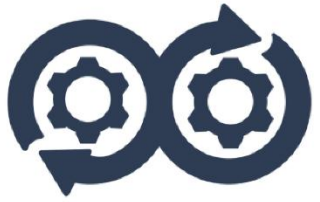
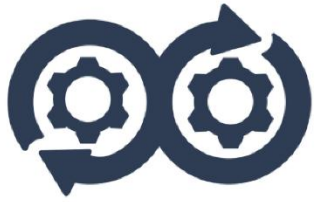
Automate Infrastructure

https://jruels.github.io/gh-auto-infra

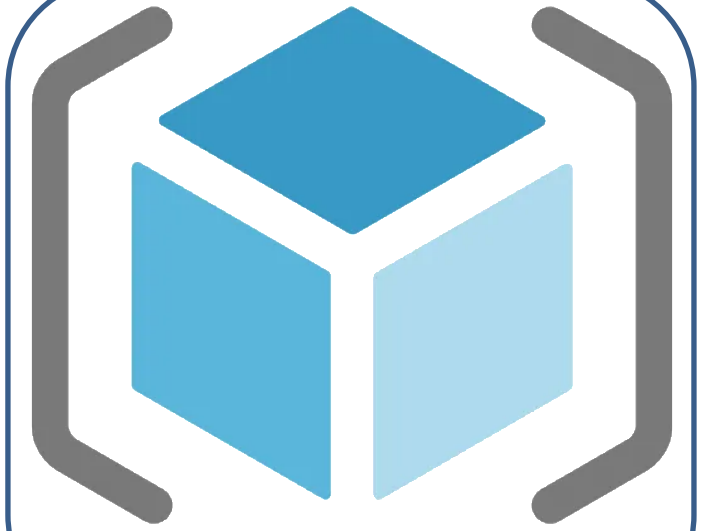


Terraform



Why Terraform?

AWS CloudFormation



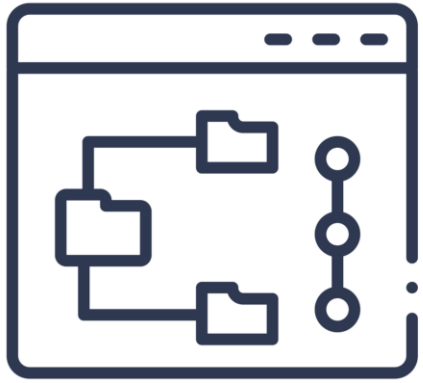
Azure ARM Templates

GCP Deployment manager

Each cloud has its own YAML or JSON based provisioning tool.
Terraform can be used across *all* major cloud providers and VM hypervisors.

Automating Infrastructure

Provisioning resources

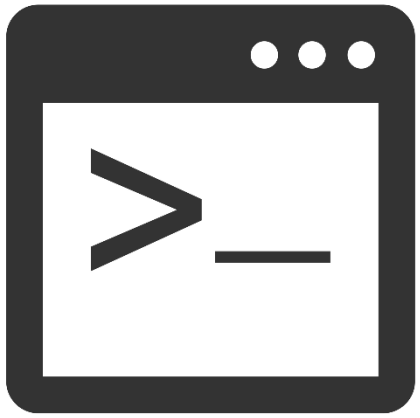


Version Control

Plan Updates

Reusable Templates

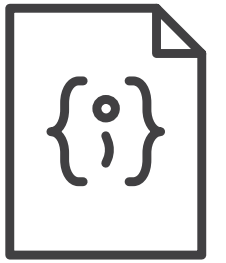
Terraform components



Terraform executable

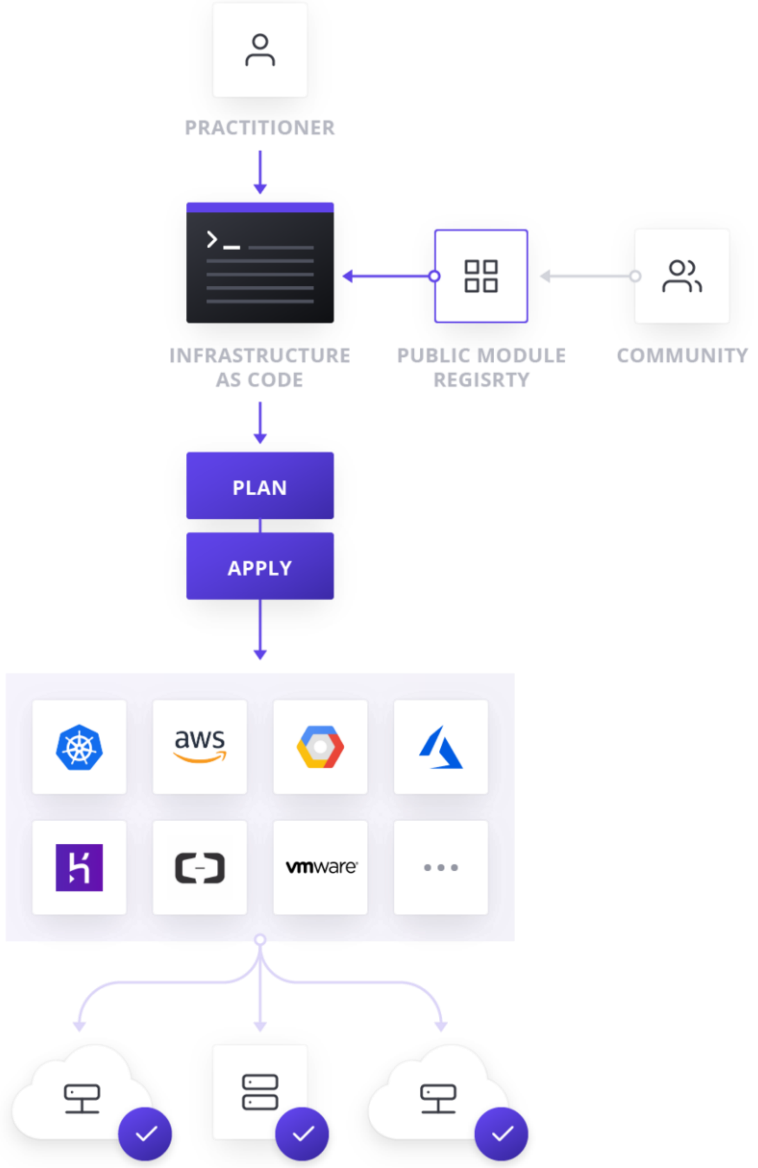
Terraform files

Terraform plugins

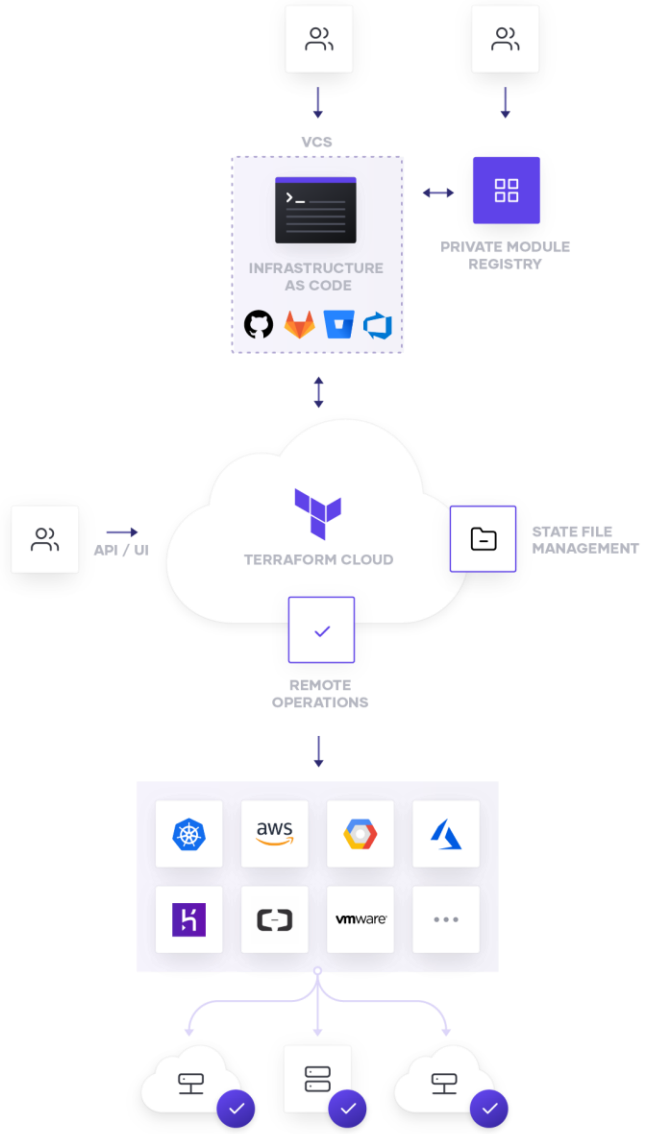


Terraform state

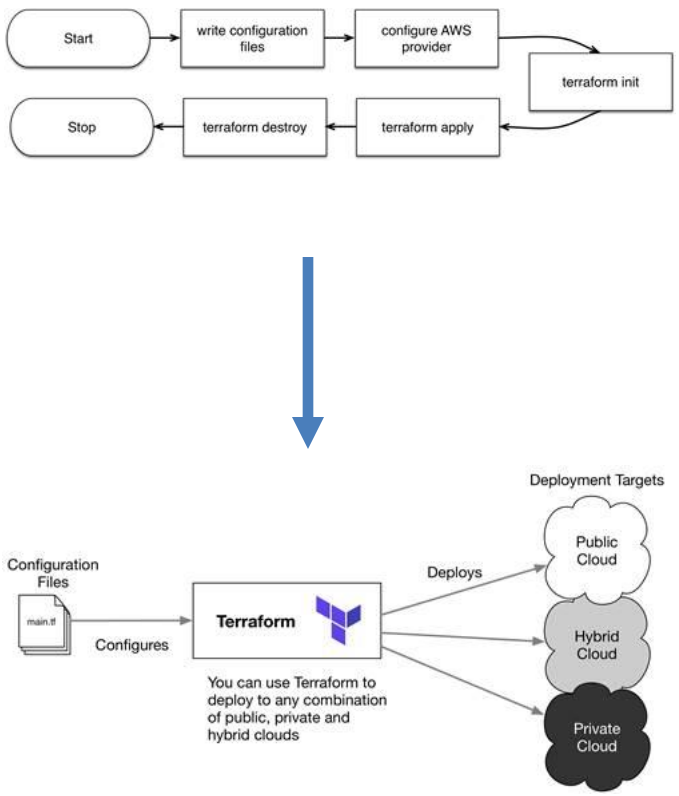
Terraform architecture



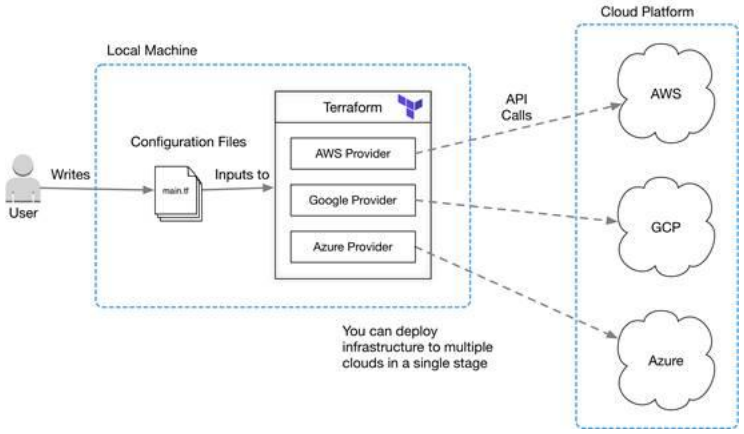
Terraform architecture

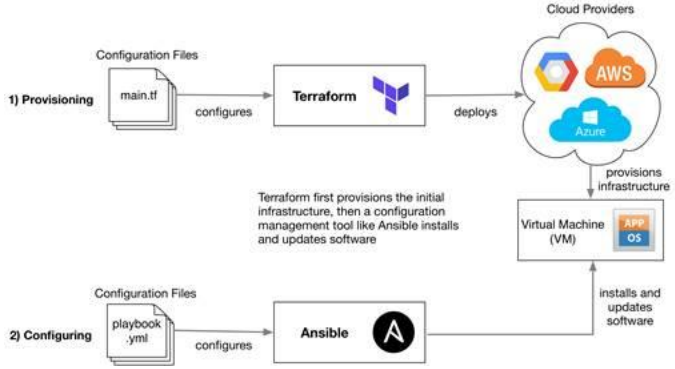


Terraform architecture

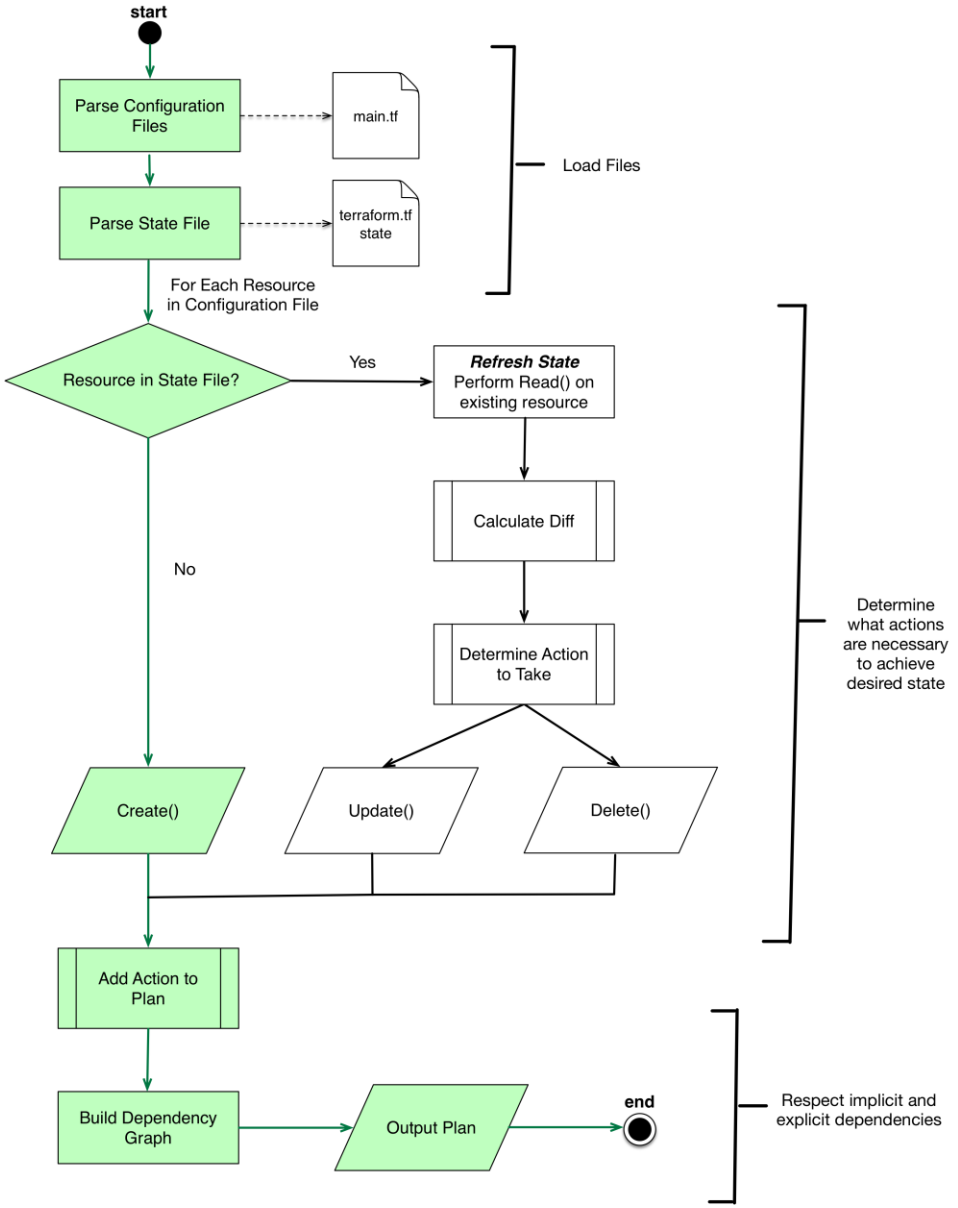


Terraform architecture





Terraform architecture



Terraform vs JSON

ARM JSON:

```
"name" : "[concat(parameters('PilotServerName'), '3')]",
```

Terraform:

```
name = "${var.PilotServerName}3"
```

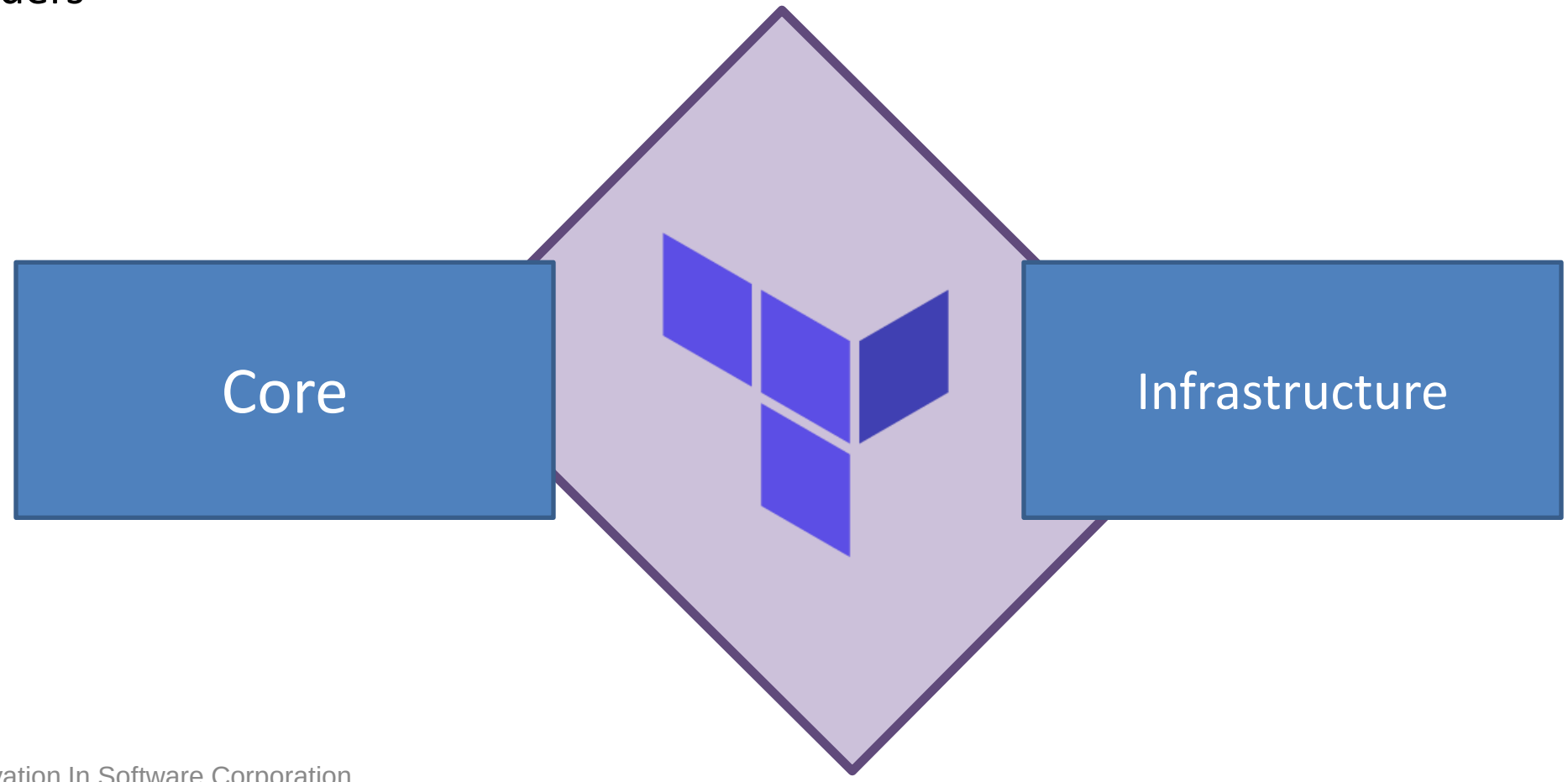
Terraform code (HCL) is "easy" to learn and "easy" to read. It is also more compact than equivalent JSON configuration.

terraform model

Terraform code is broken into three parts: Core Terraform, Modules and Providers

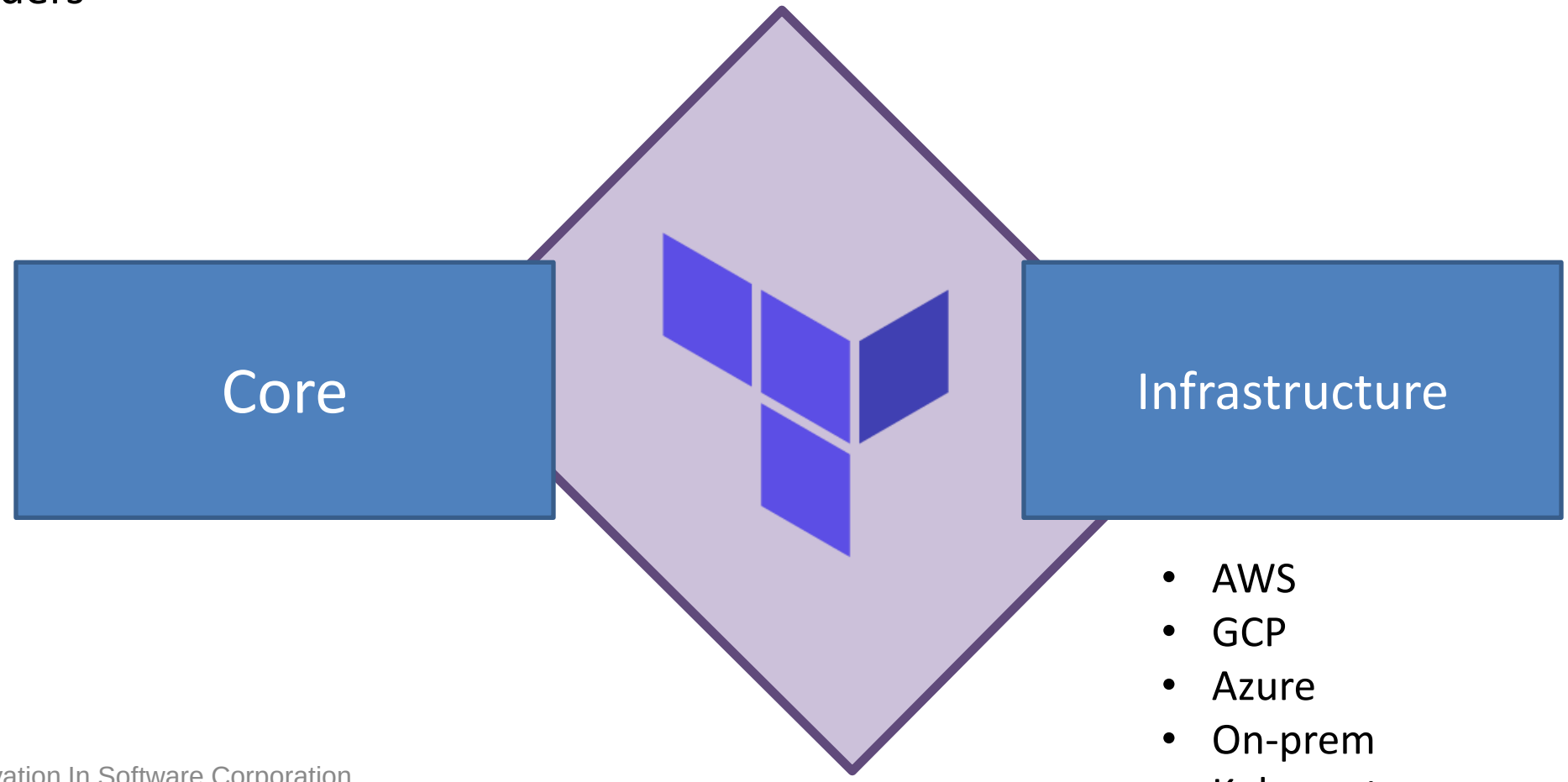
terraform model

Terraform code is broken into three parts: Core Terraform, Modules and Providers



terraform model

Terraform code is broken into three parts: Core Terraform, Modules and Providers



- AWS
- GCP
- Azure
- On-prem
- Kubernetes

configuration files

Terraform has a pretty confusing file hierarchy. We are going to start with the basics.

`provider.tf` - specify provider information

`main.tf` - Create infrastructure resources.

