservices through the envoy proxy and establishing identity access management between pods via mutual TLS. It is a robust defense against man-in-the-middle attacks through out-of-the-box **traffic encryption** between pods. This mutual authentication ensures that only trusted frontends can connect to backends, creating a strong trust relationship. Consequently, even if one of the pods is compromised, it cannot compromise the rest of your application. Istio further enhances security with **fine-grained access control policies** and introduces auditing tools currently lacking in Kubernetes, enhancing your cluster's overall security posture.

Observability

Thanks to the envoy sidecar, Istio maintains a keen awareness of the traffic flowing through the pods, enabling you to gather crucial telemetry data from the services. This wealth of data aids in gaining insights into service behavior and offers a window into future optimization possibilities for your applications. Additionally, Istio consolidates application logs and facilitates **traffic tracing** through multiple microservices. These features empower you to identify and resolve issues more swiftly, helping you isolate problematic services and expedite debugging.

Developer-friendly

Istio's most remarkable feature is its ability to relieve developers from the burdens of managing security and operational intricacies within their implementations.

Istio's Kubernetes-aware nature permits developers to continue building their applications as standard Kubernetes deployments. Istio seamlessly and automatically injects sidecar containers into the pods, sparing developers the need to worry about these technical intricacies.

Once these sidecar containers have been integrated, operations and security teams can then step in to enforce policies related to traffic management, security, and the overall operation of the application. This results in a mutually beneficial scenario for all involved parties.

Istio empowers security and operations teams to efficiently oversee microservices applications without hampering the development team's productivity. This collaborative approach ensures that each team within the organization can maintain its specialized focus and effectively contribute to the app's success. Now that we understand Istio, let's look at its architecture.

Understanding the Istio architecture

Istio simplifies microservices management through two fundamental components:

- **Data plane**: This comprises the sidecar envoy proxies that Istio injects into your microservices. These proxies take on the essential role of routing traffic between various services, and they also collect crucial telemetry data to facilitate monitoring and insights.

- **Control plane**: The control plane serves as the command center, instructing the data plane on how to route traffic effectively. It also handles the storage and management of configuration details, making it easier for administrators to interact with the sidecar proxy and take control of the Istio service mesh. In essence, the control plane functions as the intelligence and decision-making hub of Istio.

Similarly, Istio manages two types of traffic:

- **Data plane traffic**: This type of traffic consists of the core business-related data exchanged between your microservices. It encompasses the actual interactions and transactions that your application handles.
- **Control plane traffic**: In contrast, the control plane traffic consists of messages and communications between Istio components, and it is chiefly responsible for governing the behavior of the service mesh. It acts as the control mechanism that orchestrates the routing, security, and overall functioning of the microservices architecture.

The following diagram describes the Istio architecture in detail:

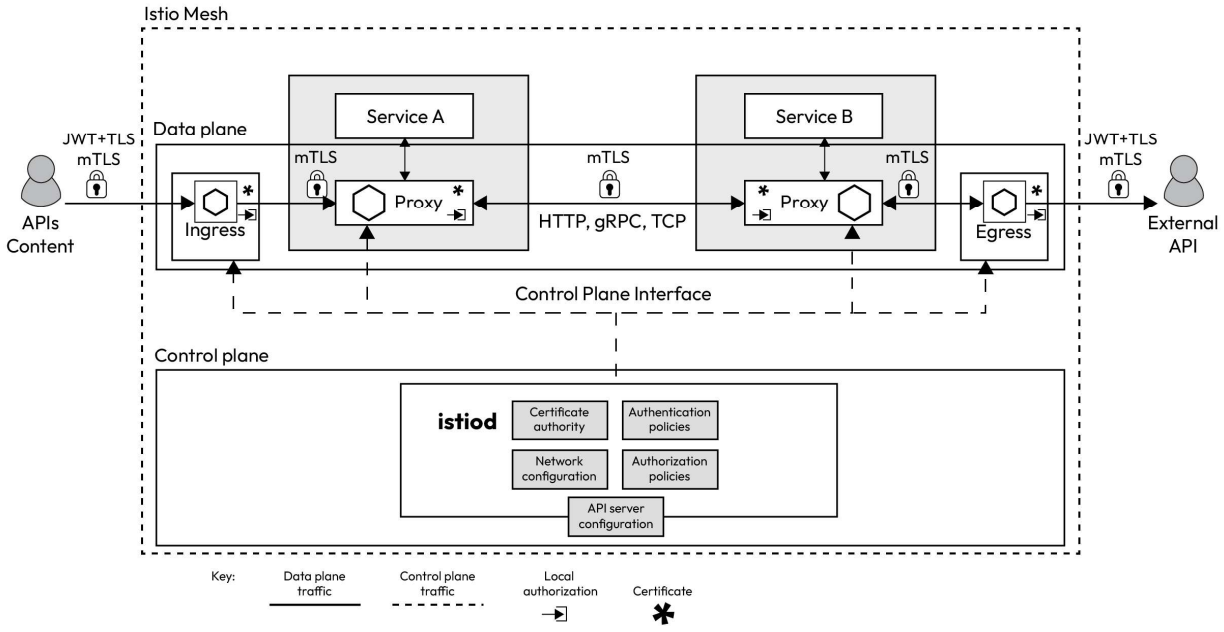

Figure 15.3 – Istio architecture

As we can see two distinct parts in the preceding diagram, the control plane and the data plane, let's go ahead and understand them.

The control plane architecture

Istio ships the control plane as a single **istiod** component. The Istio control plane, or istiod, comprises several critical components, each playing a distinct role in managing your service mesh.

