Infrastructure Automation Tools

TEKsystems Global Services

WORKFORCE DEVELOPMENT

CI/CD and Infra Automation

Learning Objectives

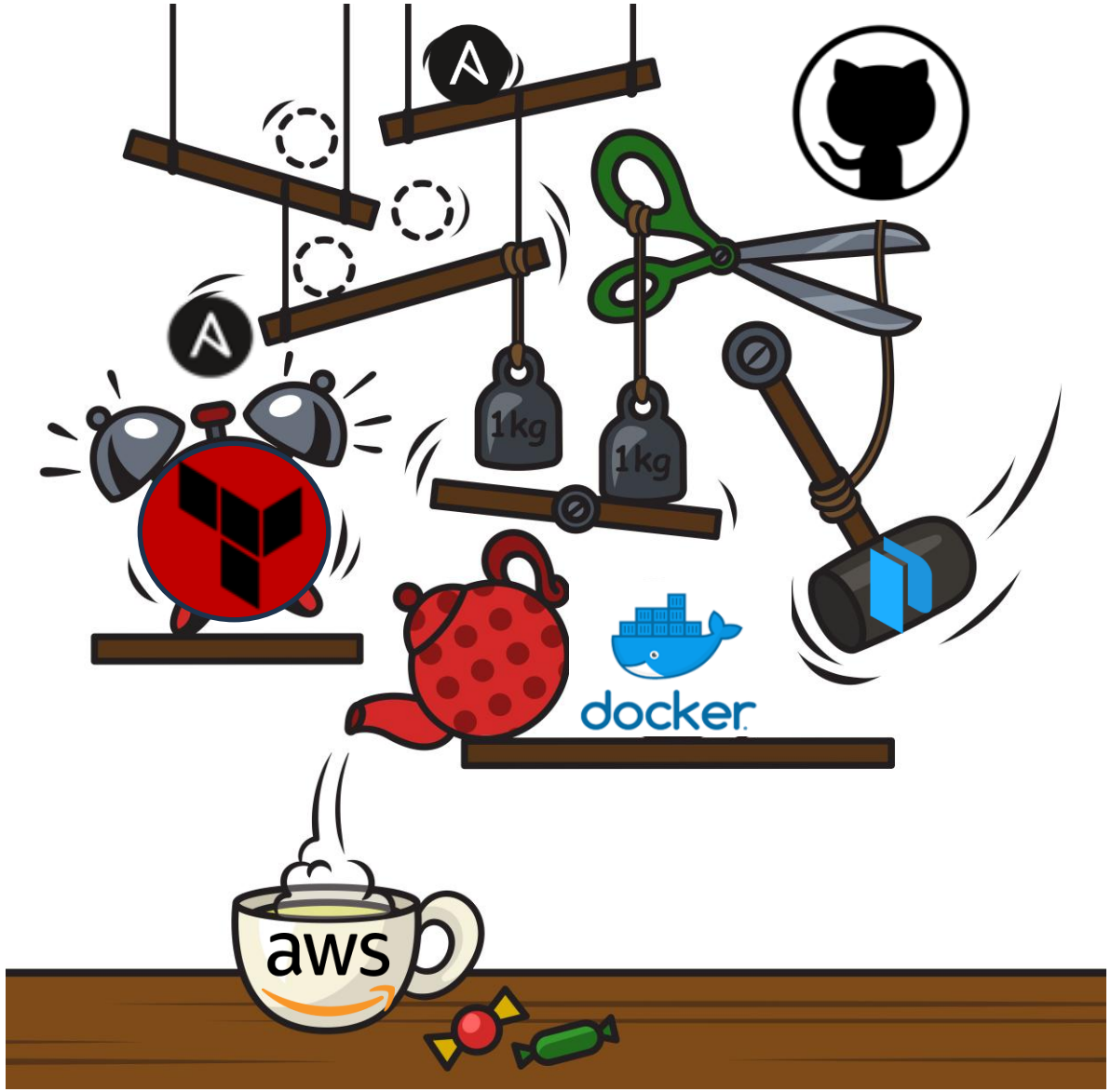
Day 3 Objectives

Today you will build production-grade infrastructure using Terraform and learn how to create and deploy your own custom AMIs with Packer. This gives you the foundation for scalable, repeatable, high-availability environments.