`variables.tf` - define values for variables in main.tf

Terraform files

```
##Amazon Infrastructure
provider "aws" {
  region = "${var.aws_region}"
}
```

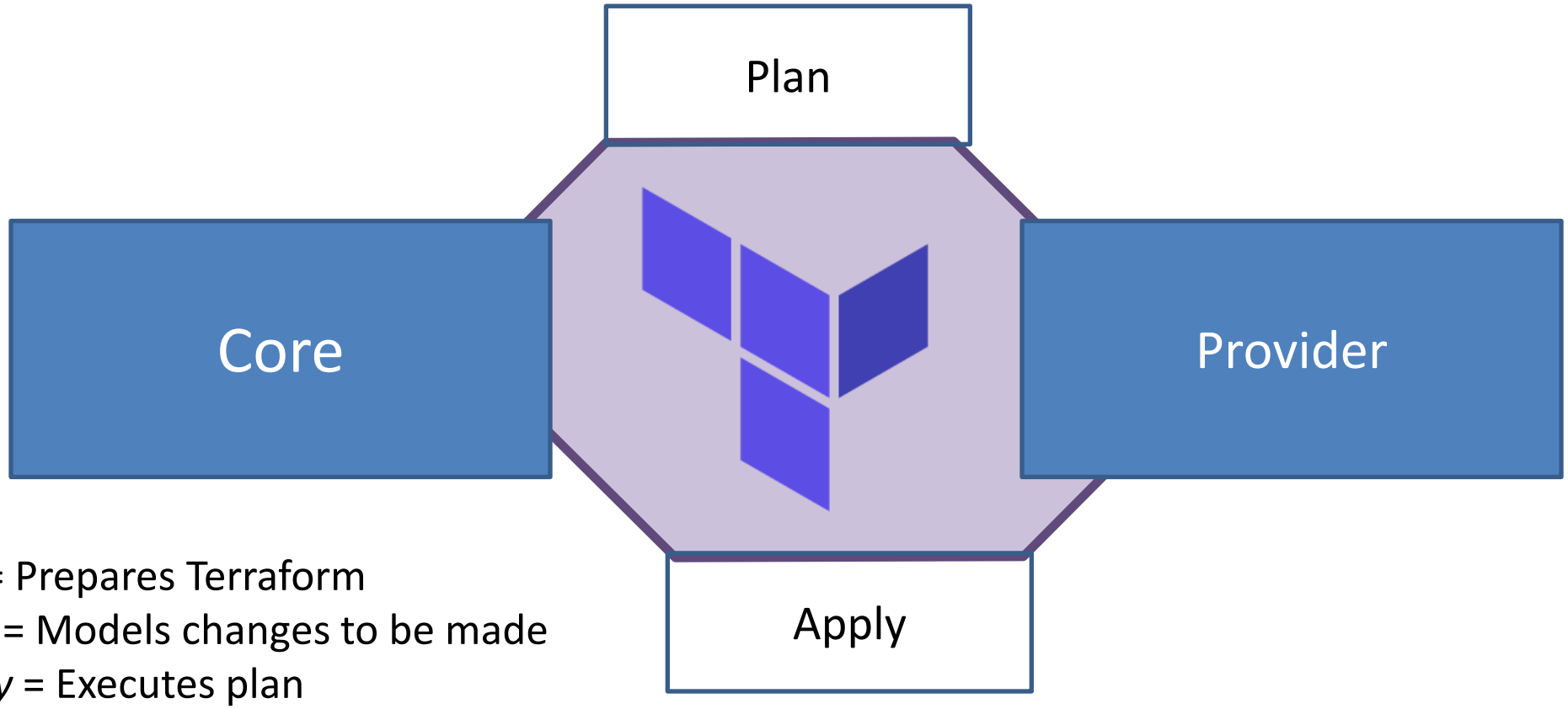
- Provider defines what infrastructure Terraform will be managing
  - Pass variables to provider
    - Region, Flavor, Credentials

Terraform files

- core = Terraform language, logic, and tooling.
- provider = Pluggable code to allow Terraform to talk to vendor APIs
- modules = A container for multiple resources that are used together.

terraform workflow

Terraform workflow consists of three common commands, Init, Plan and Apply.

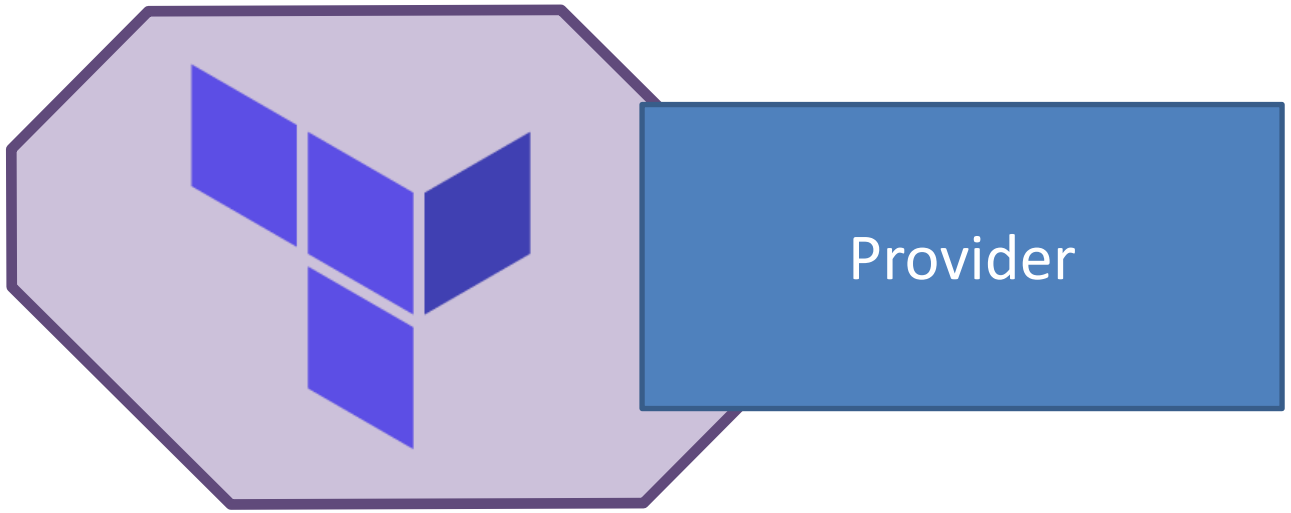


- *init* = Prepares Terraform
- *plan* = Models changes to be made
- *apply* = Executes plan

terraform providers

Terraform has over 150 "supported" providers.

- Clouds (AWS, GCP, Rackspace, Azure etc.)
- Version control (GitHub, GitLab, Bitbucket)
- Software (Grafana, Consul, Docker, Kubernetes)
- more!



Terraform provider configuration

```
provider "azurerm" {
  version = "=1.30.1"
}
```

Terraform allows you to configure the providers in code. configure specific versions or pull in latest.

Terraform provider configuration

```
provider "azurerm" {
  version = "=1.30.1"
  subscription_id = "SUBSCRIPTION-ID"
  client_id       = "CLIENT-ID"
  client_secret   = "CLIENT-SECRET"
  tenant_id       = "TENANT_ID"
}
```

Providers allow you to authenticate in the code block. **This is a very BAD idea.**

Terraform provider configuration

For Azure that can be az login, Managed Service Identity, or environment variables.

```
az login
```

```
export ARM_TENANT_ID=
export ARM_SUBSCRIPTION_ID=
export ARM_CLIENT_ID=
export ARM_CLIENT_SECRET=
```

Terraform provider configuration

Terraform will use the native tool's method of authentication. Environment variables, shared credentials files or static credentials.

```
provider "aws" {}
```

```
export AWS_ACCESS_KEY_ID=
export AWS_SECRET_ACCESS_KEY=
```

Terraform provider configuration

Terraform will use the native tool's method of authentication. Environment variables, shared credentials files or static credentials.

```
provider "aws" {
  region                  = "us-west-2"
  shared_credentials_file = "~/.aws/creds"
  profile                 = "customprofile"
}
```

Terraform provider configuration

Terraform will use the native tool's method of authentication. Environment variables, shared credentials files or static credentials.

```
provider "aws" {
  access_key     = "MYKEY"
  secret_key     = "MYSECRET"
}
```

Terraform Resource

Every Terraform resource is structured the exact same way.

```
resource "type" "name" {
  parameter = "foo"
  parameter2 = "bar"
  list = ["one", "two", "three"]
```

resource = top level keyword

Terraform Resource

Every Terraform resource is structured the exact same way.

```
resource "type" "name" {
  parameter = "foo"
  parameter2 = "bar"
  list = ["one", "two", "three"]
```

type = this is the name of the resource. The first part tells you which provider is belongs to. Example: `azurerm_virtual_machine`. This means the provider is Azure and the specific type of resources is a virtual machine.

Terraform Resource

Every Terraform resource is structured the exact same way.

```
resource "type" "name" {
  parameter = "foo"
  parameter2 = "bar"
  list = ["one", "two", "three"]
```

name = arbitrary name to refer to resource. Used internally by TF and cannot be a variable.

resources (building blocks)

Create an Azure Resource Group.

- Azure requires all resources be assigned to a resource group.

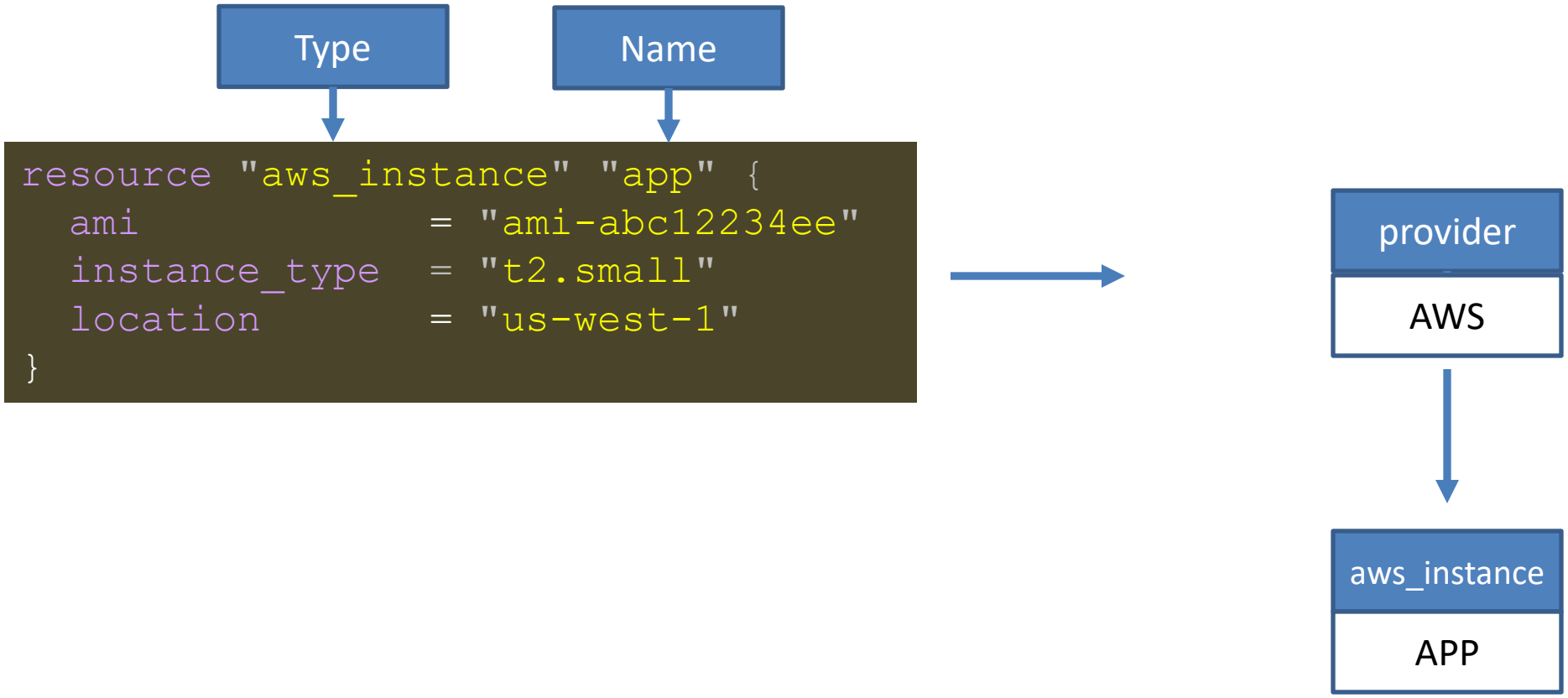
```
resource "azurerm_resource_group" "training" {
  name     = "training-workshop"
  location = "westus"
}
```

resources (building blocks)

TIP: You can assign random names to resources.

```
resource "random_id" "project"name" {
  byte_length = 4
}
resource "azurerm_resource_group" "training" {
  name       = "${random_id.project_name.hex}-training"
  location   = "westus"
}
```

resources (building blocks)



variables

Set default values for variables in `variables.tf`
If default values are omitted, user will be prompted.

```
##AWS Specific Vars
variable "aws_master_count" {
  default = 10
}
variable "aws_worker_count" {
  default = 20
}
variable "aws_key_name" {
  default = "k8s"
}
```

Terraform state file

Terraform is a *stateful* application.

- Keeps track of everything you build inside of a state file
  - terraform.tfstate
- State file is Terraform's source of truth

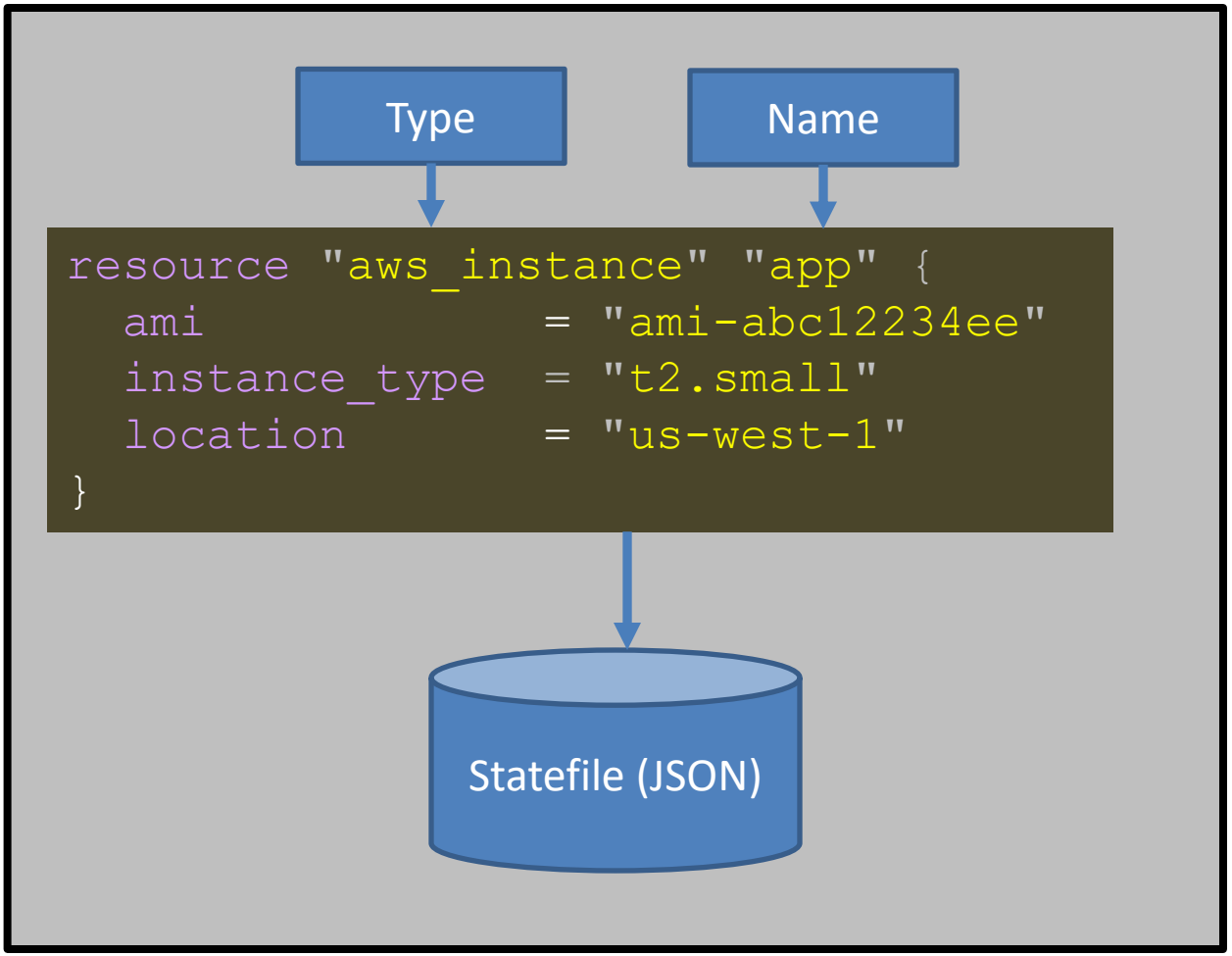
```
{
  "version": 3,
  "terraform_version": "0.12.29",
  "serial": 6,
  "lineage": "0a209e29-de63-9e87-2cd2-4f2071717cee",
  "modules": [
      {
         "path": [
            "root"
         ],
         "outputs": {
            "MySQL_Server_FQDN": {
               "value": "labtest1-mysql-server.azure.com
```

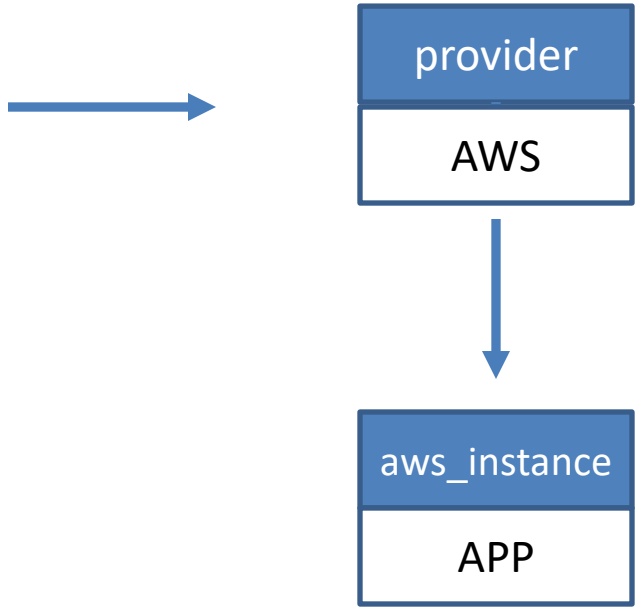
Terraform state file

Terraform supports local and remote state file storage.

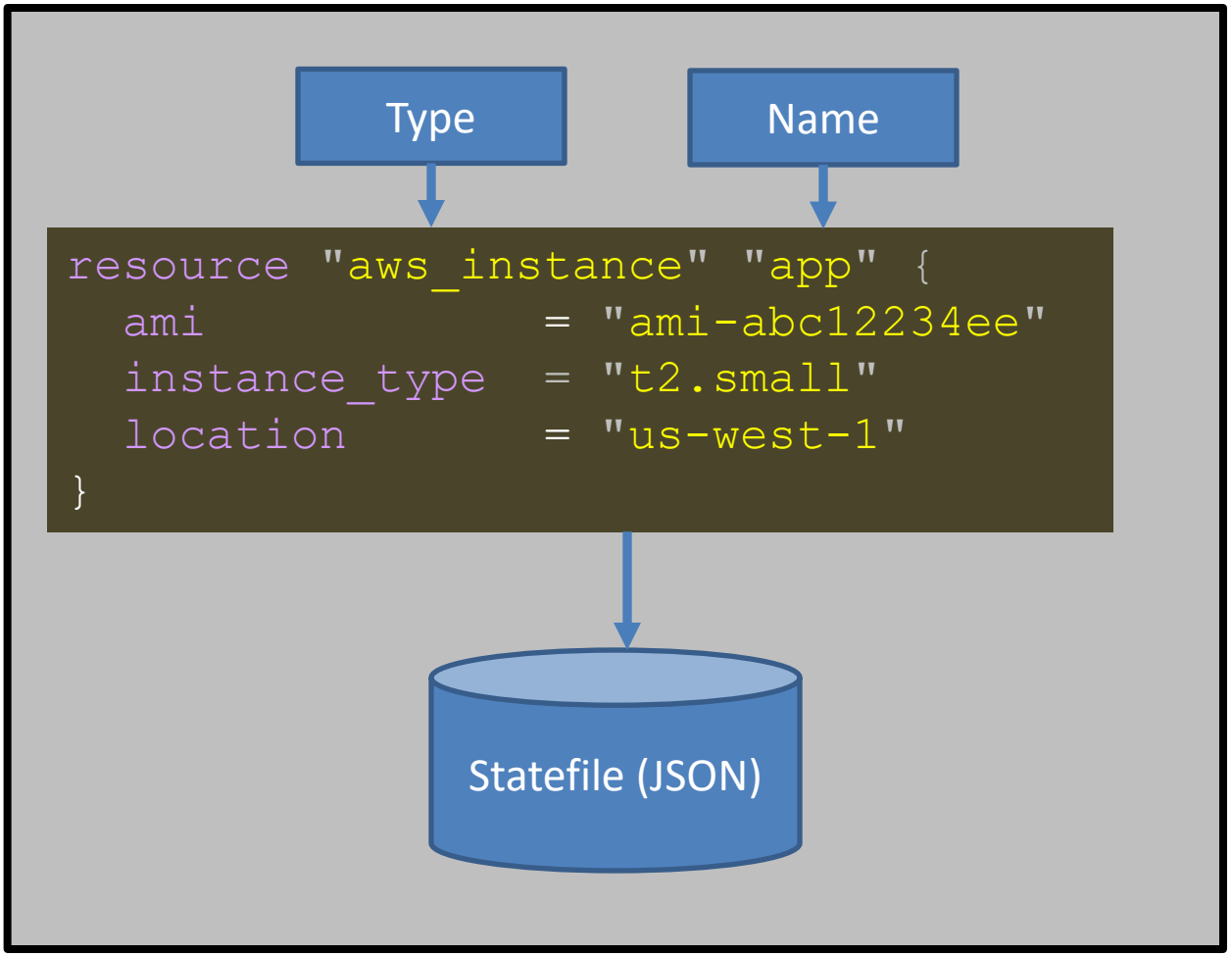
- Default is local storage
- Remote backends:
  - S3, Azure Storage, Google Cloud Storage

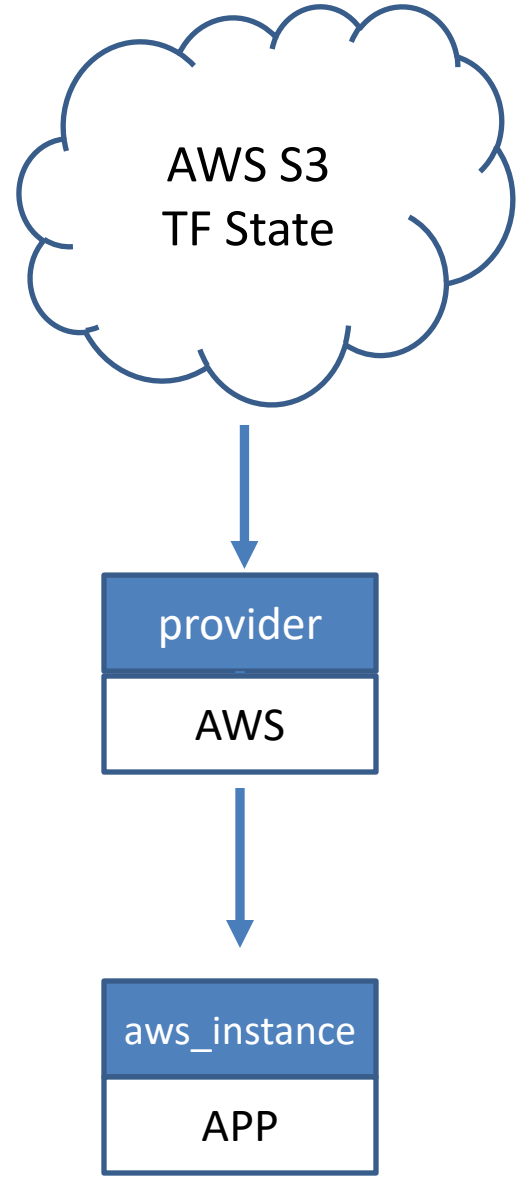
Terraform state (local)



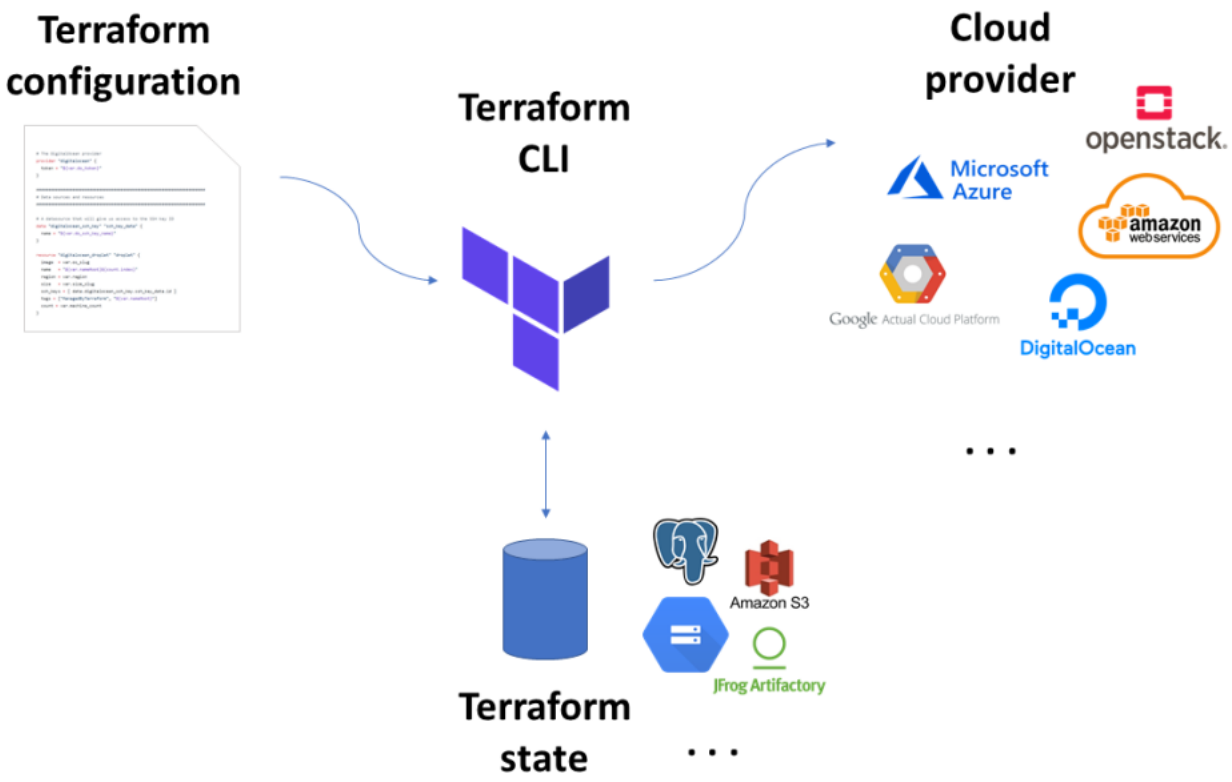


Terraform state (remote)





Terraform state (remote)



Terraform CLI

Run `terraform` command

Output:

```
Usage: terraform [-version] [-help] <command> [args]

The available commands for execution are listed below.
The most common, useful commands are shown first, followed by
less common or more advanced commands. If you're just getting
started with Terraform, stick with the common commands. For the
other commands, please read the help and docs before usage.

Common commands:
    apply              Builds or changes infrastructure
    console            Interactive console for Terraform
interpolations
    destroy            Destroy Terraform-managed infrastructure
    env                Workspace management
    fmt                Rewrites config files to canonical format
    get                Download and install modules for the
configuration
    graph              Create a visual graph of Terraform
resources
```

Terraform CLI

Command:

```
terraform init
```

Output:

```
Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "docker"...
```

Terraform fetches any required providers and modules and stores them in the .terraform directory. Check it out and you'll see a plugins folder.