Pilot

Pilot serves as the central control hub of the service mesh. It communicates with the envoy sidecars using the Envoy API and translates the high-level rules specified in Istio manifests into envoy configurations. Pilot enables service discovery, intelligent traffic management, and routing capabilities. It empowers you to implement practices such as A/B testing, Blue/Green deployments, canary rollouts, and more. Additionally, Pilot enhances the resiliency of your service mesh by configuring sidecars to handle tasks such as timeouts, retries, and circuit breaking. One of its notable features is providing a bridge between Istio configuration and the underlying infrastructure, allowing Istio to run on diverse platforms such as Kubernetes, **Nomad**, and **Consul**. Regardless of the platform, Pilot ensures consistent traffic management.

Citadel

Citadel focuses on identity and access management within your service mesh, fostering secure communication between Kubernetes pods. It safeguards your pods by ensuring encrypted communication, even if your developers have designed components with insecure TCP connections. Citadel simplifies the implementation of mutual TLS by managing the complexities of certificates. It offers user authentication, credential management, certificate handling, and traffic encryption, ensuring pods can securely validate one another when necessary.

Galley

Galley is responsible for essential configuration tasks within your service mesh. It validates, processes, and distributes configuration changes throughout the Istio control plane. For example, when you apply a new policy, Galley ingests the configuration, validates its accuracy, processes it for the intended components, and seamlessly disseminates it within the service mesh. In essence, Galley serves as the interface through which the Istio control plane interacts with the underlying APIs, facilitating the smooth management of your service mesh.

Now, let's dive deep into understanding the data plane architecture.

The data plane architecture

The data plane component of Istio is composed of **envoy proxies**, **ingress gateways**, and **egress gateways**.

Envoy proxies

Envoy proxies play a pivotal role in enabling various aspects of your service mesh. These **Layer 7** proxies are uniquely capable of making crucial decisions based on the content of the messages they

handle, and they are the sole components that directly interact with your business traffic. Here's how these envoy proxies contribute to the functionality of Istio:

- **Traffic control**: They provide fine-grained control over how traffic flows within your service mesh, allowing you to define routing rules for various types of traffic, including **HTTP**, **TCP**, **WebSockets**, and **gRPC**.
- **Security and authentication**: Envoy proxies enforce **identity and access management**, ensuring that only authorized pods can interact with one another. They implement **mutual TLS** and **traffic encryption** to prevent **man-in-the-middle attacks** and offer features such as rate limiting to safeguard against runaway costs and **denial-of-service attacks**.
- **Network resiliency**: They enhance network resiliency by supporting features such as **retries**, **failover**, **circuit breaking**, and **fault injection** to maintain the reliability and robustness of your services.

Next, let's look at Ingress and egress gateways.

Ingress and egress gateways

In Istio, ingress is a collection of one or more envoy proxies, which Pilot dynamically configures upon their deployment. These envoy proxies are crucial in controlling and routing incoming external traffic into your service mesh, ensuring that it is appropriately directed to the relevant services based on defined routing rules and policies. This dynamic configuration allows Istio to effectively manage and secure external traffic flows without requiring extensive manual intervention, ensuring that your applications can operate efficiently and securely within the service mesh.

Egress gateways are similar to ingress gateways but they work on outgoing traffic instead. To understand this better, let's use *Figure 15.3* as a reference and understand the traffic flow through **Service A** and **Service B**.

In this architecture, traffic within the service mesh follows a structured path through **Ingress**, microservices (**Service A** and **Service B**), and **Egress**, ensuring efficient routing and security measures. Let's break down the flow of a traffic packet through your service mesh.

Ingress

Traffic enters the service mesh through an ingress resource, which is essentially a cluster of envoy proxies. Pilot configures these envoy proxies upon deployment. Ingress proxies are aware of their backends due to configurations based on Kubernetes service endpoints. Ingress proxies conduct health checks, perform load balancing, and make intelligent routing decisions based on metrics such as load, packets, quotas, and traffic balancing.

Service A

Once Ingress routes the traffic to a pod, it encounters the sidecar proxy container of the Service A pod, not the actual microservice container. The envoy proxy and the microservice container share the same network namespace within the pod and have identical IP addresses and IP Table rules. The envoy proxy takes control of the pod, handling all traffic passing through it. The proxy interacts with Citadel to enforce policies, checks whether traffic needs encryption, and establishes TLS connections with the backend pod.

Service B

Service A's encrypted packet is sent to Service B, where similar steps are followed. Service B's proxy verifies the sender's identity through a TLS handshake with the source proxy. Upon establishing trust, the packet is forwarded to the Service B container, continuing the flow toward the egress layer.

Egress

The egress resource manages outbound traffic from the mesh. Egress defines which traffic can exit the mesh and employs Pilot for configuration, similar to the ingress layer. Egress resources enable the implementation of policies restricting outbound traffic to only necessary services.

Telemetry data collection

Throughout these steps, proxies collect telemetry data from the traffic. This telemetry data is sent to **Prometheus** for storage and analysis. This data can be visualized in **Grafana**, offering insights into the service mesh's behavior. The telemetry data can also be sent to external tools such as **ELK** for more in-depth analysis and machine learning applications on metrics collected.

This structured flow ensures traffic moves securely and efficiently through the service mesh while providing valuable insights for monitoring, analysis, and decision-making processes.

Now that we've understood the Istio architecture and its features, let's go ahead and see how we can install it.

Installing Istio

The general way of installing Istio is to download Istio using the provided link and run a shell, which will install Istio on our system, including the **istioctl** component. Then, we need to use **istioctl** to install Istio within a Kubernetes cluster. However, since we're using GitOps, we will use the GitOps principles to install it. Istio offers another method to install Istio – that is, using Helm. Since we know that Argo CD supports Helm, we will use that instead.