You will learn how to:

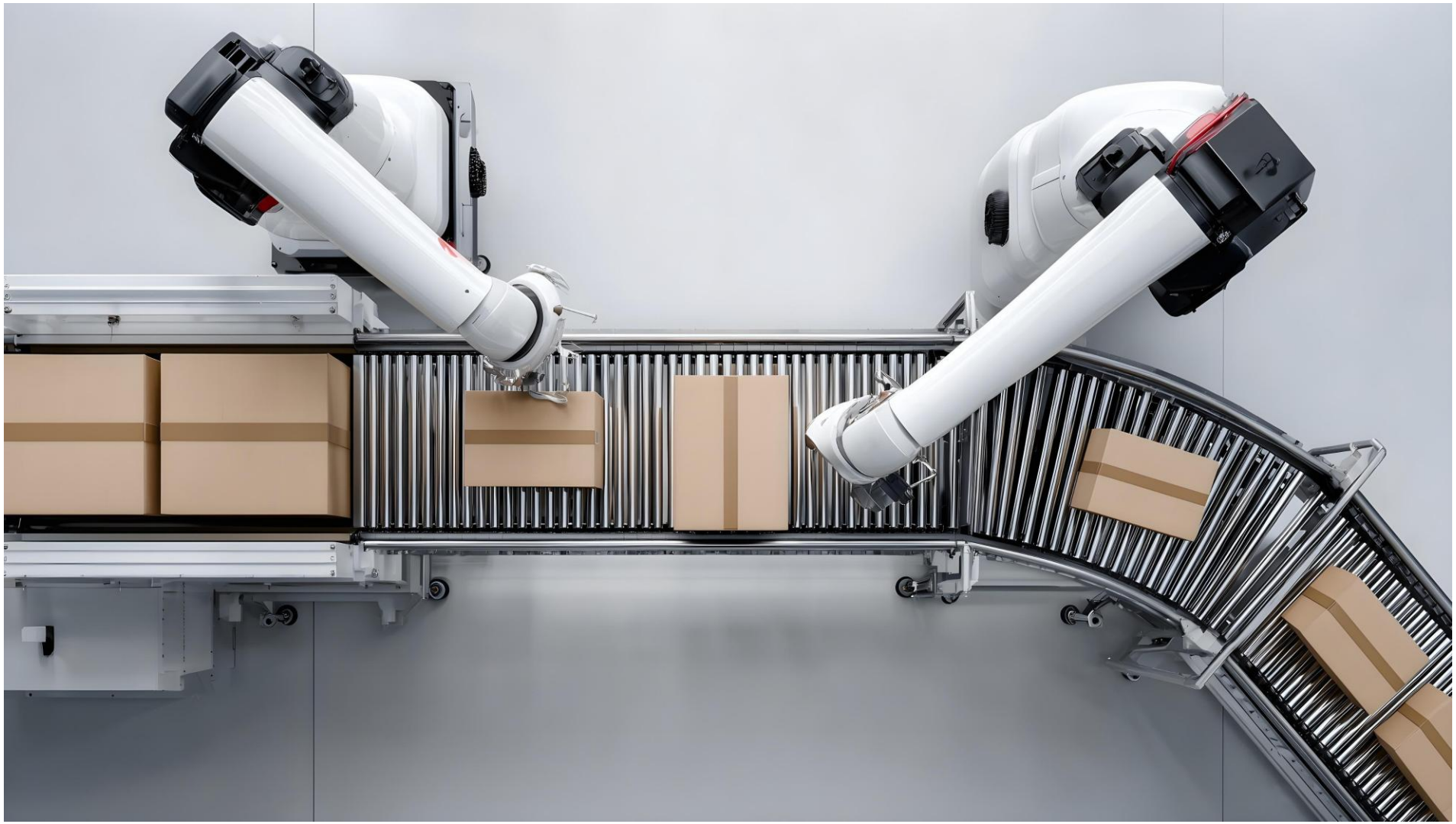
- Deploy infrastructure with **Terraform**, including VPC, subnets, security groups, and an **Auto Scaling Group**
- Build a **high-availability, multi-AZ** architecture
- Use **Packer** to create a custom AMI preconfigured with your application
- Update Terraform to deploy your newly built AMI
- Understand how AMIs + ASGs enable fast, reliable, self-healing infrastructure