Terraform CLI

Command:

```
terraform validate
```

Validate all of the Terraform files in the current directory. Validation includes basic syntax check as well as all variables declared in the configuration are specified.

Terraform CLI

Command:

```
terraform plan
```

Output:

```
Terraform will perform the following actions:

  + aws_instance.aws-k8s-master
      id:                                   <computed>
      ami:                                  "ami-01b45..."
      instance_type:                        "t3.small"
```

Plan is used to show what Terraform will do if applied. It is a dry run and does not make any changes.

Terraform CLI

Command:

```
terraform apply
```

Output:

```
❯ terraform apply "rapid-app.out"
aws_security_group.k8s_sg: Creating...
  arn:                                      "" => "<computed>"
  description:                              "" => "Allow all
inbound traffic necessary for k8s"
  egress.#:                                 "" => "1"
  egress.482069346.cidr_blocks.#:           "" => "1"
  egress.482069346.cidr_blocks.0:           "" => "0.0.0.0/0"
```

Performs the actions defined in the plan

Terraform CLI

Command:

```
terraform destroy [-auto-approve]
```

Destroys all the resources in the state file.
-auto-approve (don't prompt for confirmation)

Terraform ASA provider

```
provider "ciscoasa" {
  api_url           = "https://10.0.0.5"
  username          = "admin"
  password.         = "YOUR SECRET PASSWORD"
  ssl_no_verify     = false
}
```

Providers allow you to authenticate in the code block. **This is a very BAD idea.**

Use environment variables:

`CISCOASA_USER`
`CISCOASA_PASSWORD`

Terraform ASA ACL resource

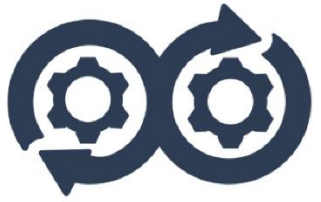
```
resource "ciscoasa_acl" "foo" {
  name = "aclname"
  rule {
    source                = "192.168.10.5/32"
    destination           = "192.168.15.0/25"
    destination_service = "tcp/443"
  }
  rule {
    source                = "192.168.10.0/24"
```

Example of creating ACL

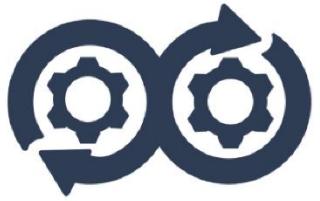
Terraform ASA static route resource

```
resource "ciscoasa_static_route" "ipv4_static_route" {
  interface               = "inside"
  network                 = "10.254.0.0/16"
  gateway                 = "192.168.10.20"
  }

resource "ciscoasa_static_route" "ipv6_static_route" {
  interface               = "inside"
  network                 = "fd01:1337::/64"
  gateway                 = "fd01:1338::1"
```

Lab: Terraform Variables and Parameters



Ansible



Why Ansible?

Efficient

- Agentless architecture
- Extendable through modules

Secure

- No agent (smaller attack surface)
- Runs on SSH

Simple

- Does not require hours of troubleshooting SSL

Inventory

Ansible supports multiple systems in your environment

- Servers
- Networking
- Containers
- Storage
- Cloud APIs
- More!

Inventory

Ansible keeps track of resources it's managing through the inventory.
There are two types of inventory

- Static (local file, that must be updated)
- Dynamically (Queries cloud API to find out which resources are available)

Inventory

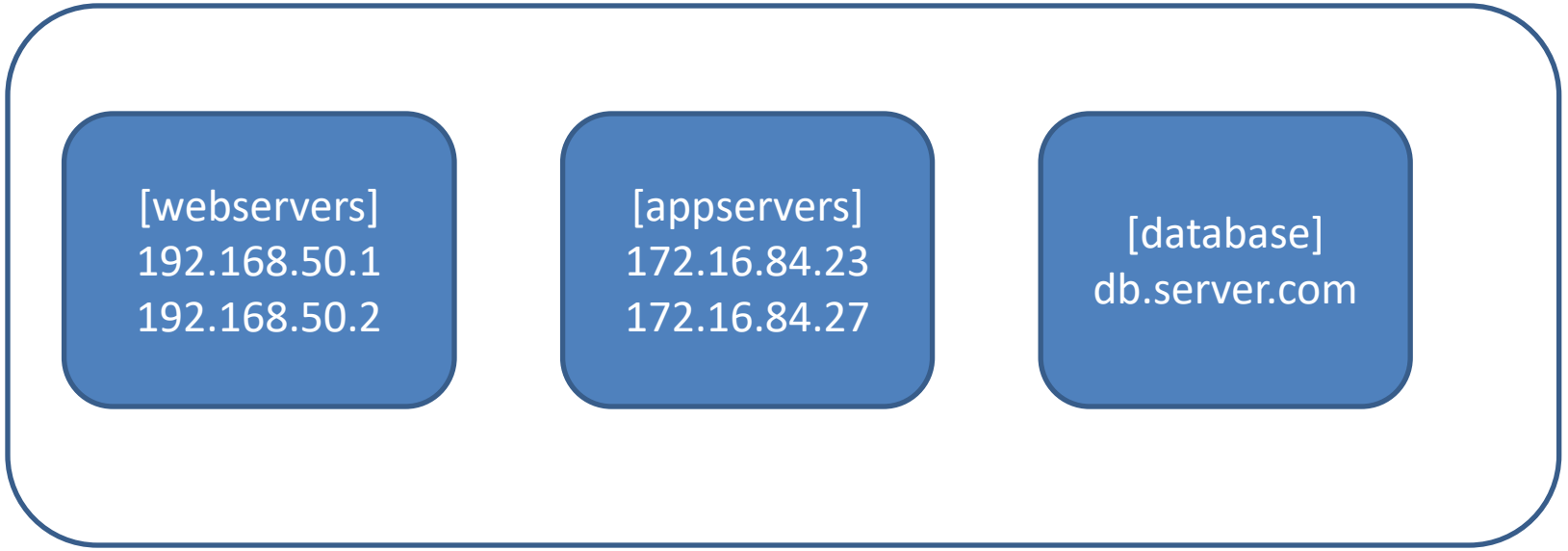
Ansible's inventory is located in /etc/ansible/hosts by default, but can be overwritten on command line

```
ansible-playbook -i /path/to/inventory playbook.yml
```

Inventory

Ansible's inventory is located in /etc/ansible/hosts by default, but can be overwritten on command line

```
ansible-playbook -i /path/to/inventory playbook.yml
```

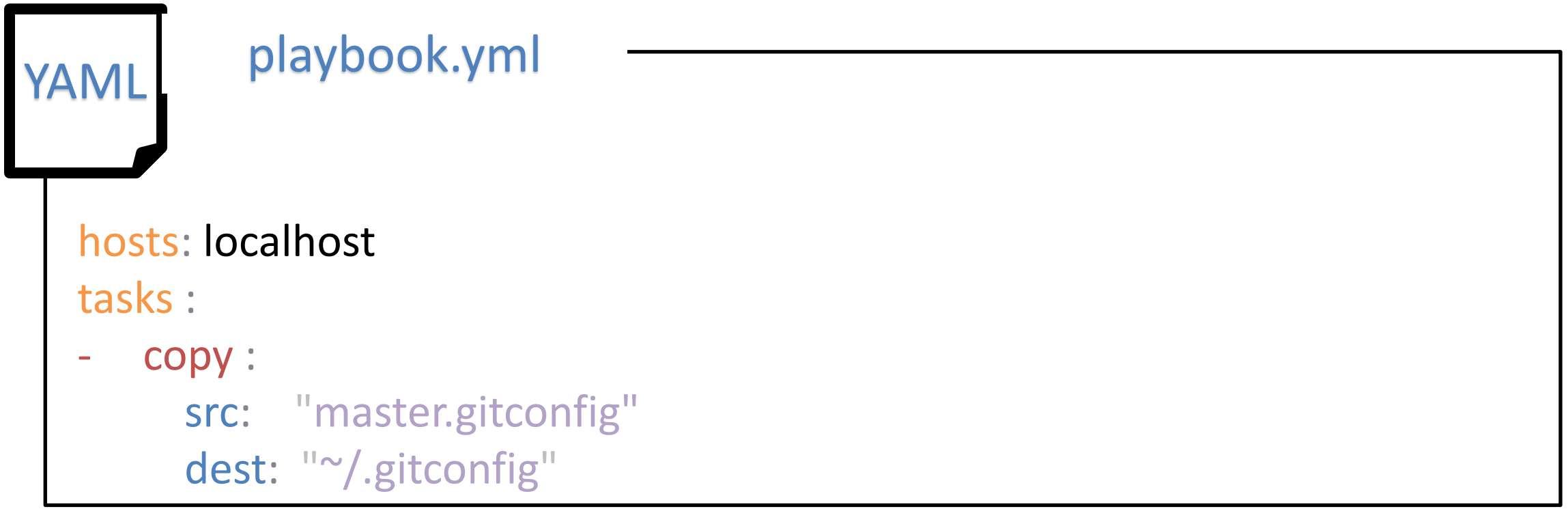


Inventory

Ansible inventory supports variables

```
[back-end-servers]
database ansible_host=10.0.0.1 ansible_port=22 ansible_user=postgres
webserver ansible_host=10.0.0.2 ansible_port=22 ansible_user=root
```

Playbook



Playbook

playbook.yml

```
hosts: localhost
tasks :
-    copy :
        src:    "master.gitconfig"
        dest:  "~/.gitconfig"
```

```
ansible-playbook playbook.yml
```

More complex playbook

```
- name: Install Docker
  hosts: tag_role_k8s_master:tag_role_k8s_member
  remote_user: ubuntu
  gather_facts: no
  handlers:
    - include: roles/handlers/main.yml
  tasks:
    - package:
        name: "docker-ce"
        state: latest
    - user:
        name: "{{ ansible_ssh_user }}"
        groups: docker
        append: yes
      notify: restart docker
```

Name of Play

More complex playbook

```yaml
- name: Install Docker
  hosts: tag_role_k8s_master:tag_role_k8s_member
  remote_user: ubuntu
  gather_facts: no
  handlers:
    - include: roles/handlers/main.yml
  tasks:
    - package:
        name: "docker-ce"
        state: latest
    - user:
        name: "{{ ansible_ssh_user }}"
        groups: docker
        append: yes
      notify: restart docker
```

Hosts to run Play on

More complex playbook

```
- name: Install Docker
  hosts: tag_role_k8s_master:tag_role_k8s_member
  remote_user: ubuntu
  gather_facts: no
  handlers:
    - include: roles/handlers/main.yml
  tasks:
    - package:
        name: "docker-ce"
        state: latest
    - user:
        name: "{{ ansible_ssh_user }}"
        groups: docker
        append: yes
      notify: restart docker
```

User to run Play as

More complex playbook

```yaml
- name: Install Docker
  hosts: tag_role_k8s_master:tag_role_k8s_member
  remote_user: ubuntu
  gather_facts: no
  handlers:
    - include: roles/handlers/main.yml
  tasks:
    - package:
        name: "docker-ce"
        state: latest
    - user:
        name: "{{ ansible_ssh_user }}"
        groups: docker
        append: yes
      notify: restart docker
```

Define what to do after package is installed.

More complex playbook

```yaml
- name: Install Docker
  hosts: tag_role_k8s_master:tag_role_k8s_member
  remote_user: ubuntu
  gather_facts: no
  handlers:
    - include: roles/handlers/main.yml
  tasks:
    - package:
        name: "docker-ce"
        state: latest
    - user:
        name: "{{ ansible_ssh_user }}"
        groups: docker
        append: yes
      notify: restart docker
```

Call the defined handler

More complex playbook

```yaml
- name: Install Docker
  hosts: tag_role_k8s_master:tag_role_k8s_member
  remote_user: ubuntu
  gather_facts: no
  handlers:
    - include: roles/handlers/main.yml
  tasks:
    - package:
        name: "docker-ce"
        state: latest
    - user:
        name: "{{ ansible_ssh_user }}"
        groups: docker
        append: yes
      notify: restart docker
```

Tasks to run
- install Docker

More complex playbook

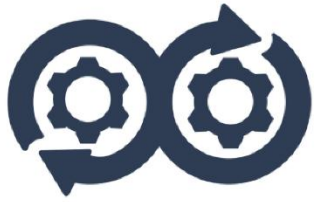
```
- name: Install Docker
  hosts: tag_role_k8s_master:tag_role_k8s_member
  remote_user: ubuntu
  gather_facts: no
  handlers:
    - include: roles/handlers/main.yml
  tasks:
    - package:
        name: "docker-ce"
        state: latest
    - user:
        name: "{{ ansible_ssh_user }}"
        groups: docker
        append: yes
      notify: restart docker
```

Create user account

Ansible loops

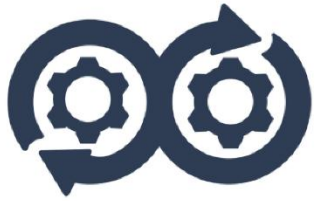
```
tasks:
   - package:
       name: "{{item}}"
       state: latest
     with_items:
       - apt-transport-https
       - ca-certificates
       - curl
       - software-properties-common
```

Demo: Ansible Playbook Repo

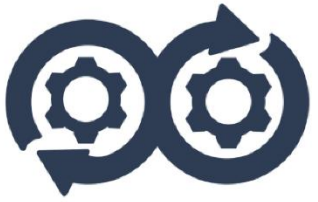


Lab: Ansible Error Handling

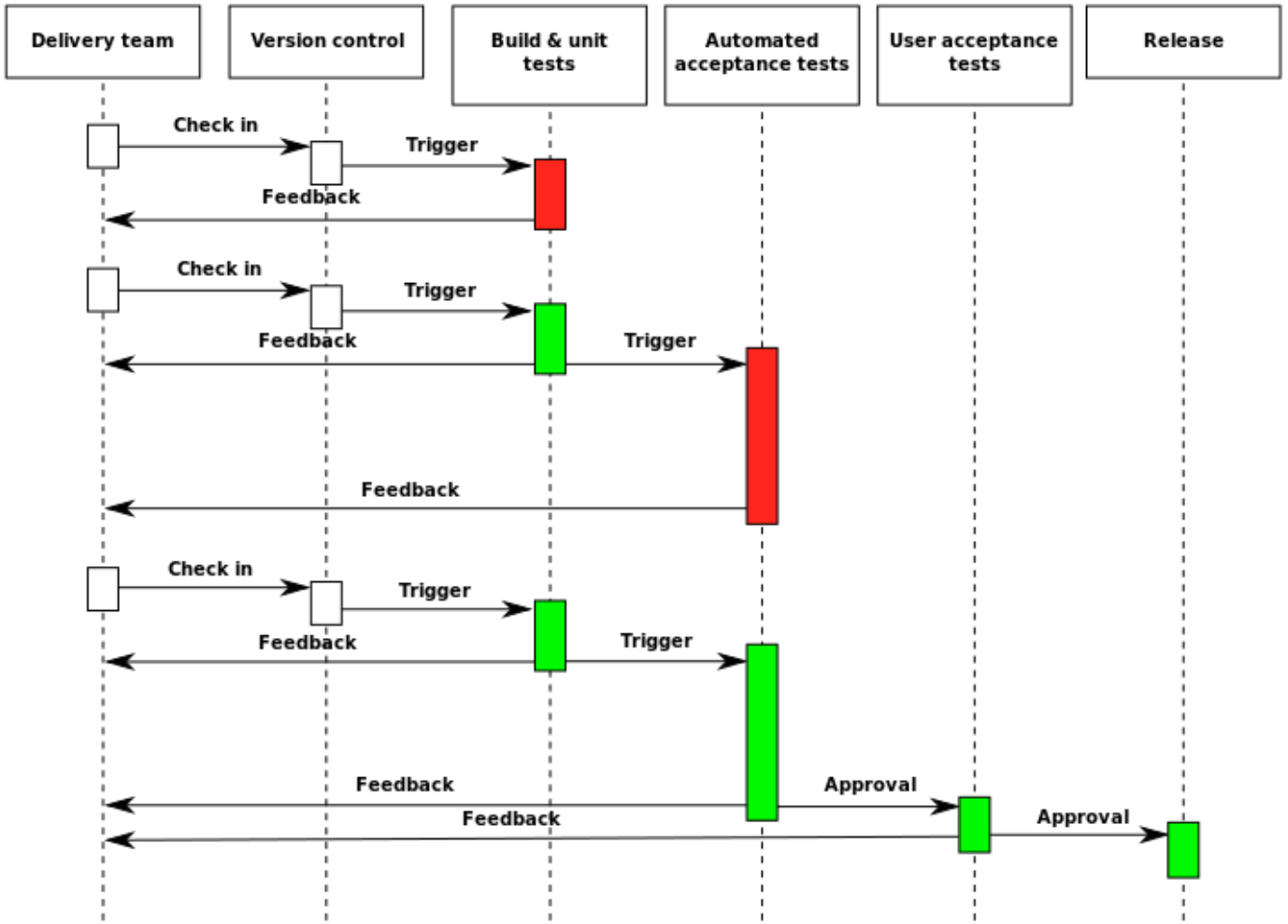
Lunch and Lab Resume at 1:30pm Eastern



What is Continuous Delivery?



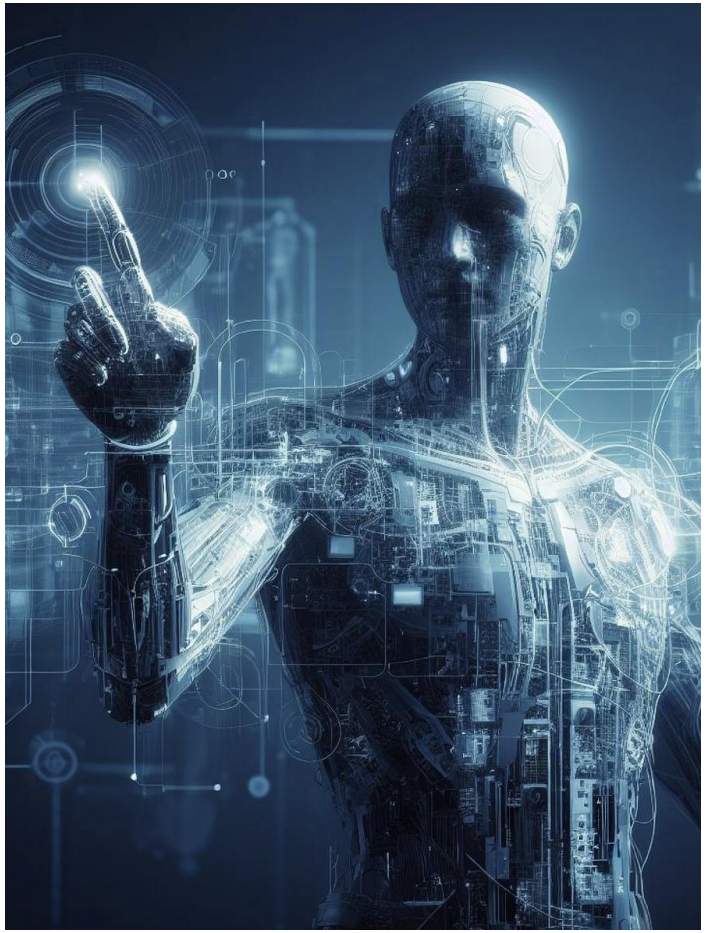
UNDERSTANDING CONTINUOUS DELIVERY



Continuous Delivery (CD) is a software development practice that ensures code changes are automatically built, tested, and prepared for deployment. It guarantees that every change is always in a deployable state, reducing the risk of errors and making releases faster and more reliable. By automating repetitive tasks like testing and packaging, CD allows teams to focus on delivering high-quality software.

CD enables frequent, incremental releases, ensuring updates, features, and bug fixes reach users quickly. For example, a SaaS platform using CD can deploy updates weekly or even daily, delivering value to customers while maintaining confidence in the software's stability.

KEY PRINCIPLES OF CONTINUOUS DELIVERY

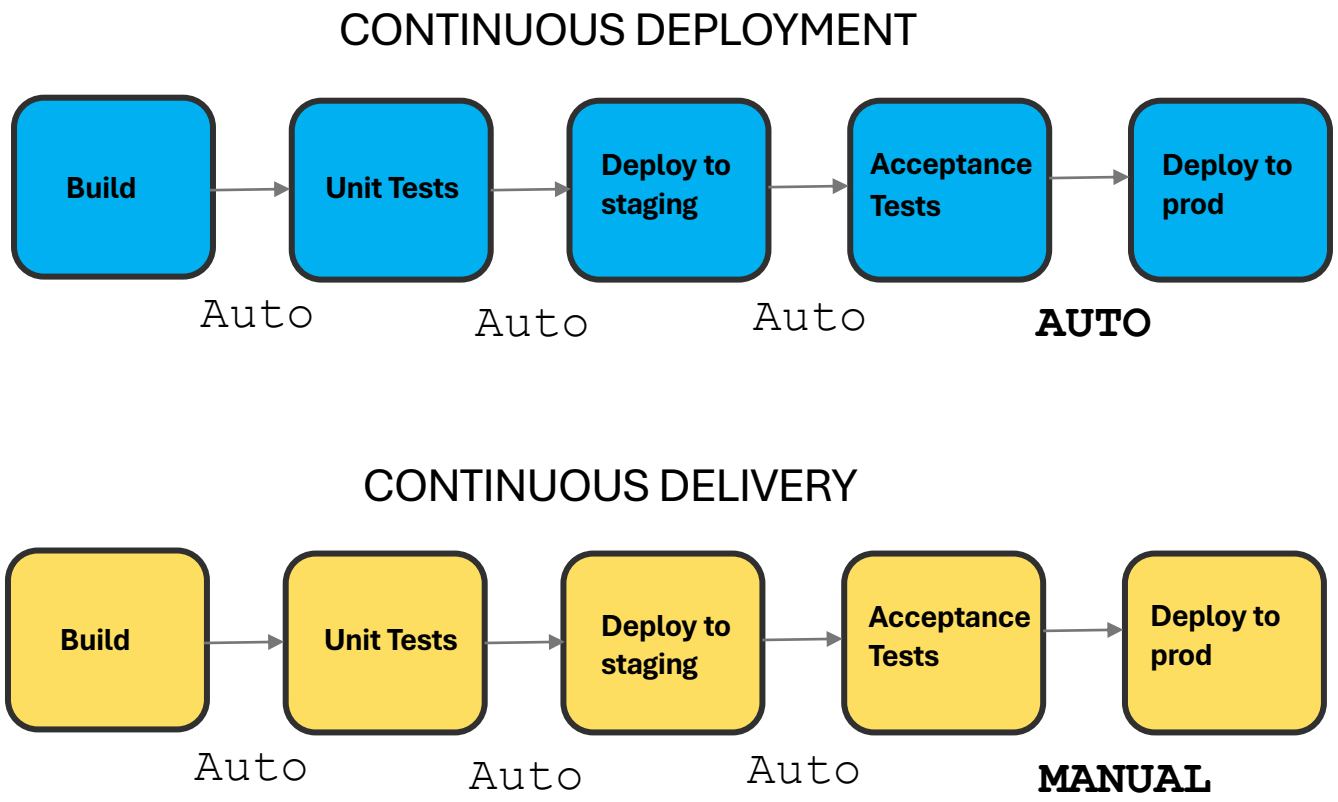


Continuous Delivery is guided by principles that ensure reliability and efficiency in delivering software:

- **Automation:** Automate repetitive tasks like building, testing, and deploying, minimizing manual intervention and errors.
- **Incremental Updates:** Break large updates into smaller, manageable changes, making them easier to test and deploy.
- **Frequent Testing:** Use automated tests at every stage to validate changes and catch bugs early.
- **Version Control:** Store all code, configurations, and dependencies in a version control system, ensuring consistency and traceability.
- **Production-Ready Code:** Ensure every change can be deployed to production, eliminating integration challenges.

These principles enable faster and safer delivery of high-quality software.

UNDERSTANDING CONTINUOUS DELIVERY



Understanding the distinction between **Continuous Delivery** and **Continuous Deployment** is essential:

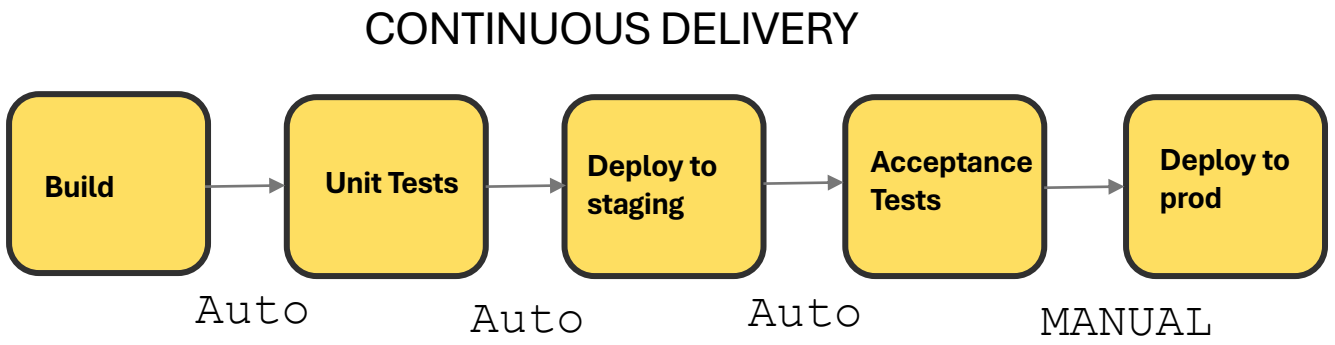
Continuous Delivery:

- Focuses on automating the preparation of code for deployment.
- Deployment to production is manual, providing a checkpoint for review.
- Suitable for teams requiring compliance or manual oversight.

Continuous Deployment:

- Extends CD by automating deployment to production.
- Every validated change is automatically released to users.
- Demands high confidence in testing and robust monitoring systems.

THE CONTNUOUS DELIVERY PIPELINE

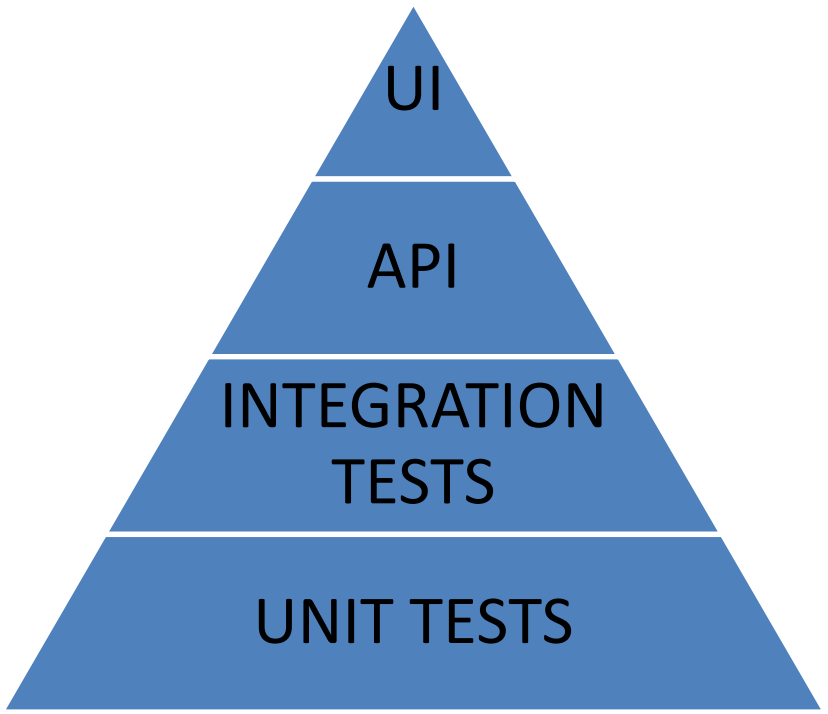


A **CD** pipeline automates the process of turning code changes into deployable builds. Key stages include:

- **Source:** Developers commit code changes to a version control system like Git, triggering the pipeline.
- **Build:** The application is compiled and packaged, resolving dependencies and preparing for deployment.
- **Test:** Automated tests (unit, integration, and end-to-end) validate the build to ensure functionality and quality.
- **Release:** Validated builds are deployed to staging or production environments.

For instance, in a banking application, the pipeline ensures that changes to transaction processing are rigorously tested before deployment.

IMPORTANCE OF AUTOMATED TESTING

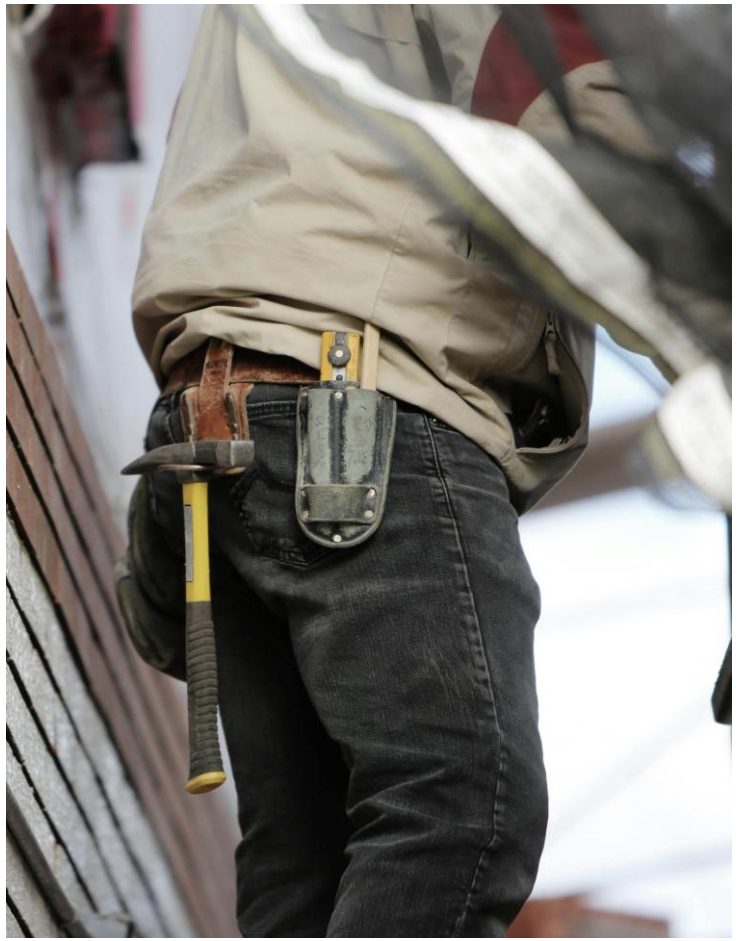


Automated testing is the backbone of Continuous Delivery, ensuring that each code change meets quality standards before progressing:

- **Unit Tests:** Test individual components or functions for correctness.
- **Integration Tests:** Validate that components interact properly.
- **API Tests:** Validate API endpoints by checking their functionality, response codes, and data accuracy under various conditions.
- **UI Tests:** Test the user interface to ensure elements are correctly displayed, interactive, and meet usability expectations.

For example, an automated test suite for a social media app might check if posts can be created, edited, and displayed correctly across devices.

CONTINUOUS DELIVERY TOOLS



A variety of tools support the implementation of Continuous Delivery:

- **CI/CD Platforms:** Tools like Jenkins, GitHub Actions, GitLab CI/CD, and Azure DevOps automate pipelines.
- **Containerization:** Docker and Kubernetes provide consistent environments across development, testing, and production.
- **Version Control:** Git tracks code changes, ensuring traceability and collaboration.
- **Testing Frameworks:** Tools like Selenium and Postman automate testing at different levels.

For example, a microservices architecture might use Kubernetes for container orchestration and Jenkins for pipeline automation.

CHALLENGES IN CONTINUOUS DELIVERY

While **Continuous Delivery** offers many benefits, it also comes with challenges:

- **Testing Complexity:** Writing and maintaining comprehensive tests for complex applications can be time-consuming.
- **Tool Integration:** Building a seamless pipeline involves integrating multiple tools and technologies.
- **Team Buy-In:** Adopting CD requires cultural shifts and collaboration across development, QA, and operations.
- **Legacy Systems:** Older systems may lack the infrastructure or compatibility needed for CD.

For instance, transitioning a legacy ERP system to a Continuous Delivery model might require significant investment in modernization.

BEST PRACTICES FOR CONTINUOUS DELIVERY

Adopting best practices can help teams implement Continuous Delivery successfully:

- **Automate Everything:** Automate builds, tests, deployments, and monitoring wherever possible.
- **Shift Left Testing:** Perform testing earlier in the pipeline to catch issues early.
- **Use Feature Flags:** Deploy features behind flags, allowing controlled rollouts.
- **Monitor Continuously:** Track performance and errors in real-time to address issues quickly.
- **Foster Collaboration:** Ensure alignment across development, QA, and operations teams.

For example, a retail application might use feature flags to enable a new checkout experience for specific user groups while monitoring performance.

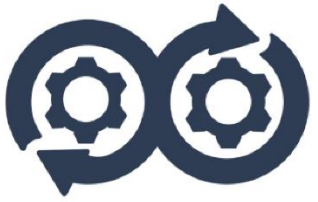
SUMMARY OF CONTINUOUS DELIVERY

Continuous Delivery is the practice of automating the preparation of code for deployment, ensuring it is always in a deployable state.

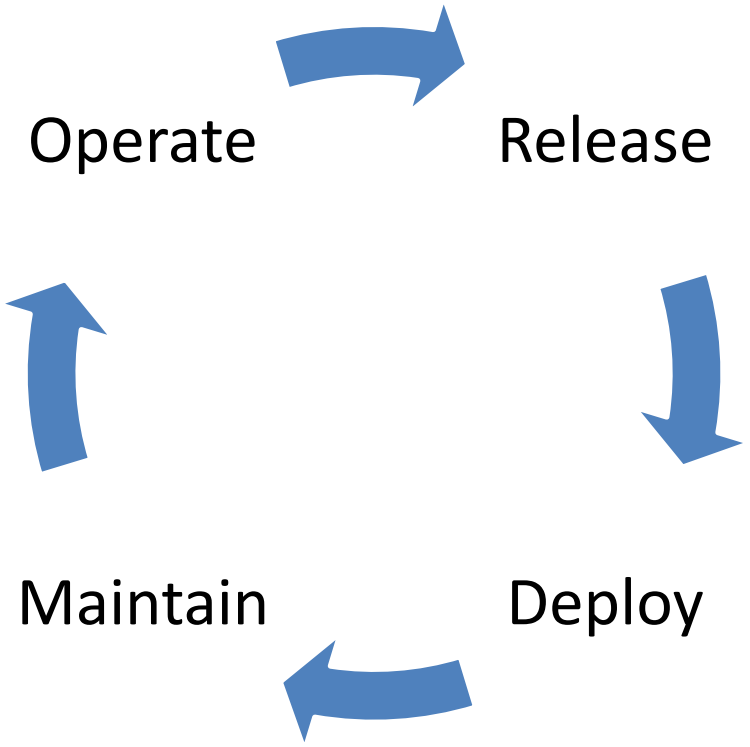
- It enables faster, safer, and more frequent releases through automation and testing.
- By reducing manual intervention and fostering collaboration, CD helps organizations deliver high-quality software consistently and efficiently.

For example, companies like Netflix and Amazon rely on Continuous Delivery to deploy updates multiple times daily, delivering features and fixes rapidly to millions of users.

CONTINUOUS DELIVERY: TYPICAL SETUP



OVERVIEW OF A TYPICAL CONTINUOUS DELIVERY SETUP



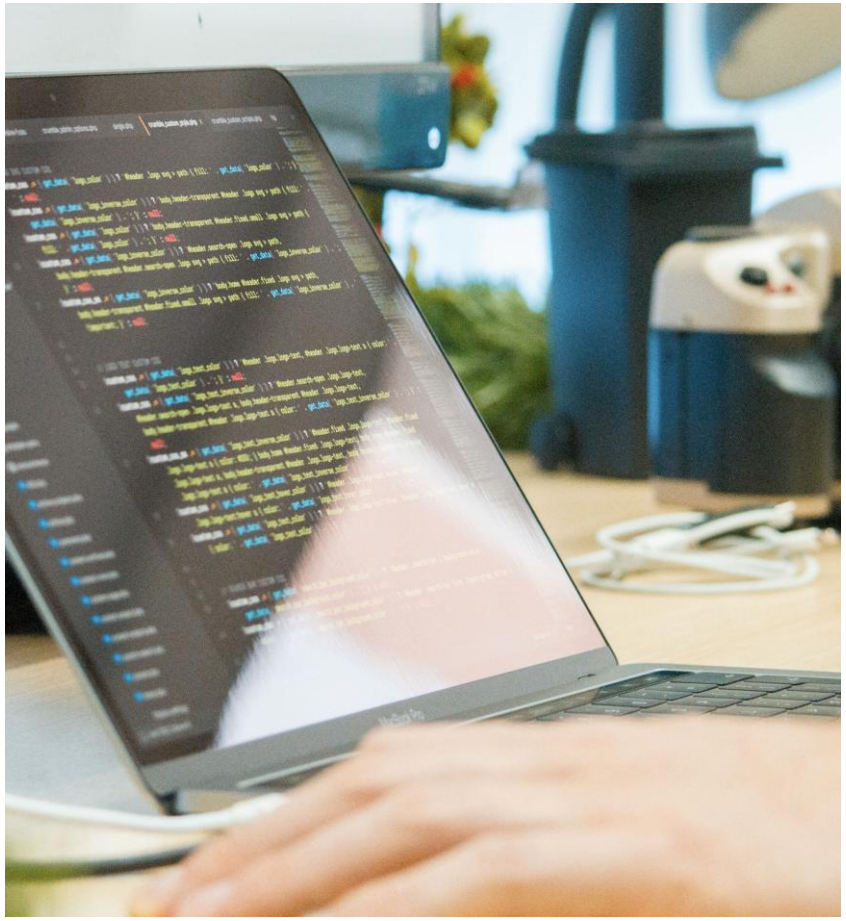
A typical Continuous Delivery setup automates the software delivery process, ensuring that code changes progress reliably from development to production. The setup includes tools and practices that streamline building, testing, and deploying software:

Key Components:

- **Source Code Management**
- **Automated Build Systems**
- **Testing Frameworks**
- **Deployment Pipelines**
- **Monitoring and Feedback Systems**

This structured approach ensures frequent, high-quality releases while minimizing manual intervention.

SOURCE CODE MANAGEMENT



Role in CD:

- Source Code Management (SCM) systems like Git, GitLab, or Bitbucket track code changes, enabling collaboration and traceability.
- Every code commit triggers the Continuous Delivery pipeline, initiating automated processes.

Best Practices:

- Use version control for all code, configurations, and dependencies.
- Implement branching strategies like GitFlow to manage feature development and releases.

Example: Developers commit code to the main branch, triggering automated testing and builds.

AUTOMATED BUILD SYSTEMS

Purpose:

- Build systems like Jenkins, GitHub Actions, or Azure DevOps automate the creation of deployable artifacts, such as binaries or containers.

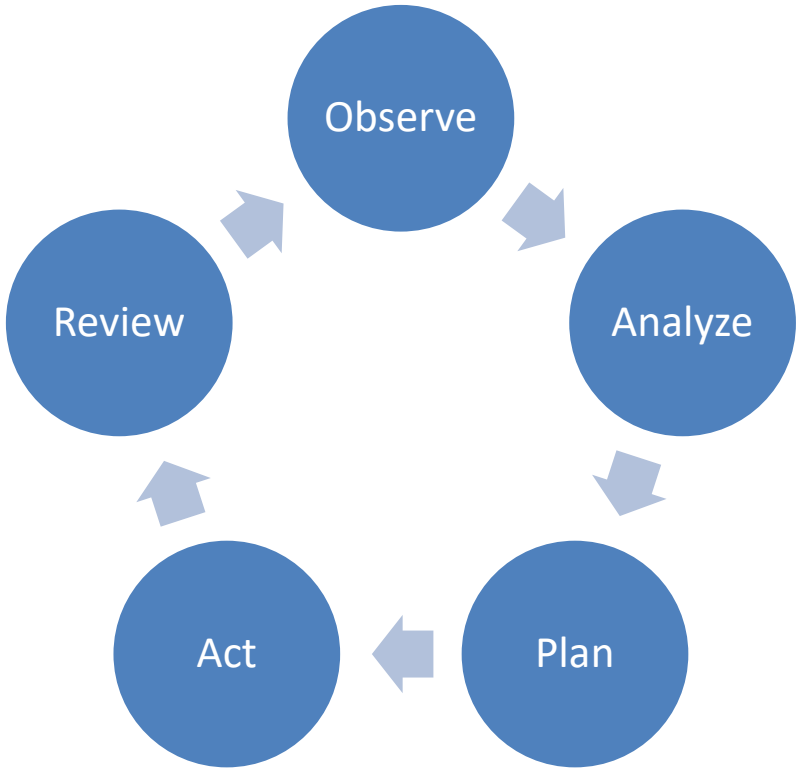
Key Features:

- Dependency management ensures that all required libraries and tools are included.
- Build processes compile and package the code for deployment.

Example:

- A Java application uses Maven for dependency management, with Jenkins automating the build process and packaging the app into a Docker container.

MONITORING AND FEEDBACK



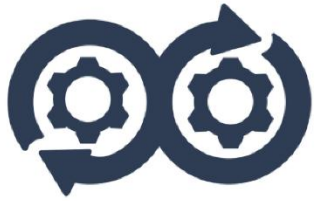
Monitoring systems provide real-time insights into application performance and health, while feedback loops ensure continuous improvement after deployment.

- **Monitoring Systems:** Use tools like Prometheus, Grafana, or New Relic to track performance metrics, detect anomalies, and alert on failures.
- **Feedback Loops:** Provide developers with insights into failed builds, test results, and application health to quickly address issues.
- **Post-Deployment Review:** Analyze metrics collected post-deployment to validate application performance and stability in production.

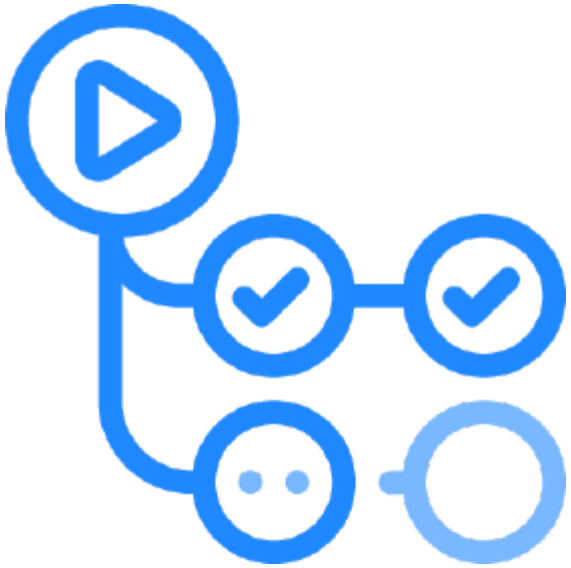
Example Workflow:

1. Deploy application using the pipeline.
2. Monitor metrics in Prometheus and Grafana to ensure the deployment meets performance benchmarks.
3. Address any detected issues or anomalies promptly.

GITHUB ACTIONS



WHAT IS GITHUB ACTIONS?



GitHub Actions is a powerful automation platform integrated directly into GitHub. It enables developers to automate workflows for building, testing, and deploying applications directly from their GitHub repositories. By using GitHub Actions, teams can streamline their CI/CD pipelines, ensuring faster and more reliable software delivery.

Core Concept: Automate tasks in response to events, such as code commits, pull requests, or manual triggers.

Key Use Cases:

- Automatically run tests and linting when code is pushed.
- Build and deploy applications to staging or production environments.
- Perform regular maintenance tasks, such as dependency updates or security scans.

Example: A developer commits code to a repository, triggering a workflow that tests the code, builds the application, and deploys it to a staging server.

KEY CONCEPTS OF GITHUB ACTIONS

Workflows: Defined automation processes written in YAML files. These describe the steps GitHub Actions should perform.

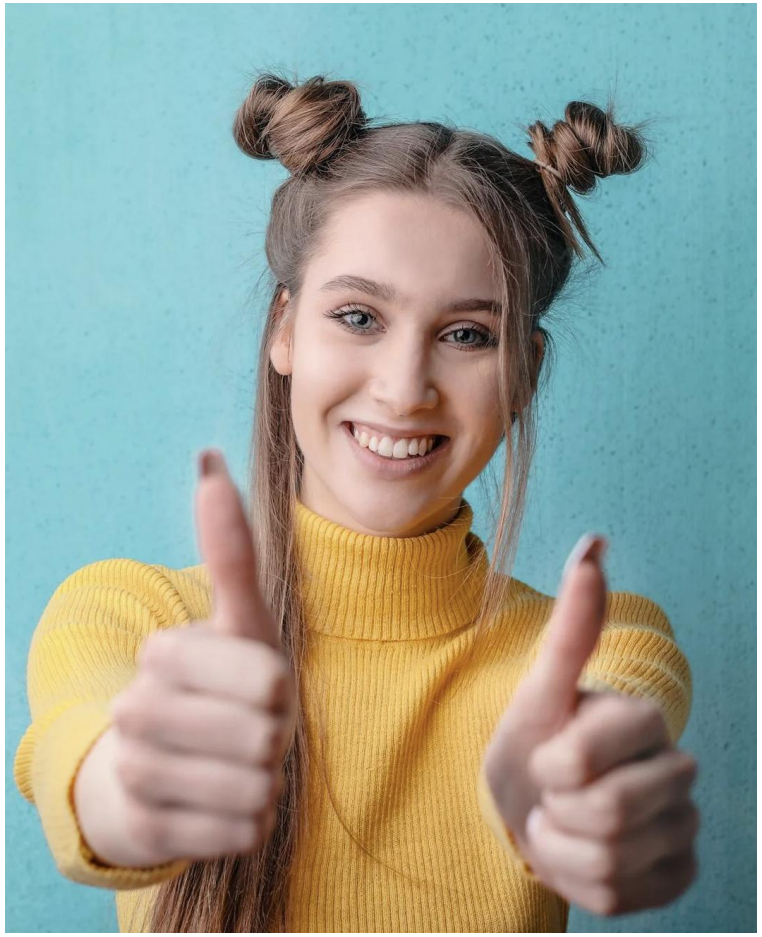
Triggers: Events in the repository (e.g., push, pull_request, schedule) that start a workflow.

Jobs: A collection of steps executed in sequence or parallel within a workflow.

Runners: Virtual machines or containers where the jobs are executed.

Example: A workflow triggered on a pull_request event runs linting and testing jobs.

BENEFITS OF GITHUB ACTIONS



Integration: Fully integrated with GitHub repositories, reducing the need for external CI/CD tools.

Scalability: Supports multiple runners, allowing workflows to execute across various operating systems and environments.

Customization: Offers pre-built actions from the GitHub Marketplace and supports creating custom actions for unique use cases.

Community Support: A large ecosystem of contributors provides reusable actions and templates.

For instance, developers can use actions from the marketplace to deploy code to AWS, Azure, or Google Cloud.