CI CD

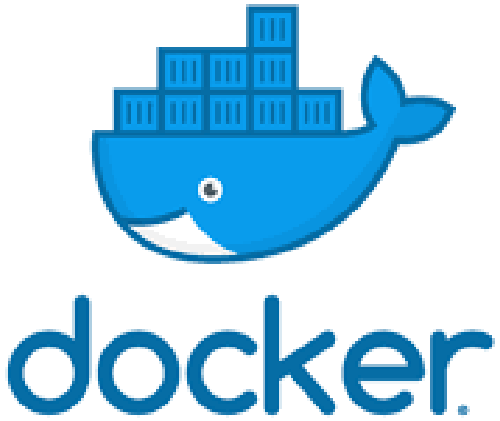


Resiliency

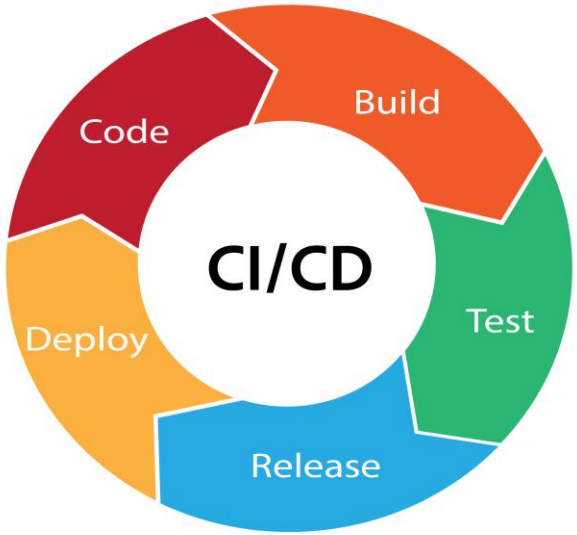
Packer



Automation tools



CI/CD

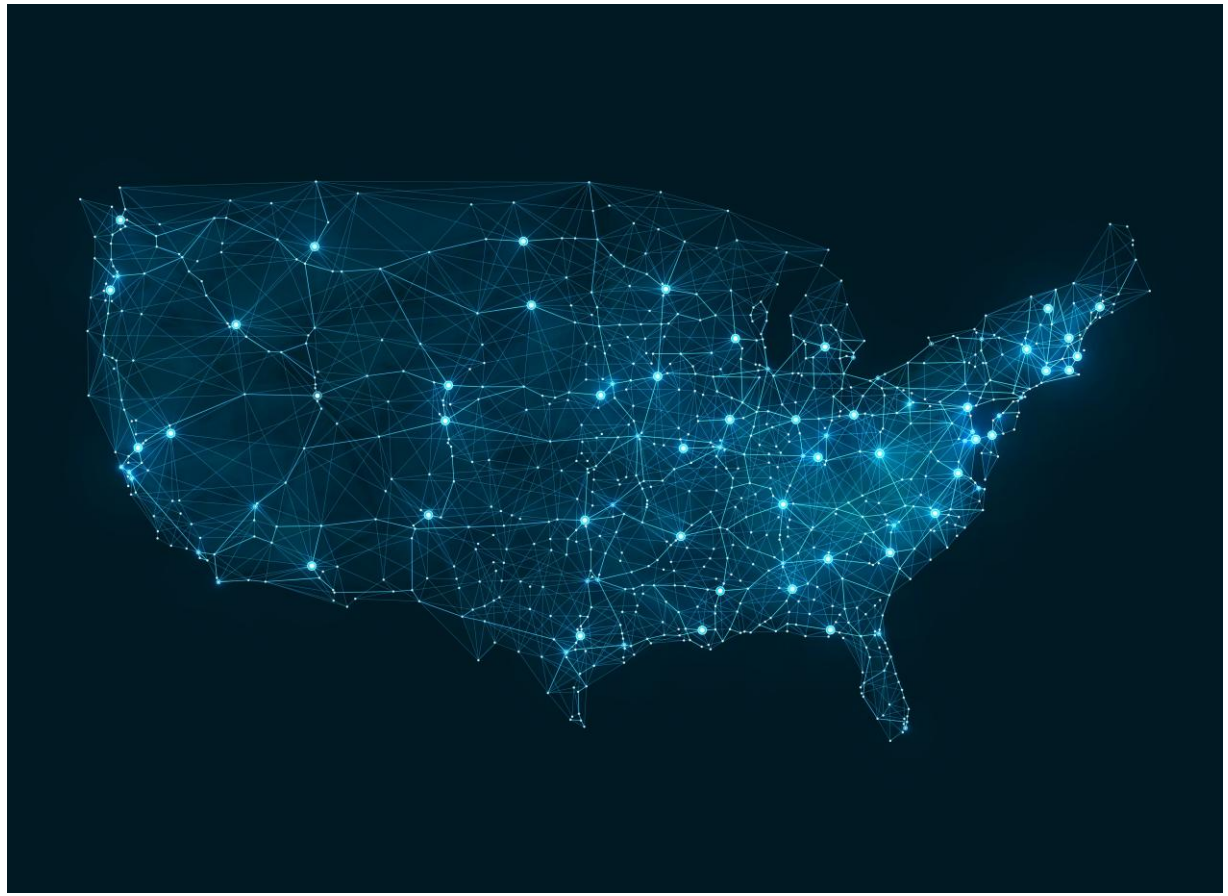


Continuous Integration (CI)
Automatically builds, tests, and validates code every time developers push changes.

Continuous Delivery (CD)
Keeps the application always ready for release with automated packaging and staging.

Continuous Deployment (CD)
Automatically deploys every approved change directly to production or staging envs.