WORKFLOW CONFIGURATION WITH YAML

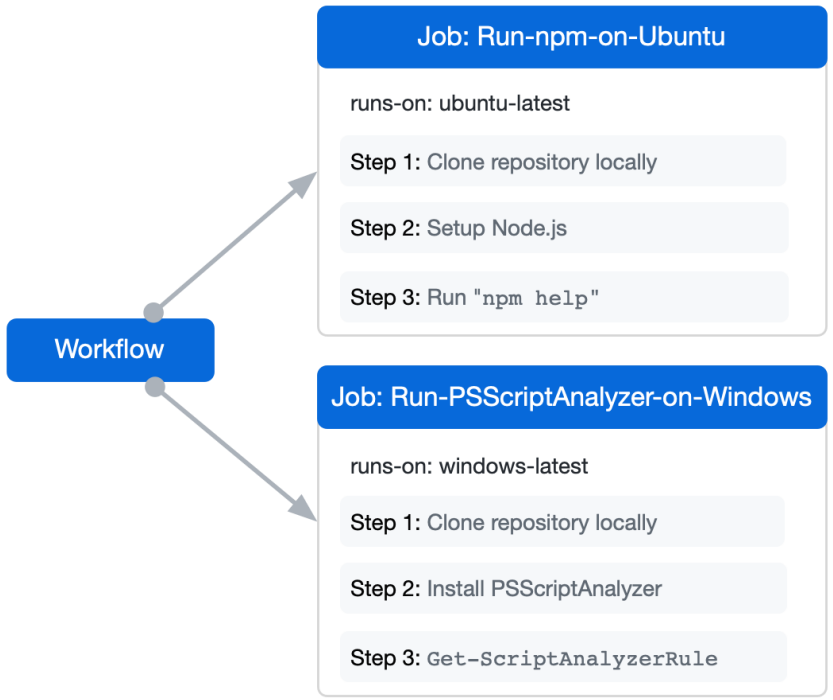
```yaml
name: Build and Test
on: push
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Run Tests
        run: npm test
```

GitHub Actions **workflows** are configured using simple YAML files stored in the repository under .github/workflows. These files define the sequence of tasks to automate.

Key Elements:

- **Name:** A descriptive name for the workflow (e.g., “Build and Test”).
- **Triggers:** Define the events that start the workflow (e.g., push, pull_request).
- **Jobs:** Specify tasks to perform, such as testing, building, or deploying.
- **Steps:** Individual actions within a job, such as running a script or installing dependencies.

EXTENSIVE SUPPORT FOR RUNNERS



GitHub Actions provides a variety of environments for running workflows:

Hosted Runners:

- Pre-configured environments maintained by GitHub.
- Support for Windows, Linux, and macOS.
- Scalable and easy to use for most CI/CD needs.

Self-Hosted Runners:

- Custom environments managed by users.
- Ideal for workflows requiring specific hardware, software, or network configurations.

Example: A team uses a self-hosted runner to execute workflows on an internal Kubernetes cluster.

GITHUB MARKETPLACE

```yaml
jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Install dependencies
        uses: actions/setup-node@v3
        with:
          node-version: 16

      - name: Deploy to AWS
        uses: aws-actions/configure-aws-credentials@v2
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
        run: aws s3 sync ./build s3://my-bucket
```

The **GitHub Marketplace** provides a centralized platform to explore and integrate pre-built actions into your workflows. These actions, created by GitHub or the community, automate complex tasks, allowing developers to focus on their core application logic.

Access Pre-Built Actions: Browse thousands of ready-to-use actions designed to handle tasks like testing, deployments, notifications, and more. The Marketplace offers a wide variety of actions suitable for various languages, platforms, and services.

Examples of Popular Actions:

- **Deployment Actions:** Actions for deploying to AWS, Azure, Google Cloud, or Kubernetes.
- **Testing Actions:** Run Selenium tests, execute unit tests, or validate infrastructure with Terratest.
- **Notification Actions:** Automate communication by sending messages to Slack, Teams, or email upon workflow completion.

GITHUB MARKETPLACE

yaml

```yaml
name: Build and Deploy Workflow

# Trigger the workflow on pushes to the main branch
on:
  push:
    branches:
      - main

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest  # Use the latest Ubuntu runner

    steps:
      # Step 1: Plugin to check out the repository code
      # This action pulls the code from the repository into the workflow runner.
      - name: Checkout repository
        uses: actions/checkout@v3  # Plugin from GitHub Marketplace
```

The **GitHub Marketplace** provides a centralized platform to explore and integrate pre-built actions into your workflows. These actions, created by GitHub or the community, automate complex tasks, allowing developers to focus on their core application logic.

Access Pre-Built Actions: Browse thousands of ready-to-use actions designed to handle tasks like testing, deployments, notifications, and more. The Marketplace offers a wide variety of actions suitable for various languages, platforms, and services.

Examples of Popular Actions:

- **Deployment Actions:** Actions for deploying to AWS, Azure, Google Cloud, or Kubernetes.
- **Testing Actions:** Run Selenium tests, execute unit tests, or validate infrastructure with Terratest.
- **Notification Actions:** Automate communication by sending messages to Slack, Teams, or email upon workflow completion.

GITHUB ACTIONS PLUGINS OVERVIEW

GitHub Actions plugins are reusable components that extend the functionality of workflows by integrating external tools and services. These plugins simplify the setup of tasks and make workflows modular and efficient.

A plugin is any action defined in a GitHub repository or available in the GitHub Marketplace that provides specific functionality, such as setting up environments or interacting with cloud services.

Examples of Plugins:

- **Setup plugins:** Actions that configure environments (e.g., actions/setup-node to configure Node.js).
- **Deployment plugins:** Tools to manage deployments (e.g., actions/deploy-pages for GitHub Pages).
- **Utility plugins:** General tools like actions/checkout for cloning repositories into workflows.

HOW PLUGINS WORK

yaml

```yaml
name: CI/CD Workflow

on:
  push:
    branches:
      - main

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest

    steps:
      # Step 1: Plugin to check out the repository code
      - name: Checkout code
        uses: actions/checkout@v3  # Plugin from GitHub Marketplace

      # Step 2: Plugin to set up Node.js environment
      - name: Setup Node.js
        uses: actions/setup-node@v3  # Plugin from GitHub Marketplace
        with:
          node-version: 16
```

GitHub Actions plugins function as essential building blocks in workflows, automating tasks and ensuring consistency. They allow developers to focus on core logic by handling repetitive processes like setup, deployment, and testing. The key components of how plugins work include:

- **Integration:** Plugins are added in YAML workflows, performing tasks like environment setup or deploying applications seamlessly.
- **Input Parameters:** Plugins allow configurable inputs, such as specifying versions or paths, to tailor them to workflow requirements.
- **Secrets Management:** Sensitive data like API keys or tokens is securely stored and accessed through GitHub's built-in secrets.
- **Output Data:** Plugins generate outputs that subsequent steps in the workflow can use, enabling flexible and dynamic processes.

BEST PRACTICE FOR USING PLUGINS

To make the most of GitHub Actions plugins, it is important to follow best practices that ensure workflows remain secure, efficient, and maintainable. These practices include:

- **Verified Sources:** Choose plugins from GitHub Marketplace or reputable repositories to avoid security vulnerabilities.
- **Specify Plugin Versions:** Pin plugins to fixed versions (e.g., @v3) to ensure workflow stability and prevent unexpected updates.
- **Optimize Plugin Usage:** Limit the number of plugins to essential tasks, selecting lightweight options to improve performance.
- **Regular Updates:** Update plugins periodically to benefit from new features, bug fixes, and security patches.

WHAT ARE BUILD PIPELINES

Build pipelines are automated workflows that manage the steps required to build, test, and prepare software for deployment. They provide a structured way to ensure that code changes are validated and production-ready at every stage of the development lifecycle.

A build pipeline automates activities like compiling code, resolving dependencies, and generating deployable artifacts, ensuring repeatability and consistency.

- **Stages of a Pipeline:** Typical stages include build (source code compilation), test (validation of code quality), and deploy (packaging and preparing for release).
- **Benefits:** Build pipelines reduce manual errors, accelerate feedback loops for developers, and ensure the software adheres to quality standards before deployment.

KEY COMPONENTS OF A BUILD PIPELINE

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Build application
        run: make build

      - name: Run tests
        run: make test
```

Build pipelines rely on several critical components that work together to automate software delivery effectively. These components are designed to handle various stages of development and ensure the pipeline is efficient and reliable.

- **Source Control Integration:** Pipelines monitor repositories for code changes and trigger workflows automatically upon commits or pull requests.
- **Build Stage:** This stage compiles the source code, resolves dependencies, and generates binaries or other deployable artifacts.
- **Testing Stage:** Automated tests—such as unit, integration, or end-to-end tests—are run to ensure the code behaves as expected and remains stable.
- **Artifact Generation:** Pipelines create deployable outputs, such as container images or packaged files, which are ready for deployment.

BEST PRACTICES FOR BUILD PIPELINES

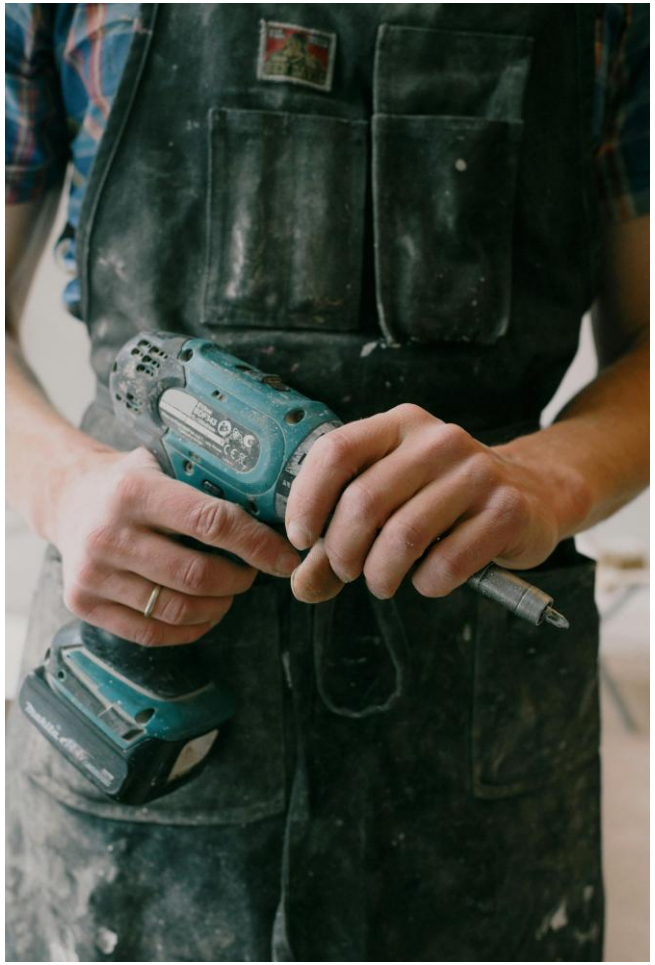
yaml

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Build application
        run: make build

  # Parallel test jobs to run tests
  # on multiple Node.js versions
  test:
    runs-on: ubuntu-latest
    needs: build  # Ensures the 'test' job runs
                  # only after the 'build' job
    strategy:
      matrix:
        node: [14, 16, 18]  # Creates a matrix to run
                            # tests on Node.js versions
                            # 14, 16, and 18 concurrently
    steps:
      - name: Run tests on Node.js ${{ matrix.node }}
        run: npm test
```

To make build pipelines efficient, maintainable, and reliable, it's essential to follow industry best practices. These practices optimize workflows, reduce errors, and ensure smooth operation across stages:

- **Use Modular Stages:** Divide pipelines into separate stages (build, test, deploy) to isolate issues and make debugging easier.
- **Fail Fast:** Configure pipelines to stop execution immediately upon encountering an error to save time and resources.
- **Parallel Execution:** Optimize performance by running independent tasks, such as testing across environments, in parallel.
- **Monitor and Optimize:** Use logs and analytics to identify bottlenecks, optimize performance, and ensure pipelines remain efficient.
- **Pipeline as Code:** Store pipeline configurations as code in the repository, enabling version control and collaboration.

WHAT ARE BUILD TRIGGERS?



Build triggers are events or conditions that automatically initiate the execution of a pipeline. They eliminate the need for manual intervention and ensure that workflows are executed in response to specific actions or schedules, enabling seamless automation.

Definition: Build triggers define when and how a pipeline starts, based on events like code changes, pull requests, or timed schedules.

Types of Triggers: Triggers can be event-driven (e.g., on a commit or pull request) or time-based (e.g., scheduled builds).

Benefits: Build triggers ensure builds are consistent, reduce manual steps, and allow teams to catch issues early by automating workflow execution.

COMMON BUILD TRIGGER TYPES

yaml

```yaml
name: Build and Test Pipeline

# Define triggers for the workflow
on:
  # Push events: Trigger when changes are pushed to the main branch or tags
  push:
    branches:
      - main
    tags:
      - v*

  # Pull request events: Trigger when a PR is opened or updated, targeting the main branch
  pull_request:
    branches:
      - main

  # Scheduled triggers: Run every day at 2 AM using cron syntax
  schedule:
    - cron: '0 2 * * *'

  # Manual trigger: Allow users to manually start the workflow through the Actions UI
  workflow_dispatch:
    inputs:
      environment:
        description: 'Target environment (e.g., staging, production)'
        required: true
        default: staging
```

Triggers in GitHub Actions are highly customizable and can adapt to various workflow needs. Key types include:

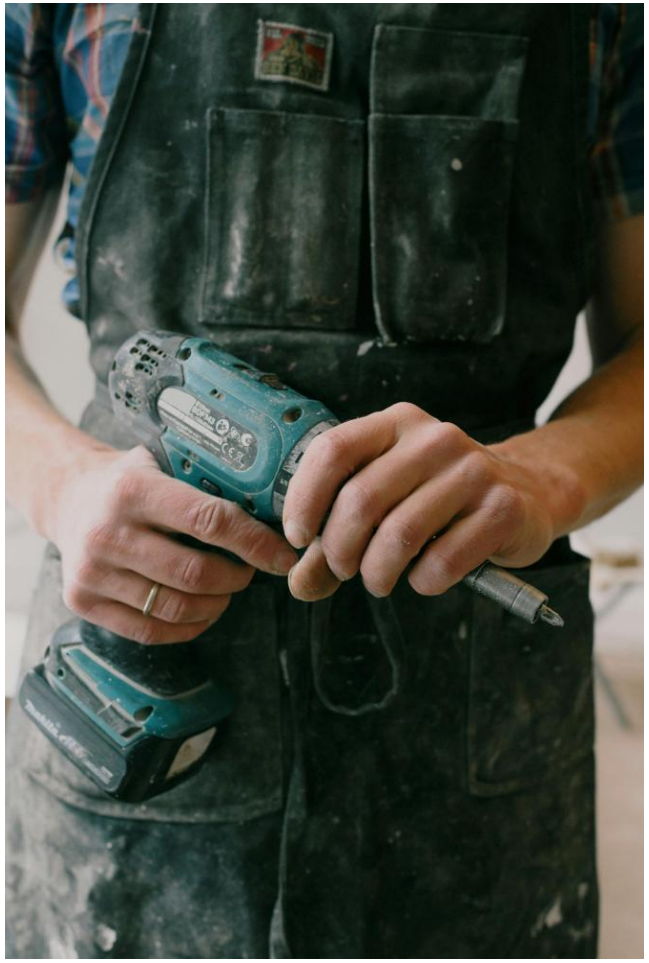
Push Events: Automatically trigger workflows when changes are pushed to a branch or a tag. Useful for continuous integration on the main branch.

Pull Requests: Start workflows when a pull request is opened or updated. Helps validate code changes before merging.

Scheduled Triggers: Use cron syntax to schedule builds at specific intervals, such as nightly tests or weekly deployments.

Manual Triggers: Enable workflows to be triggered manually through the GitHub Actions UI or via API calls for ad-hoc needs.

BEST PRACTICES FOR BUILD TRIGGERS



Configuring build triggers thoughtfully ensures workflows run efficiently and only when necessary. Best practices include:

Scope Triggers to Relevant Changes: Limit triggers to specific branches (e.g., main) or tags to avoid running workflows on irrelevant updates.

Use Filters: Apply paths or paths-ignore to trigger workflows only when specific files or directories are modified.

Combine Triggers: Use a combination of triggers, such as push and pull_request, to address different scenarios like direct commits and code reviews.

Avoid Over-Triggering: Ensure workflows aren’t triggered excessively, which could waste resources or cause unnecessary delays.

WHAT ARE ENVIRONMENT VARIABLES?

yaml

```yaml
env:
  NODE_ENV: production

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Print variable
        run: echo $NODE_ENV
```

Environment variables are dynamic values that store configuration data, secrets, or other information required by workflows during execution. They allow developers to customize workflows without hardcoding values directly into the pipeline.
Environment variables are key-value pairs accessible to workflows and steps, used for storing reusable configuration like API keys, URLs, or environment-specific settings.

Scope of Variables: Variables can be scoped globally (workflow-wide), per job, or per step. This flexibility ensures that sensitive or environment-specific data is kept secure and accessible where needed.

Benefits: Environment variables simplify workflows, improve reusability, and make pipelines easier to maintain by centralizing configuration data.

USING ENVIRONMENT VARIABLES IN WORKFLOWS

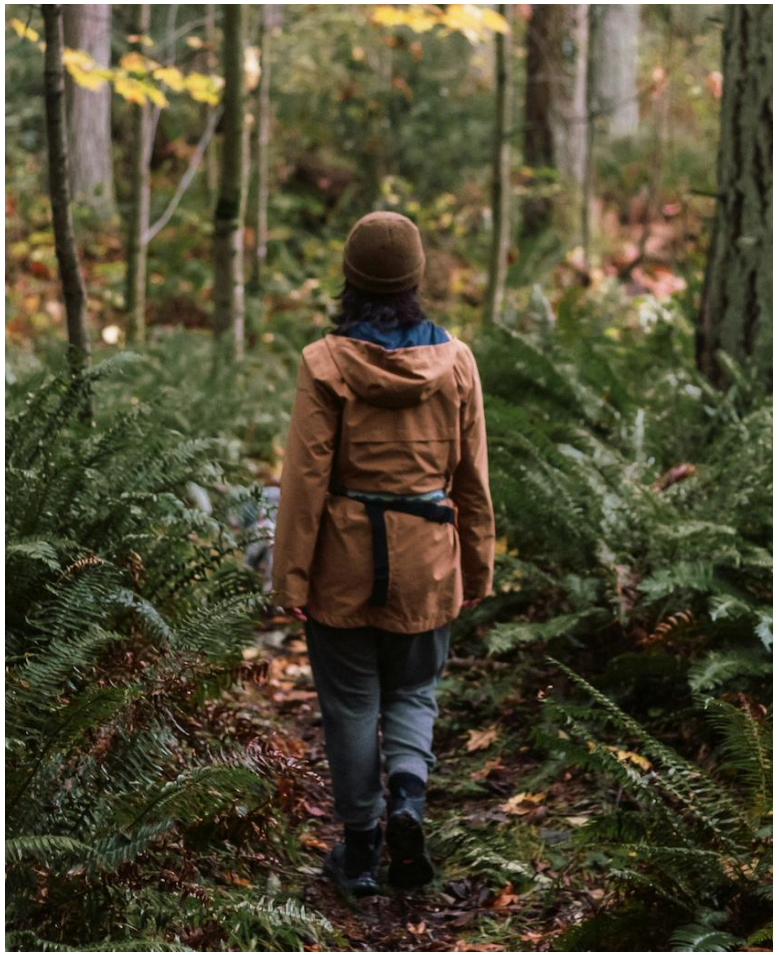
yaml

```yaml
name: Variable Scope Example

on:
  push:
    branches:
      - main

env:  # Workflow-level variables
  WORKFLOW_VAR: "This is accessible to all jobs and steps"

jobs:
  build:
    runs-on: ubuntu-latest

    env:  # Job-level variables
      JOB_VAR: "This is accessible only within the build job"

    steps:
      - name: Print workflow variable
        run: echo "Workflow variable: $WORKFLOW_VAR"

      - name: Print job variable
        run: echo "Job variable: $JOB_VAR"

      - name: Define and print step variable
        run: |
          STEP_VAR="This is only accessible within this step"
          echo "Step variable: $STEP_VAR"
```

Environment variables in GitHub Actions can be set and accessed at various levels, providing flexibility in managing configuration data. Env vars can apply at many different levels:

- **Workflow Level:** Variables defined at this level are accessible throughout the entire workflow.
- **Job Level:** Variables can be scoped to a specific job, ensuring other jobs cannot access them.
- **Step Level:** For maximum isolation, variables can be defined for individual steps.

BEST PRACTICES FOR ENVIRONMENT VARIABLES



Managing environment variables effectively ensures security, maintainability, and clarity in workflows. Key practices include:

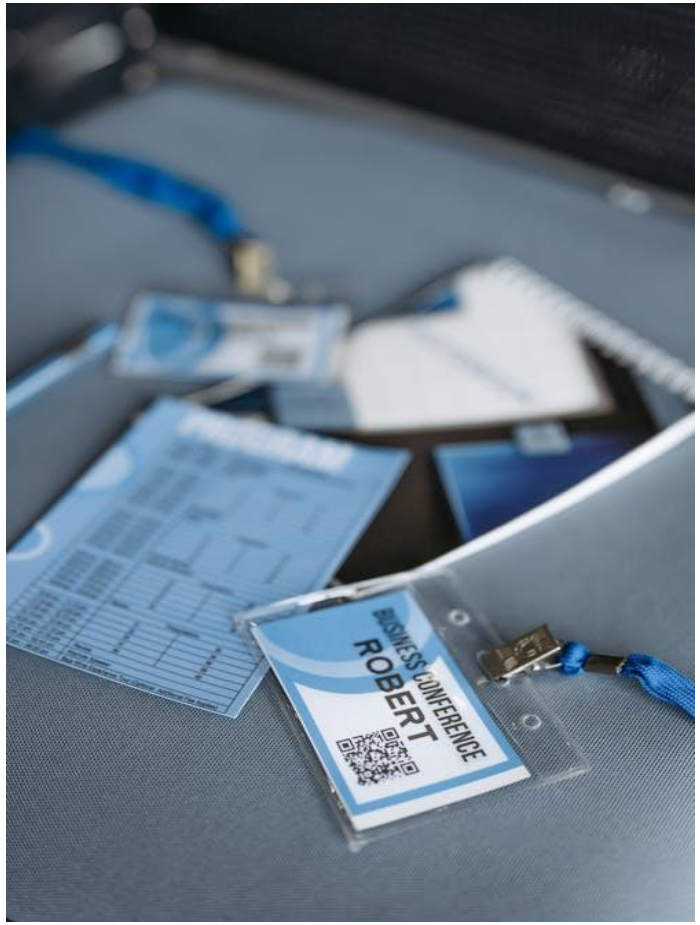
Limit Scope: Define variables only at the level where they are needed (workflow, job, or step) to minimize unintended access.

Avoid Hardcoding: Use variables for values that may change across environments (e.g., development, staging, production) to simplify updates.

Secure Secrets: Store sensitive information like API keys and credentials in GitHub Secrets, which encrypts the data.

Document Variables: Clearly document the purpose and scope of each variable in the repository's README or pipeline documentation.

GITHUB ACTIONS CREDENTIALS



Credentials are sensitive data such as authentication tokens, API keys, or certificates that workflows use to access external services securely. Proper management of credentials is critical for maintaining the security of your workflows and systems.

Credentials refer to any sensitive information required by workflows to interact with external services or perform secure operations (e.g., accessing cloud platforms or private repositories).

- **Storage:** GitHub Secrets is the primary method for securely storing credentials, encrypting the data and restricting access.
- **Usage:** Credentials enable workflows to authenticate securely without exposing sensitive data directly in the pipeline.

MANAGING CREDENTIALS

yaml

```yaml
name: Secure Credentials Example

on:
  push:
    branches:
      - main

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
      # Step 1: Checkout the repository code
      - name: Checkout code
        uses: actions/checkout@v3

      # Step 2: Configure AWS Credentials securely using GitHub Secrets
      - name: Configure AWS Credentials
        env:
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}  # Injected secret
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}  # Injected secret
        run: echo "AWS credentials configured"

      # Step 3: Deploy application to AWS using injected credentials
      - name: Deploy to AWS
        env:
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
        run: |
          aws s3 sync ./build s3://my-bucket
```

Credentials are securely managed through GitHub Secrets and are injected into workflows when needed. Key considerations for managing credentials include:

GitHub Secrets: Use the repository's Secrets feature to store credentials securely. Secrets are accessible only to authorized workflows.

Accessing Secrets: Secrets are injected as environment variables and accessed using ${{ secrets.SECRET_NAME }} in workflows.

Scoped Access: Secrets can be scoped at the repository, organization, or environment level to limit access to only the workflows that need them.

BEST PRACTICES FOR CREDENTIALS

Securely managing credentials ensures the safety of your workflows and external services. Follow these practices to maintain security:

Use GitHub Secrets: Always store sensitive information like API keys or tokens in GitHub Secrets instead of hardcoding them in workflows.

Scope Secrets Carefully: Restrict access to secrets based on the principle of least privilege. For example, use environment-level secrets for production credentials.

Rotate Secrets Regularly: Update and rotate secrets periodically to reduce the risk of compromise.

Audit Secret Usage: Regularly review workflows to ensure secrets are being used securely and that no unnecessary secrets are defined.

PARAMETERIZATION OVERVIEW

Parameterization allows workflows and pipelines to use dynamic inputs instead of hardcoding values. By leveraging parameters, pipelines can adapt to different environments, configurations, or inputs, improving flexibility and maintainability.

Parameterization introduces variables that are set dynamically, enabling workflows to behave differently based on the inputs provided.

- **Use Cases:** Deploying to multiple environments (e.g., dev, staging, prod), running tests for different configurations, or customizing resource provisioning.
- **Benefits:** Improves reusability, reduces redundancy, and simplifies updates by centralizing configurable values.

DEFINING AND USING PARAMETERS

```yaml
name: CI/CD Workflow with Parameters

on:
  # Trigger the workflow manually with inputs
  workflow_dispatch:
    inputs:
      environment:
        description: 'Target environment (staging, production)'  # Workflow Input
        required: true
        default: staging

jobs:
  build:
    runs-on: ubuntu-latest

    # Define environment variables for the entire job
    env:
      API_URL: https://api.example.com  # Environment Variable
      DEPLOY_REGION: us-east-1

    steps:
      # Step 1: Checkout repository
      - name: Checkout code
        uses: actions/checkout@v3

      # Step 2: Print workflow input and environment variables
      - name: Print parameters
        run: |
          echo "Target environment: ${{ inputs.environment }}"
          echo "API URL: $API_URL"
          echo "Deploy region: $DEPLOY_REGION"

  test:
    runs-on: ubuntu-latest

    # Use a matrix to run tests across multiple configurations
    strategy:
      matrix:
        node-version: [14, 16, 18]  # Matrix Parameter for Node.js versions
        os: [ubuntu-latest, windows-latest]  # Matrix Parameter for operating systems

    steps:
      # Step 1: Setup Node.js with the current matrix version
      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: ${{ matrix.node-version }}
```

GitHub Actions workflows allow for dynamic behavior by defining and consuming parameters. These parameters control workflow execution and can be defined at different levels or passed from external sources.

Workflow Inputs: Inputs are set at the workflow level, ideal for manual triggers or reusable workflows. For example, users can specify target environments like staging or production when triggering a workflow.

Environment Variables: These store configuration values shared across multiple steps or jobs, such as API URLs or deployment regions, enabling centralized and reusable settings.

Matrix Parameters: Matrices enable running jobs in parallel with varying configurations, such as testing across Node.js versions or operating systems, ensuring broader coverage and efficiency.

BEST PRACTICES FOR PARAMETERIZATION

Effectively parameterizing workflows ensures better reusability, security, and maintainability. Key practices include:

Validate Inputs: Use conditional logic to validate parameter values and prevent invalid configurations.

Keep Parameters Relevant: Only define parameters necessary for the workflow's flexibility and avoid overcomplicating pipelines.

Combine with Secrets: Parameterize sensitive data with GitHub Secrets for security while still enabling flexibility.

Document Parameters: Clearly document what each parameter does and its acceptable values to aid understanding and maintainability.

END-TO-END INFRASTRUCTURE DEPLOYMENT PIPELINE

In this walkthrough, we will build a complete end-to-end infrastructure deployment pipeline using GitHub Actions and Terraform. This pipeline will automate the provisioning of infrastructure and deployment of resources in a secure and reusable manner.

What You Will Learn:

- How to set up a GitHub Actions pipeline for infrastructure deployment.
- Incorporating Terraform to automate provisioning tasks.
- Using parameters, environment variables, and credentials for flexibility and security.
- Testing, validating, and monitoring the pipeline to ensure reliability.

Key Features of the Pipeline:

- Triggered on changes to the repository.
- Fully parameterized for multi-environment support (e.g., staging, production).
- Securely manages sensitive credentials with GitHub Secrets.

CONFIGURE TERRAFORM STEPS

yaml

```yaml
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v2
        with:
          terraform_version: 1.4.0

      - name: Initialize Terraform
        run: terraform init

      - name: Plan Terraform
        run: terraform plan

      - name: Apply Terraform
        run: terraform apply -auto-approve
```

The pipeline should automate Terraform workflows by including three critical steps: initialize, plan, and apply. These steps streamline infrastructure provisioning and ensure consistency.

Initialize: Runs terraform init to set up the working directory, download provider plugins, and configure the backend for state management.

Plan: Executes terraform plan to preview infrastructure changes, showing additions, deletions, or modifications before applying them.

Apply: Runs terraform apply to provision or update resources like virtual machines, storage, or networking, ensuring the infrastructure matches the desired state.

SETTING UP THE PIPELINE

```yaml
on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main
```

Creating an end-to-end deployment pipeline starts with setting up the repository and defining the workflow. These foundational steps ensure a structured and reusable approach to infrastructure deployment.

Repository Configuration:

- **Create GitHub Repository:** Set up a repository to store all relevant Terraform configuration files (e.g., main.tf, variables.tf) and the workflow YAML file (.github/workflows/deploy.yml). This ensures that infrastructure code is version-controlled and accessible.

Workflow Triggers:

- **Define Pipeline Triggers:** Configure events to initiate the pipeline. For example, use a push event to automatically start the pipeline when changes are pushed to the main branch.
- **Scope Triggered Events:** Limit triggers to specific branches or tags (e.g., only the main branch) to prevent accidental or irrelevant workflow executions.

ENVIRONMENT SPECIFIC CONFIGURATION

```yaml
inputs:
  environment:
    description: 'Target environment'
    required: true

env:
  AWS_REGION: us-east-1

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Initialize Terraform
        run: terraform init -backend-config="env/${{ inputs.environment }}.tfstate"
```

To make the pipeline reusable and adaptable, incorporate parameters and environment-specific settings. These additions ensure workflows are flexible, handle different environments, and simplify updates without duplicating logic.

Parameterize Target Environments: Use GitHub Actions inputs or env to dynamically specify environments like staging or production. This approach allows the pipeline to adjust its behavior based on the chosen environment, such as stricter validation for production or experimental features for staging.

Environment Variables: Define variables for Terraform settings, such as AWS regions or instance types, to centralize configurations. By storing values like AWS_REGION or INSTANCE_TYPE in environment variables, the pipeline remains consistent across environments, reducing the need for code changes.

MANAGING CREDENTIALS

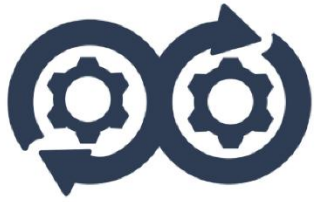
```yaml
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v2
        with:
          terraform_version: 1.4.0

      - name: Apply Infrastructure
        env:
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
        run: |
          terraform apply -auto-approve
```

To securely manage credentials in a deployment pipeline, incorporate GitHub Secrets and configure workflows to access them safely. This ensures sensitive data like API keys and cloud provider credentials are protected throughout the deployment process.

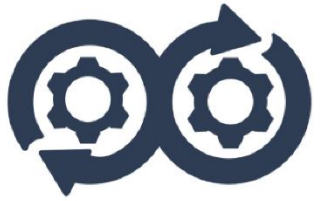
Store Secrets: Add credentials such as AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY in the repository's Secrets section to encrypt and restrict access.

Inject Secrets into Workflows: Use secrets within the workflow by mapping them to environment variables, allowing secure access during Terraform execution.

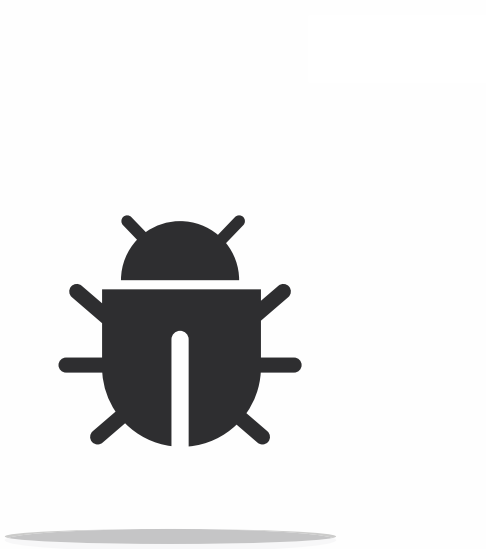
Lab: GitHub Actions



Testing



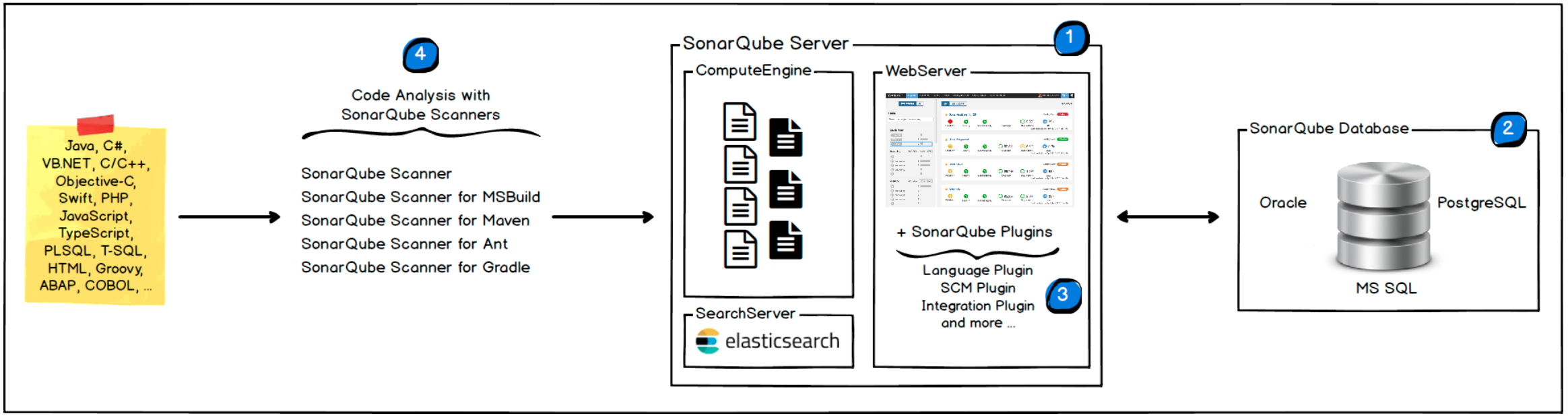
Testing



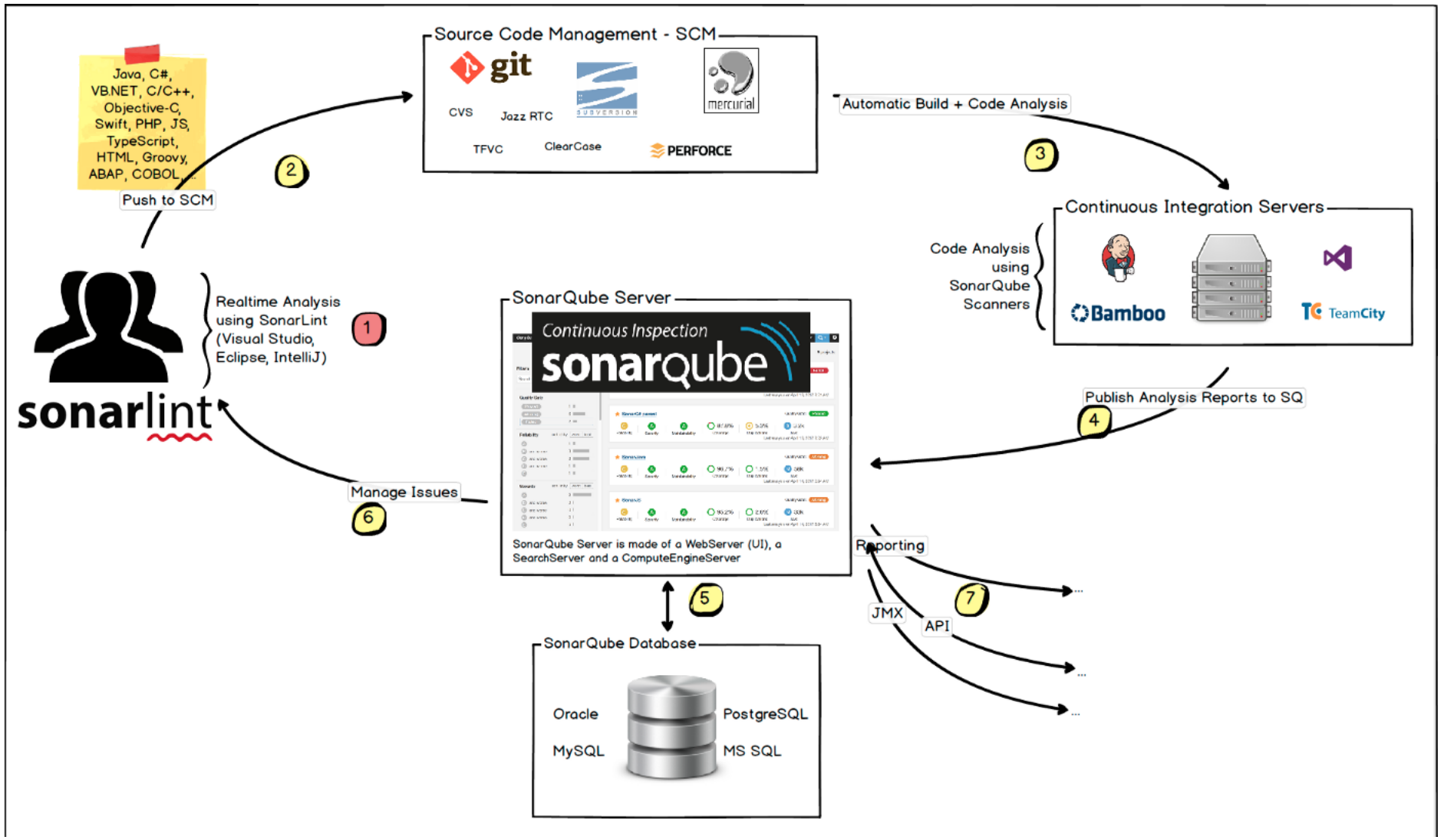
- Static code analysis
- Track project code quality
- Predict potential issues
- Objective feedback rather than comments between coworkers

Testing tools

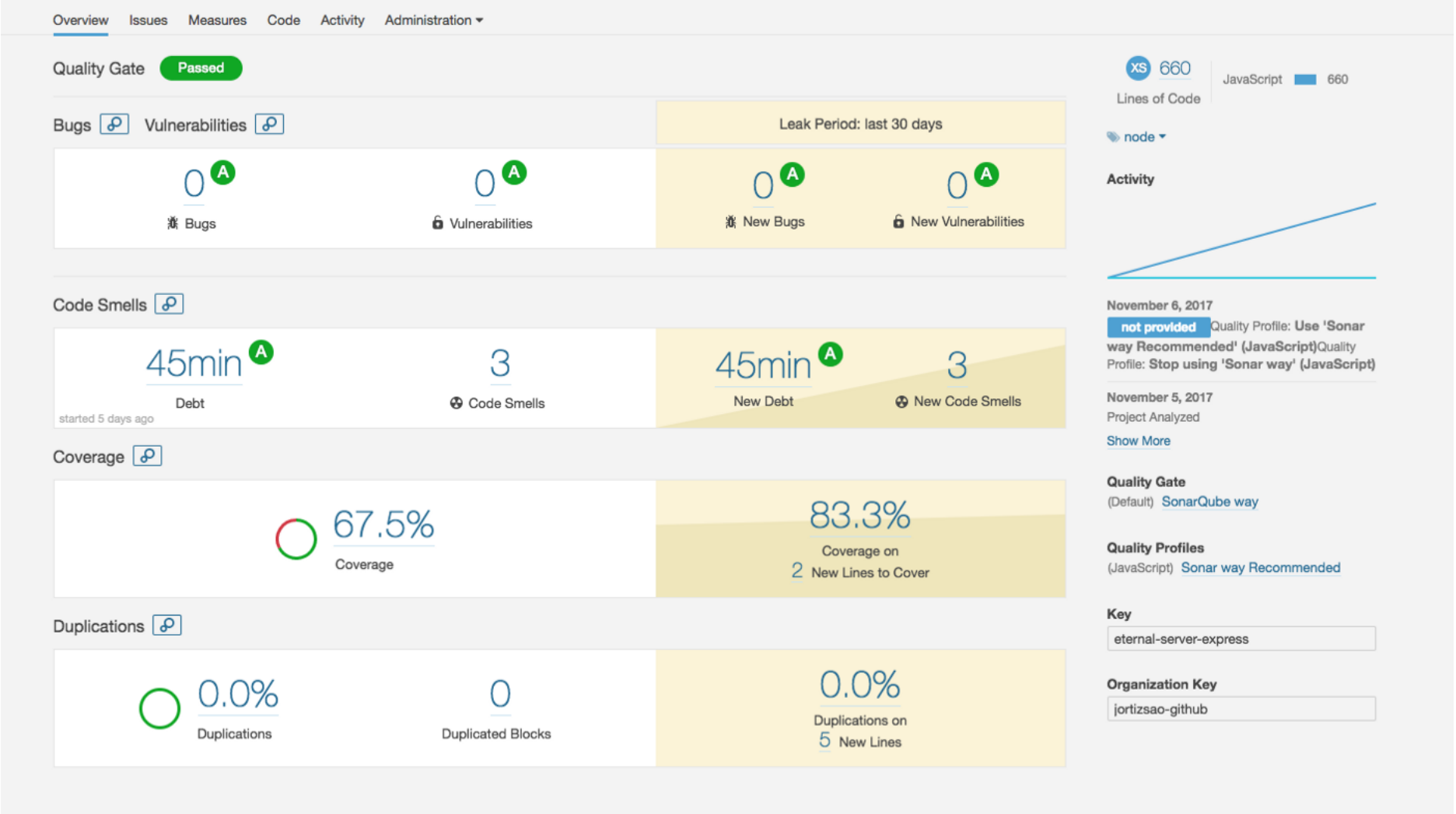
SonarQube



SonarQube



SonarQube



SonarQube

Pros:

- Powerful
- Customizable

Cons:

- Requires an external database
- Installed and runs outside of Jenkins

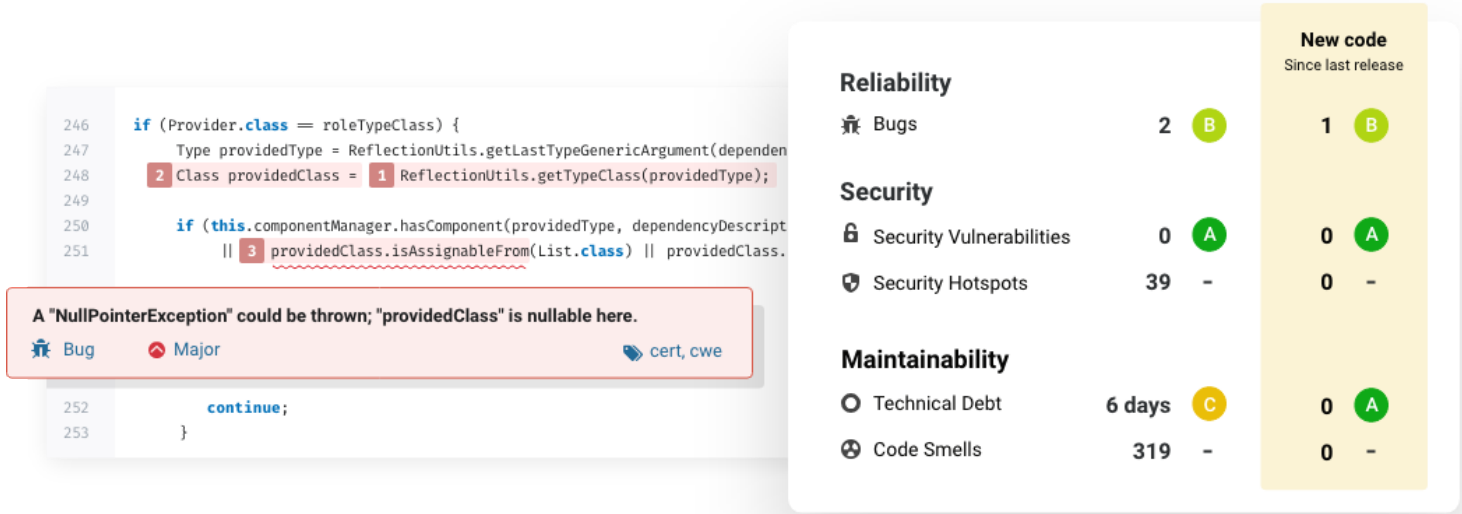
SonarQube concepts

Quality definitions:

- Bug
  - An issue that represents something wrong in the code. If this has not broken yet, it will, and probably at the worst possible moment. This needs to be fixed. Yesterday.

- Code smell:
  - A maintainability-related issue in the code. Leaving it as-is means that at best maintainers will have a harder time than they should making changes to the code. At worst, they'll be so confused by the state of the code that they'll introduce additional errors as they make changes.

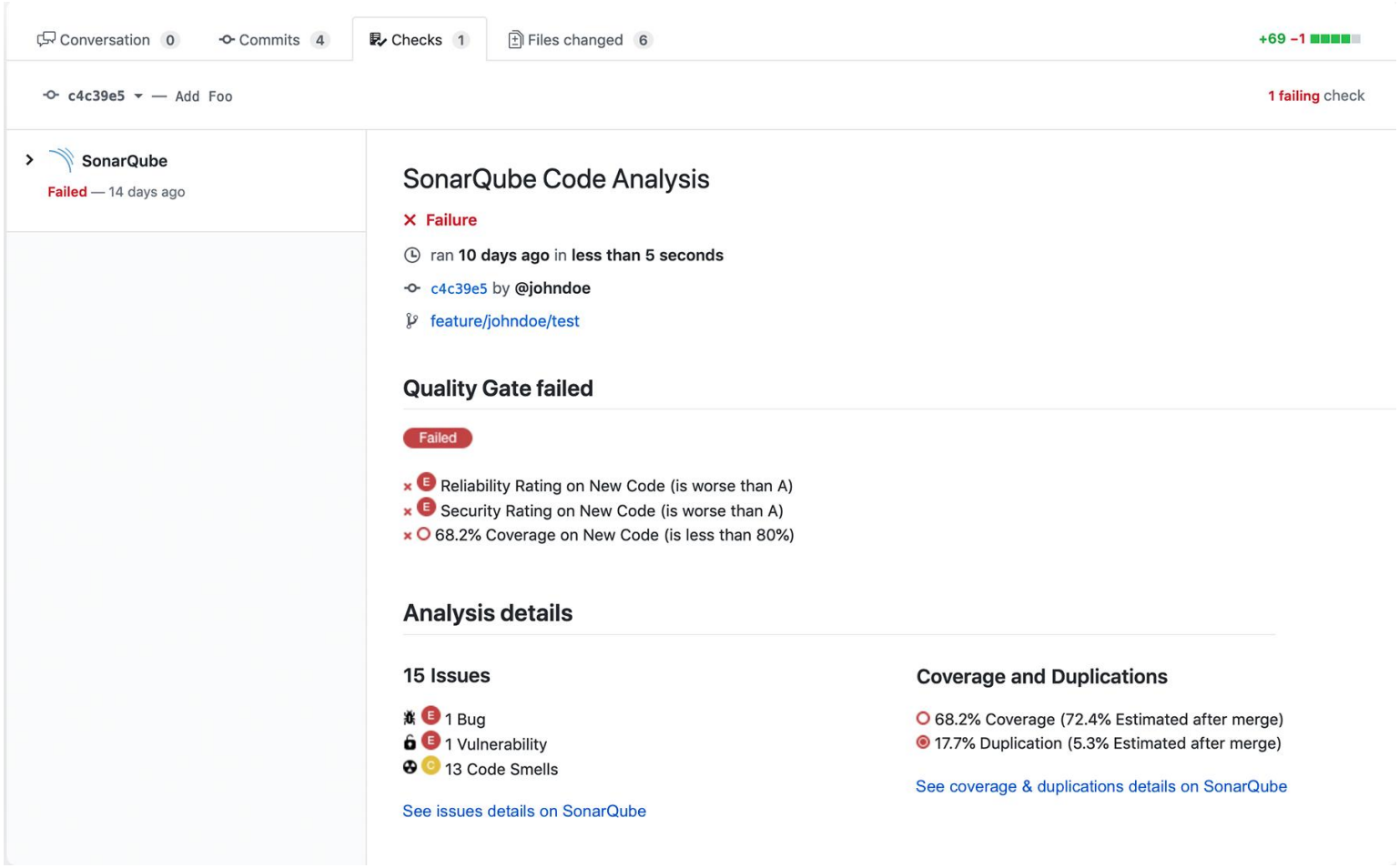
SonarQube



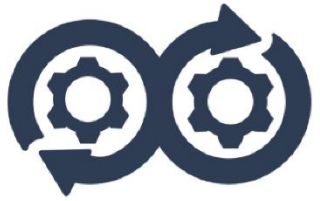
- Supports 27 languages:
  - Java
  - C
  - Python
  - Go
  - JavaScript
  - more!

SonarQube

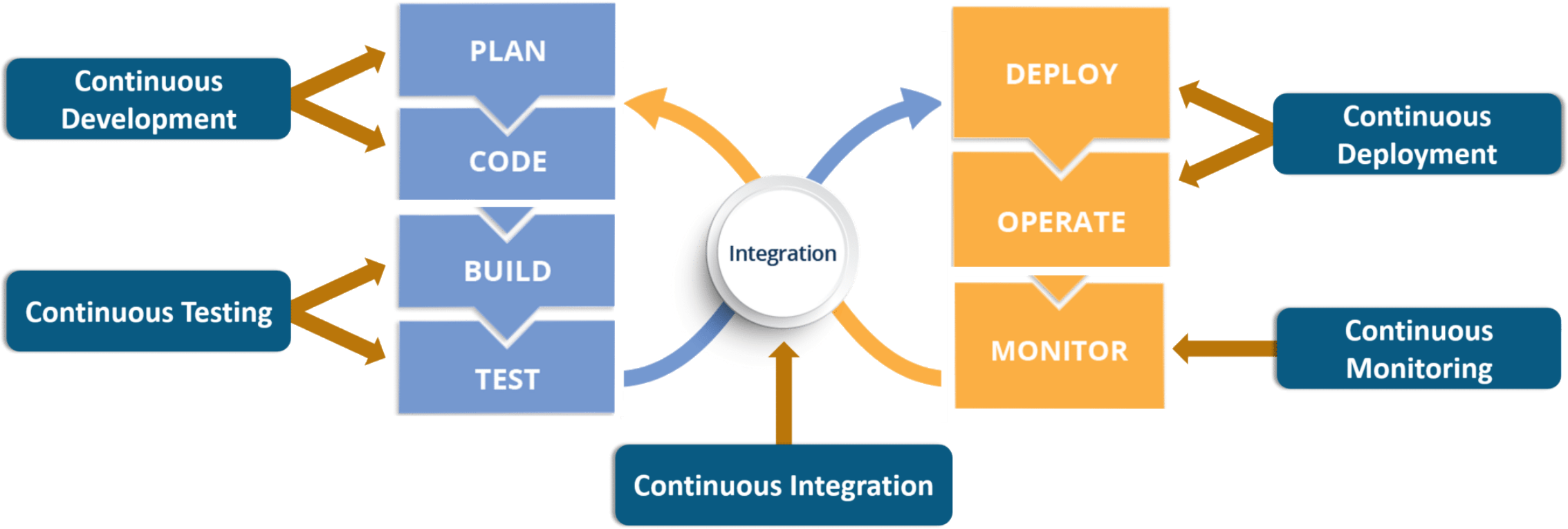
- Works with most version control providers
  - GitHub
  - Bitbucket
  - Azure DevOps
  - GitLab



Monitoring



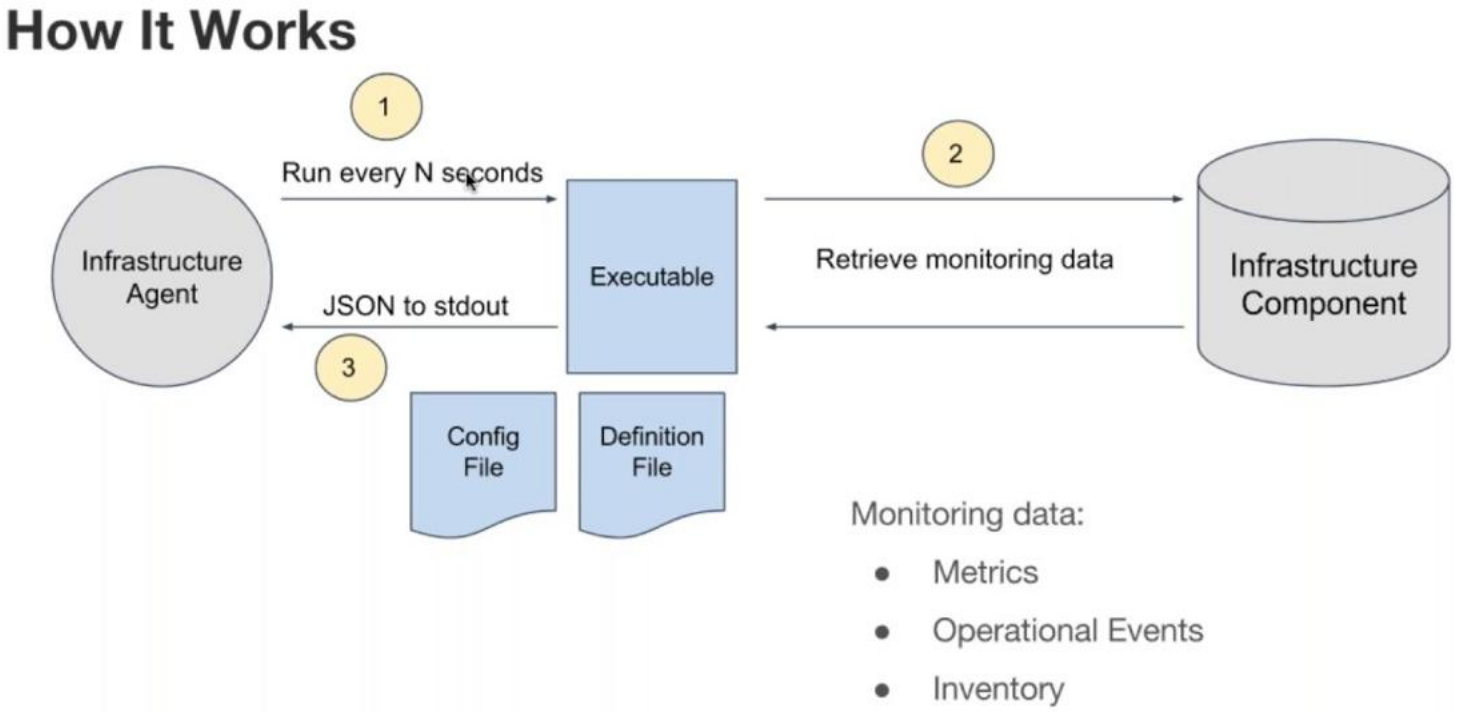
Continuous Monitoring



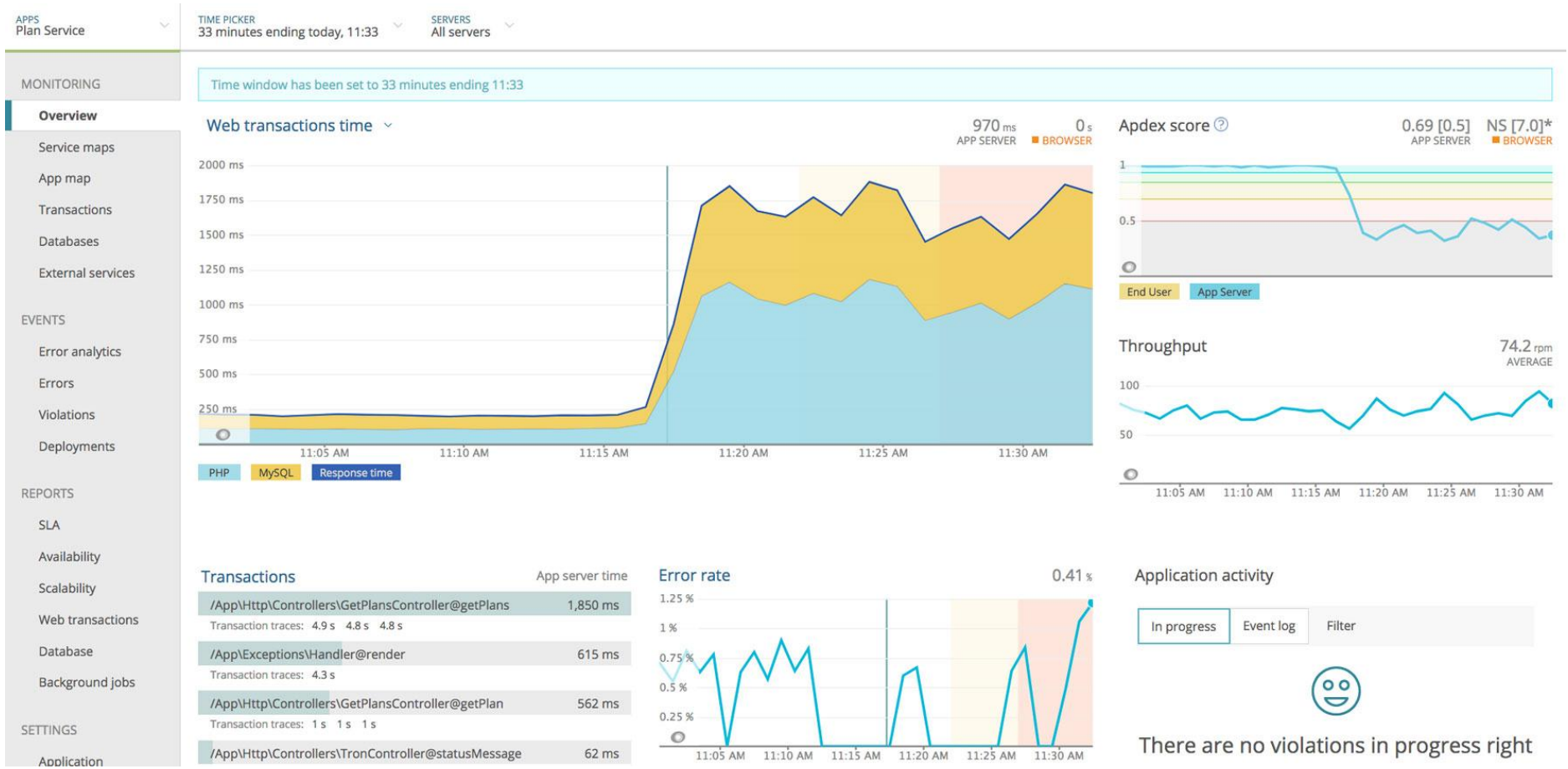
Application Performance Monitoring

- Businesses with huge and complex digital footprints and lots of websites and applications to manage can greatly benefit from application performance monitoring.
- Dashboard
- Agent

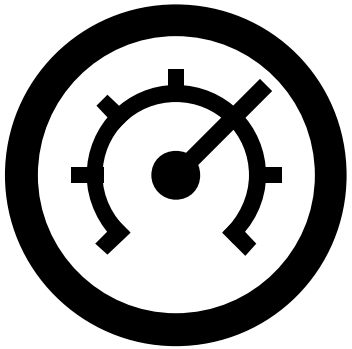
Architecture



Dashboard



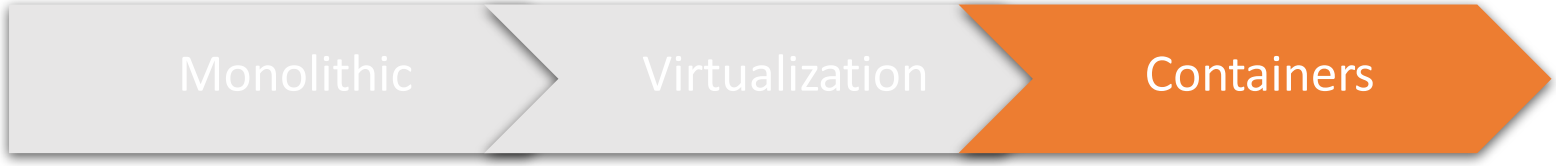
Application Monitoring

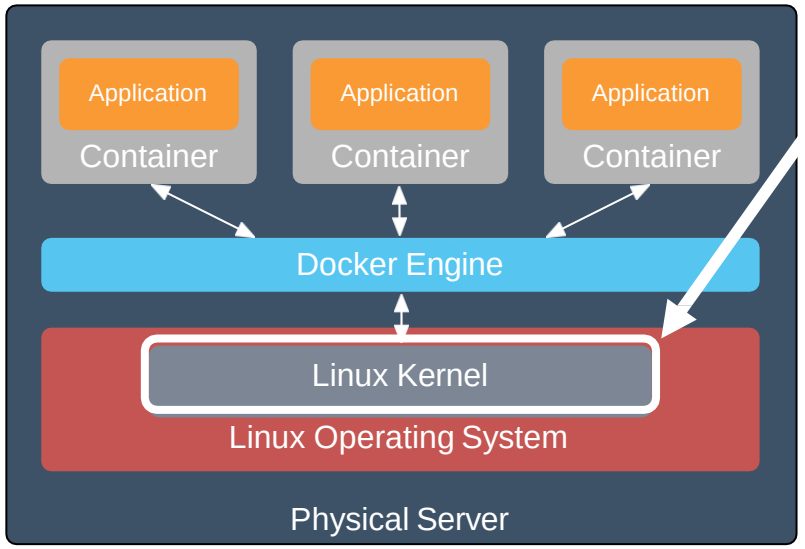


- Agent installation required
- Code snippet injected into application

```
## Required for New Relic Go Application Monitoring
app, err := newrelic.NewApplication(
    newrelic.ConfigAppName("Your Application Name"),
    newrelic.ConfigLicense("__YOUR_NEW_RELIC_LICENSE_KEY__"),
)
```

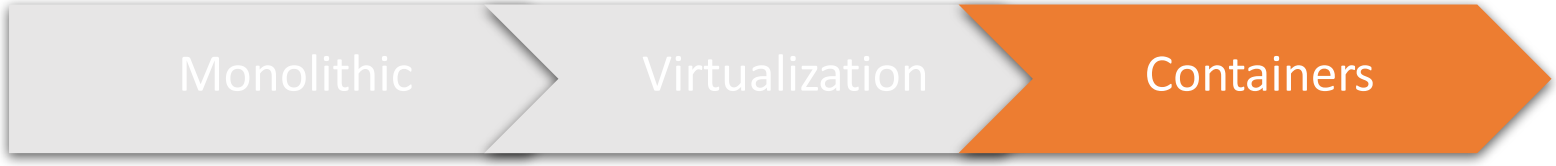
Containers

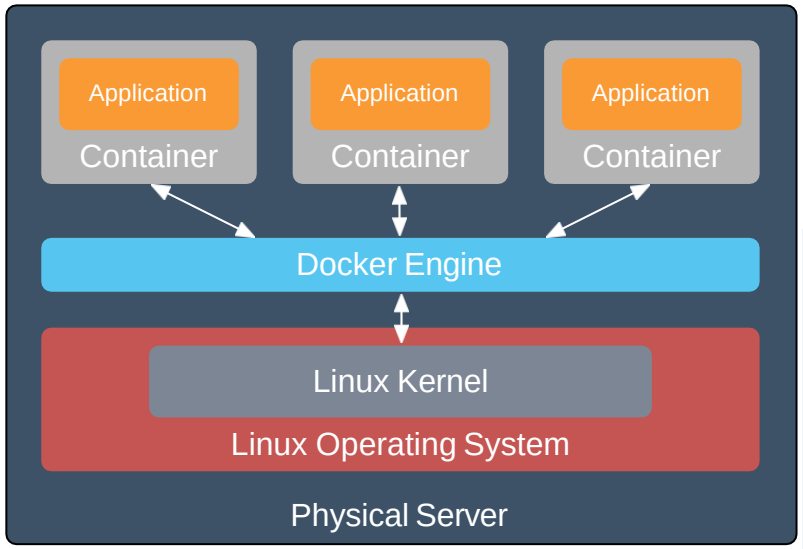




Shared kernel on the host to run multiple guest applications

Containers - Advantages



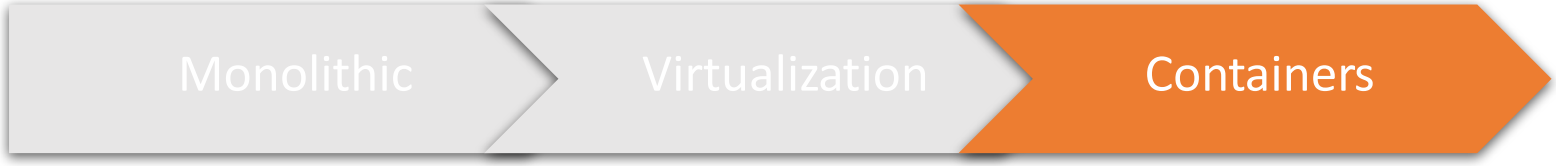


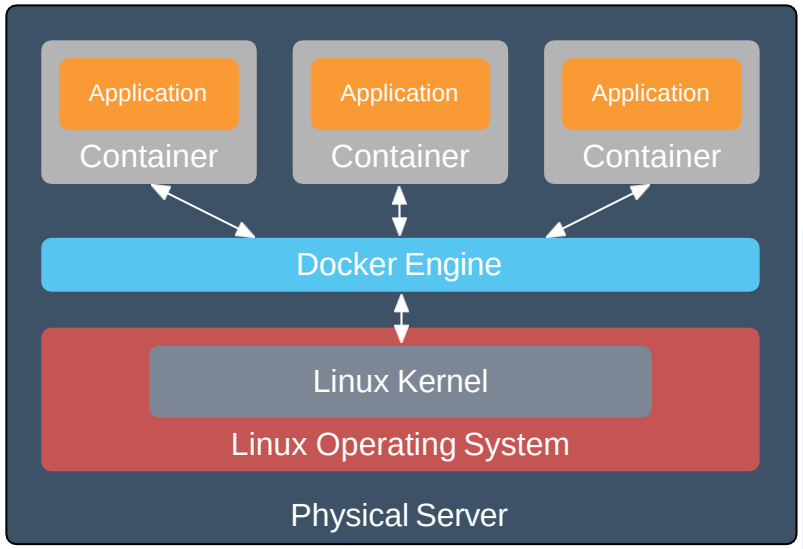
Shared kernel on the host to run multiple guest applications

Advantages over VMs

- Containers are more lightweight
- No need to install a guest Operating System
- Less CPU, RAM, storage overhead
- More containers per machine
- Greater portability

Containers - Challenges





Shared kernel on the host to run multiple guest applications

Container Challenges

- Early Docker focused on single-node operations
- Up to user to cluster Docker hosts and manage deployment of containers on cluster
- User solves for automatic scale out of applications
- User solves for service discovery between application components (microservices)

Container based virtualization

Uses the kernel on the host operating system to run multiple guest instances

- Each guest instance is a container
- Each container has its own
  - Root filesystem
  - Processes
  - Memory
  - Devices
  - Network Ports

Developers

Focus on applications inside the container

Operations

Focus on orchestrating and maintaining containers in production

Container Use Cases

Development

- Allows the ability to define the entire project configuration and tear-down/recreate it easily
- Supports multiple versions of application simultaneously

Container Use Cases

Development

- Allows the ability to define the entire project configuration and tear-down/recreate it easily
- Supports multiple versions of application simultaneously

Test Environments

- Same container that developers run is the container that runs in test lab and production – includes all dependencies
- Well formed API allows for automated building and testing of new containers

Container Use Cases

Development

- Allows the ability to define the entire project configuration and tear-down/recreate it easily
- Supports multiple versions of application simultaneously

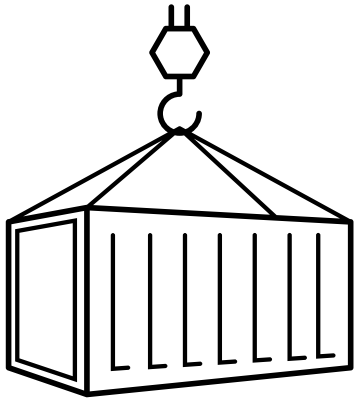
Test Environments

- Same container that developers run is the container that runs in test lab and production – includes all dependencies
- Well formed API allows for automated building and testing of new containers

Micro-Services

- Design applications as suites of services, each written in the best language for the task
- Better resource allocation
- One container per microservice vs. one VM per microservice
- Can define all interdependencies of services with templates

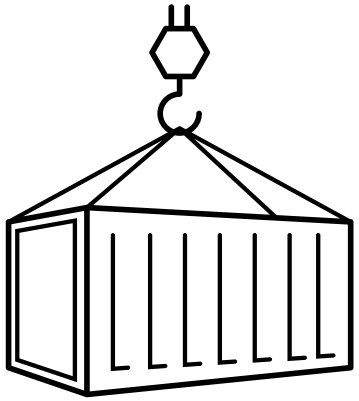
Container Monitoring



Container monitoring involves 3 main components:

- Runtimes
- Orchestrators
- Containers

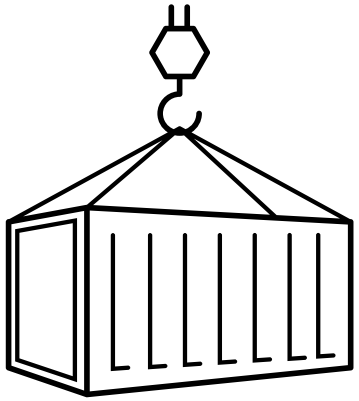
Runtime Monitoring



Container runtimes are software services used to run and manage containers.

- Docker (most popular)
- CRI-O
- containerd

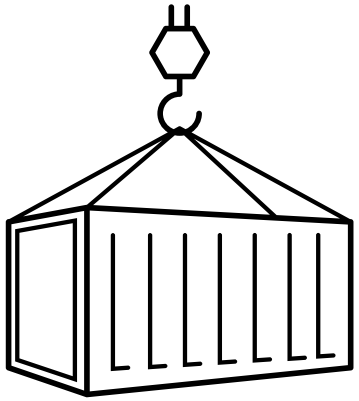
Runtime Monitoring



Metrics monitored at the runtime:

- CPU
- Memory
- Disk I/O
- Network bandwidth

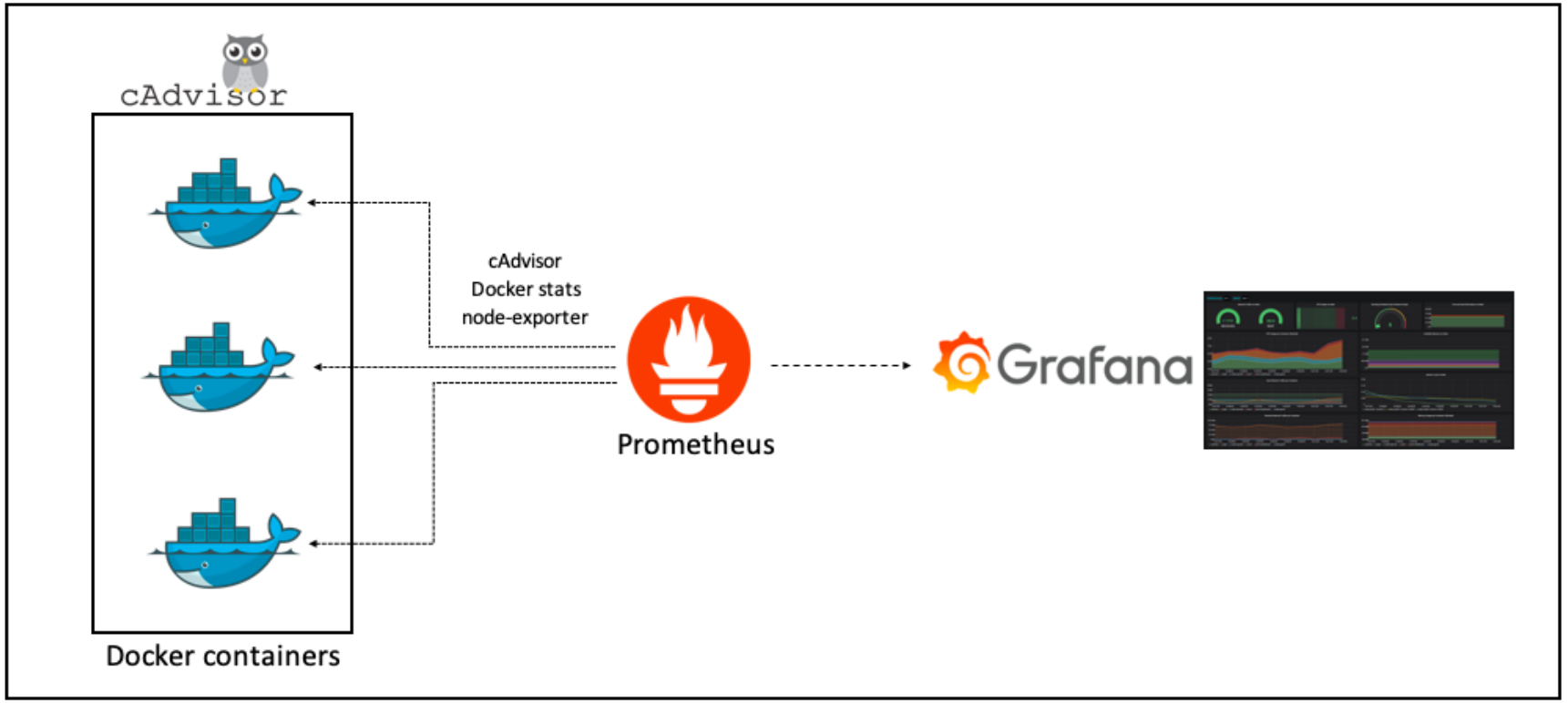
Runtime Monitoring tools



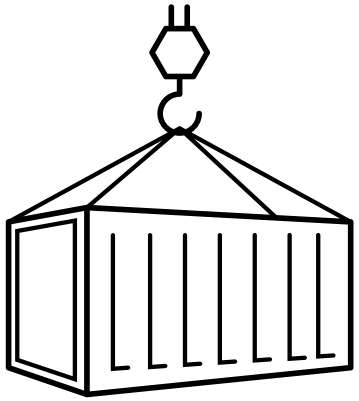
There are many solutions available to monitoring containers:

- cAdvisor
- Prometheus
- Grafana
- Elasticsearch

Runtime monitoring

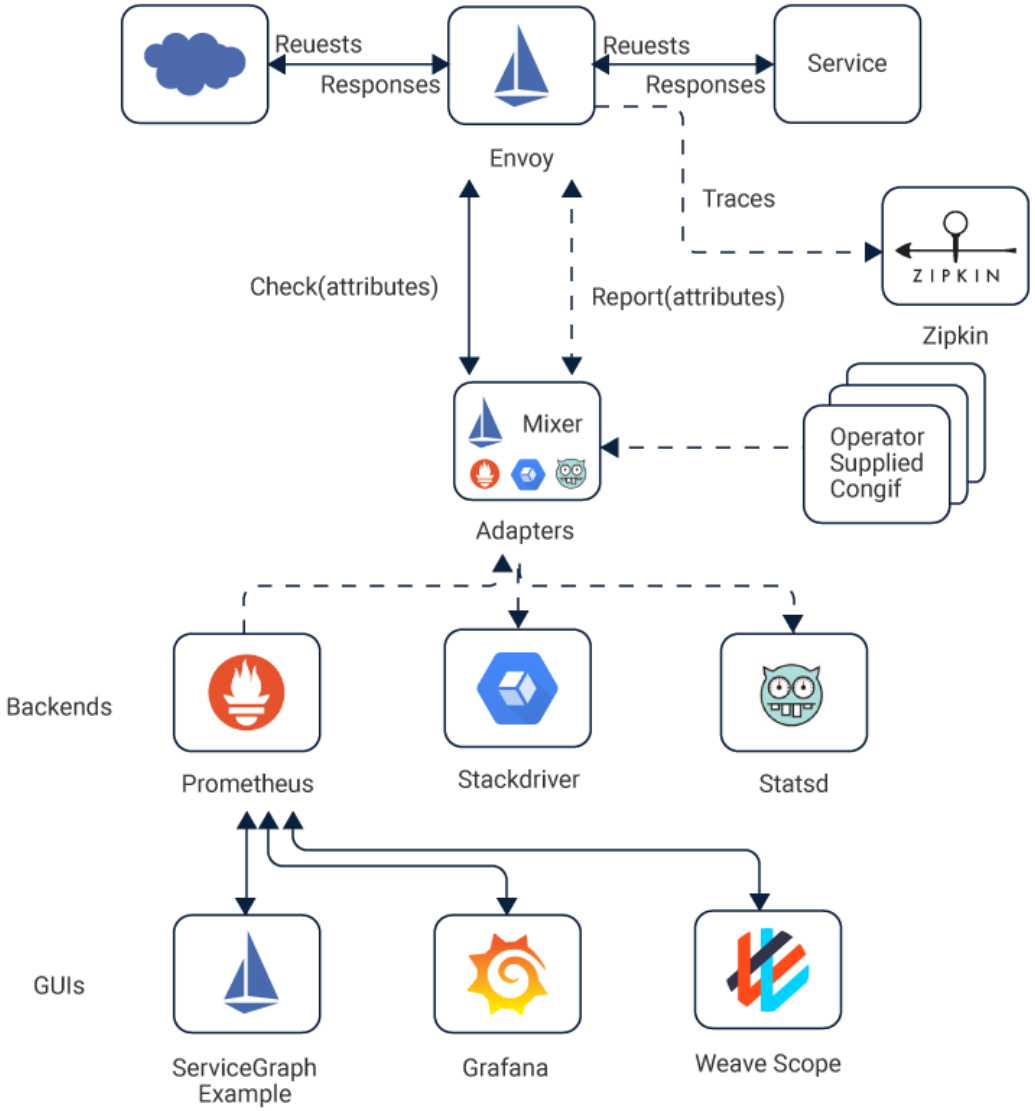


Service mesh

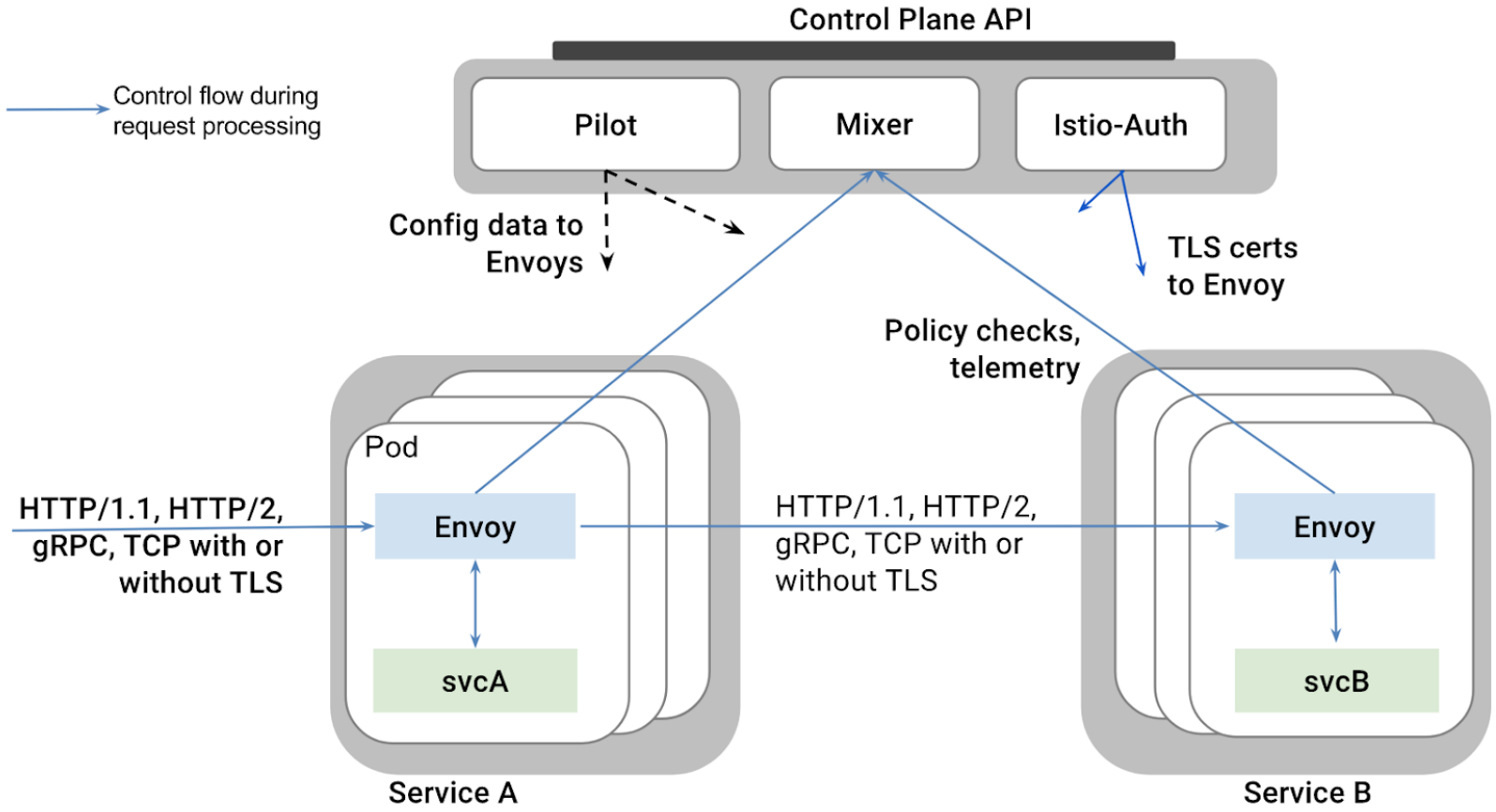


- Dedicated infrastructure layer for making service-to-service communication safe, fast, and reliable.
- Array of lightweight network proxies
  - Deployed alongside application code
- Encryption
  - Developers no longer must add encrypt/decrypt code to their applications
- Circuit breaker pattern support
  - Service signals it's "unhealthy", service mesh removes it, then when fixed adds it back in to pool.

Service mesh

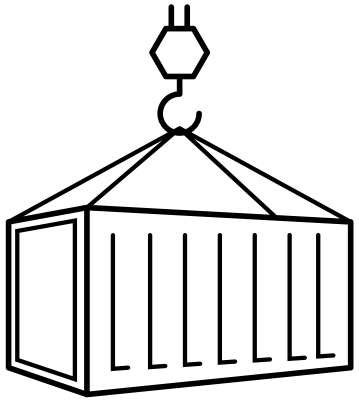


Istio



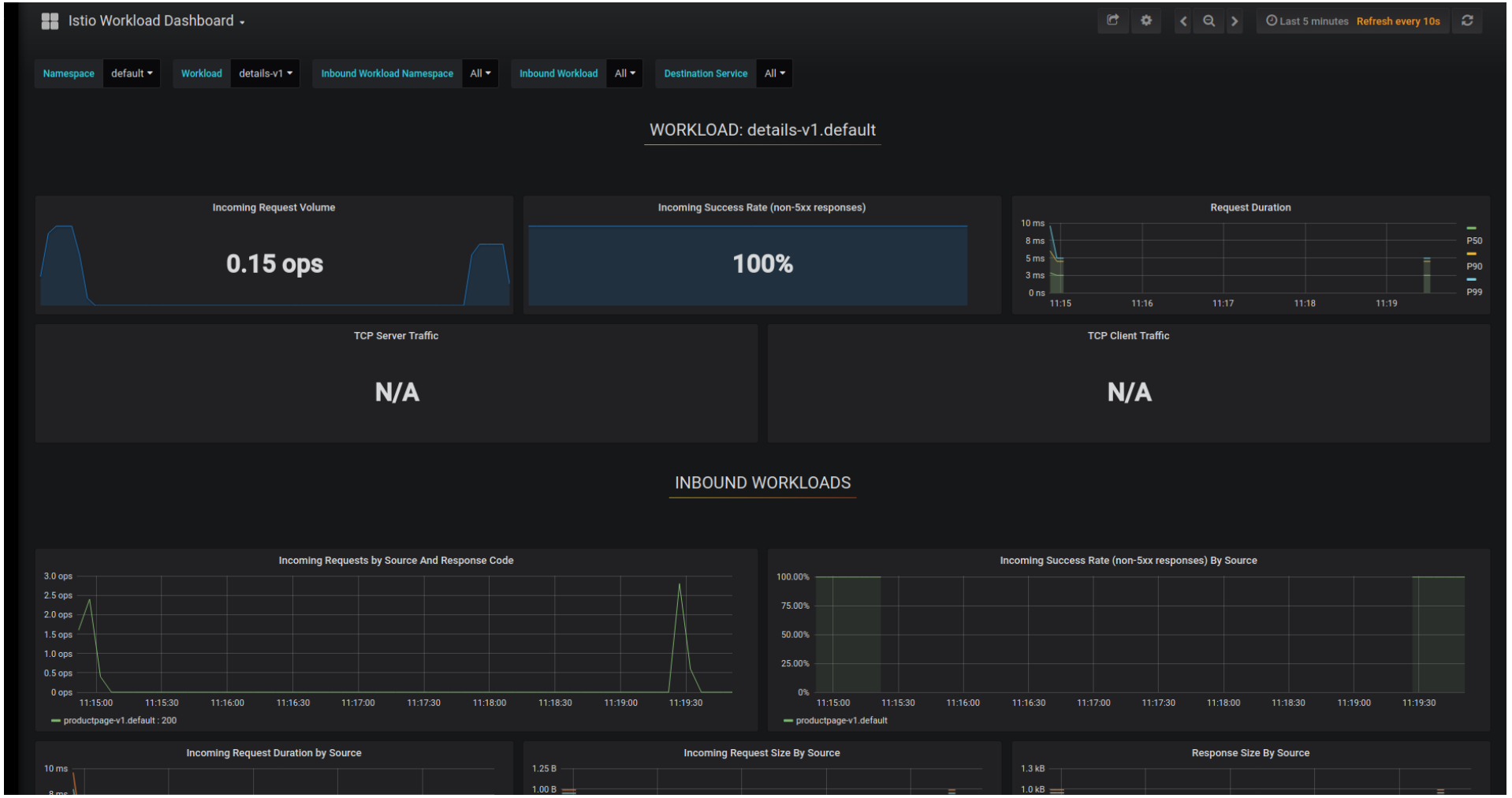
Istio Architecture

Istio monitoring features

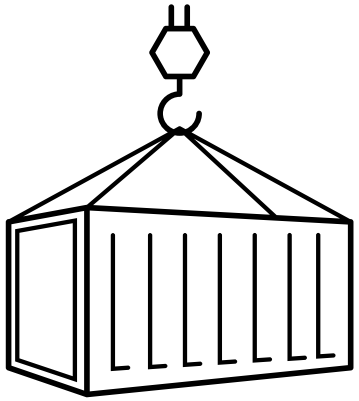


- Metrics without code changes
- Consistent metrics for every app deployed
- Trace flow of requests across services
- Portable across metric backend providers

Istio monitoring grafana

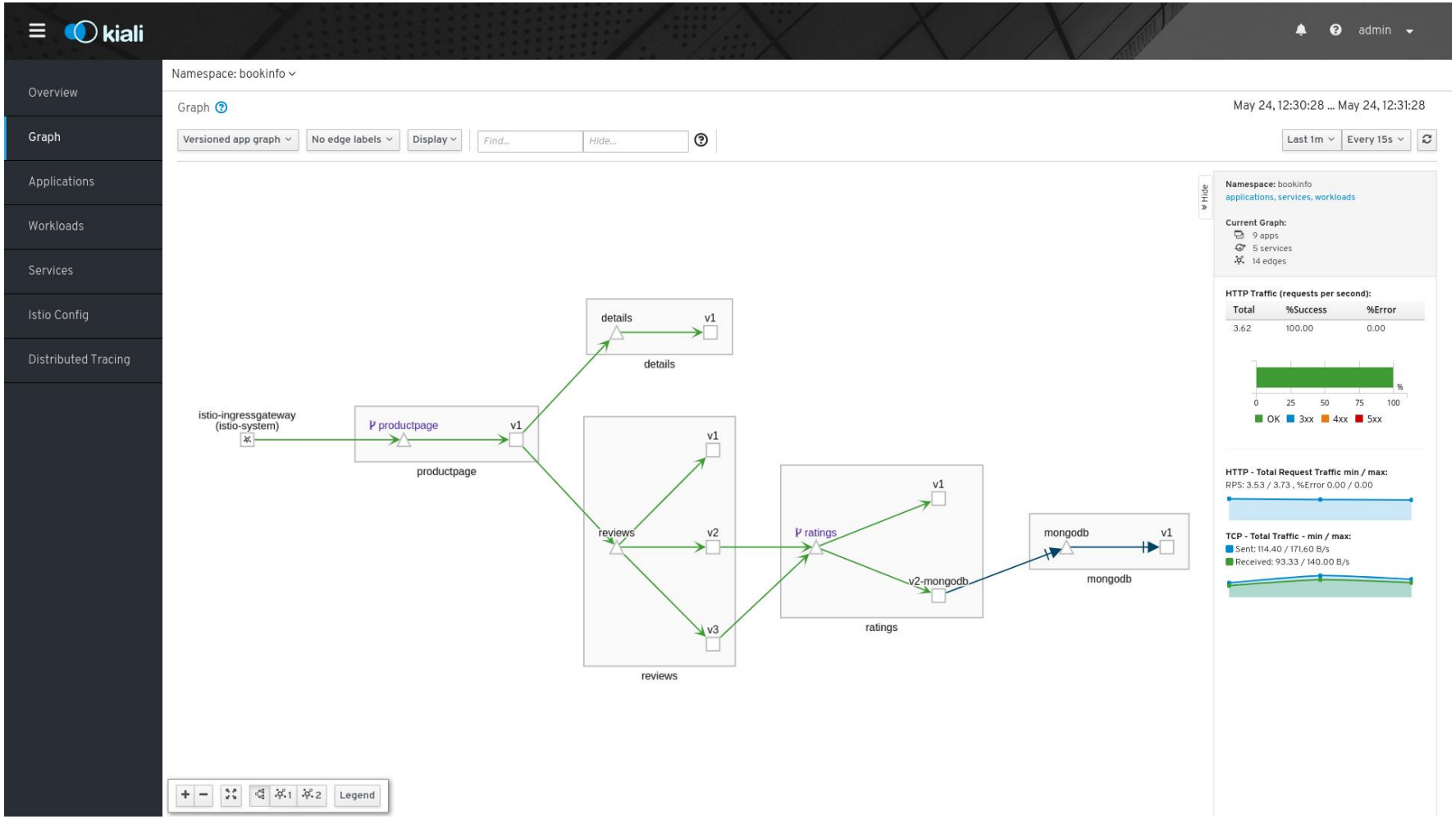


Kiali dashboard

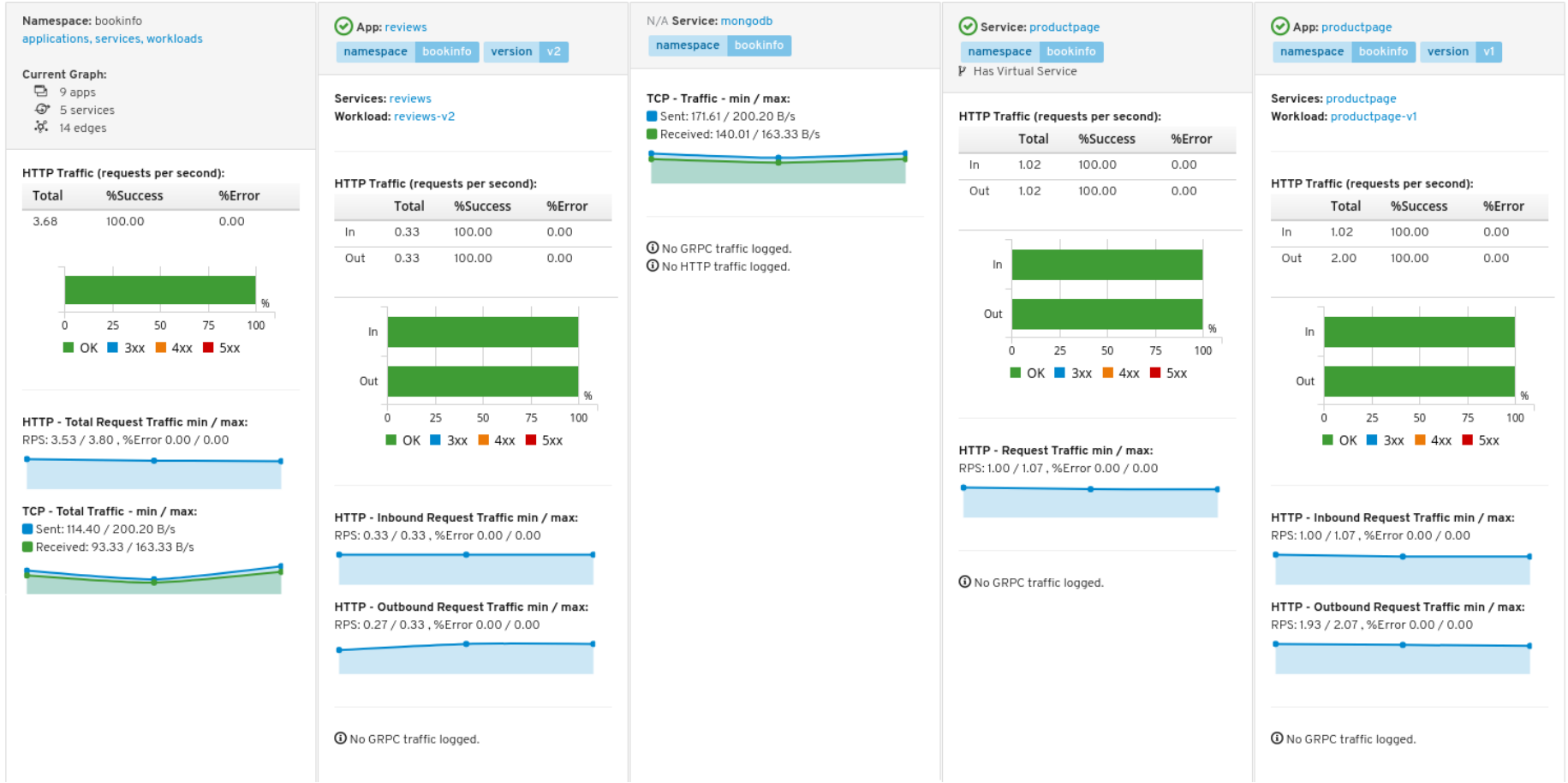


- Kiali helps you define, validate, and observe the connections and microservices of your Istio service mesh.
- Visualizes the service mesh topology
  - Circuit breakers
  - request routing
  - request rates
  - latency
  - more!

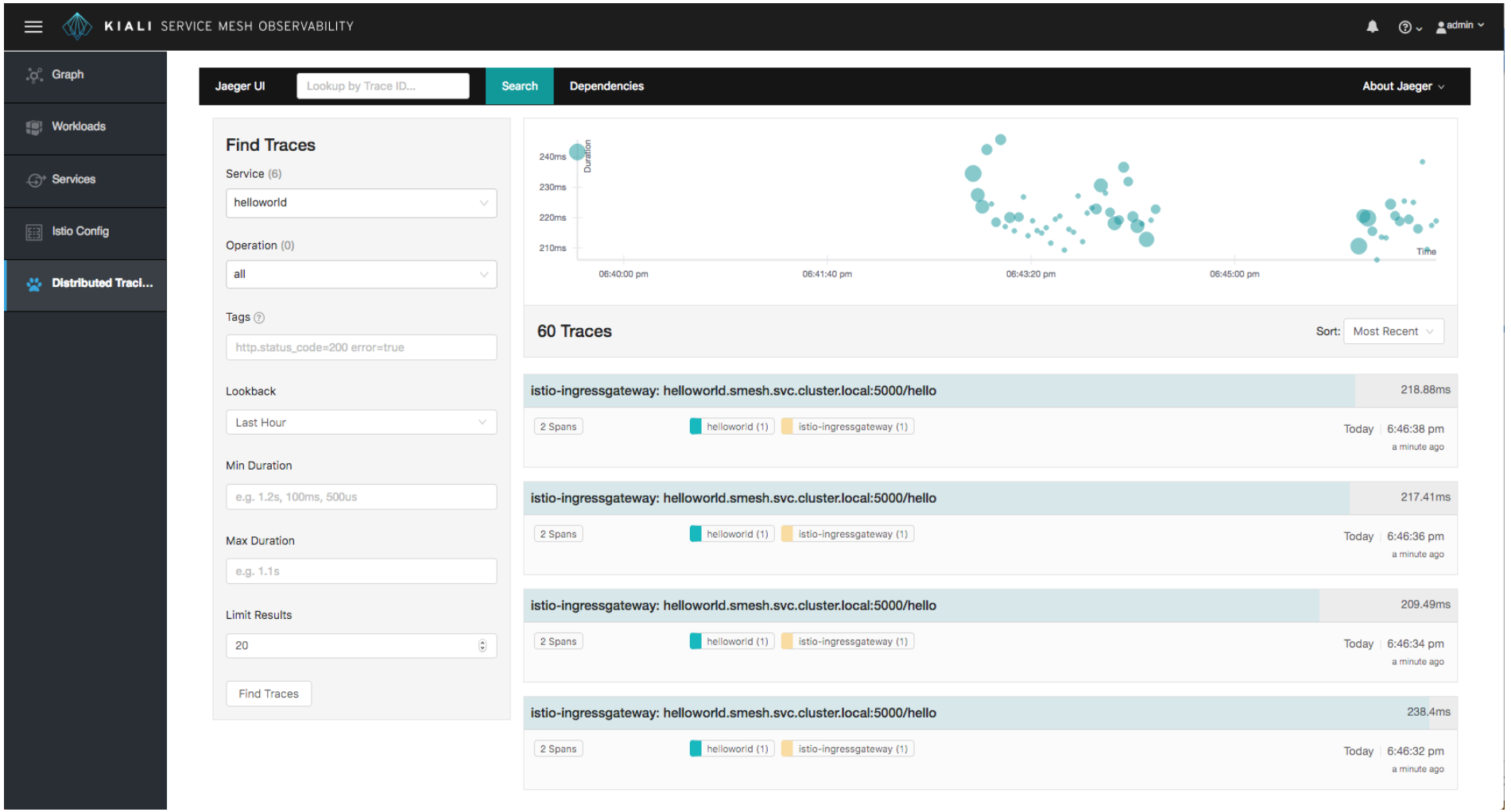
Kiali (Istio dashboard)



Kiali (Istio dashboard)



Kiali distributed tracing (Jaeger)




THANK
YOU