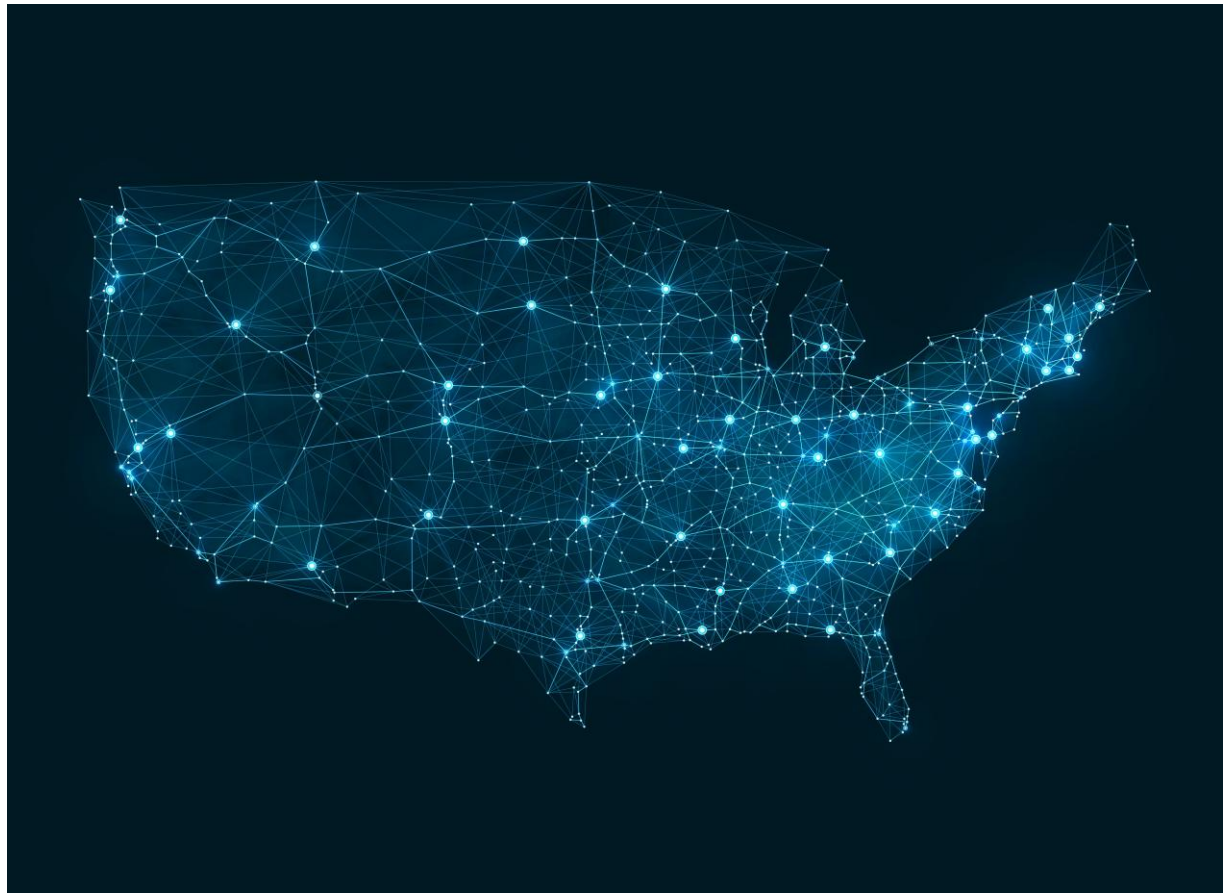
Resiliency in Infrastructure Automation



When automating infrastructure, it's important to understand how cloud regions and availability zones impact reliability and deployment workflows.

Terraform makes it easy to define these settings as code and ensure consistent, resilient deployments.

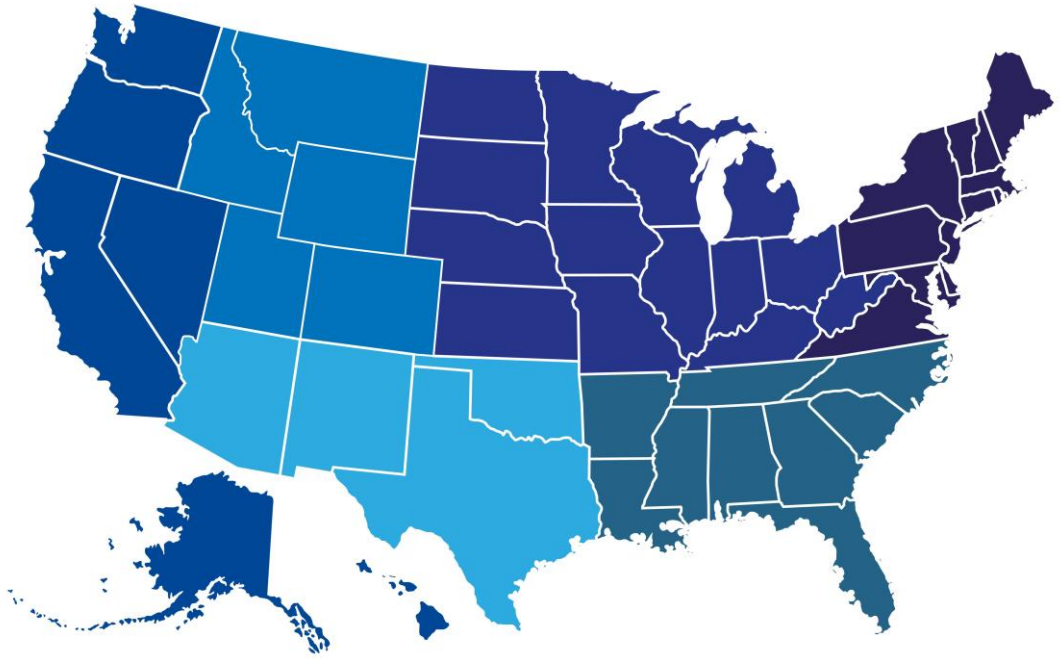
Resiliency in Infrastructure Automation



- **Regions** are geographic areas where your infrastructure runs
- **Availability Zones (AZs)** are isolated data centers inside a region
- Deploying across multiple AZs improves **fault tolerance**
- Auto Scaling Groups and load balancers distribute workloads across AZs
- Terraform's IaC approach ensures this architecture is **repeatable and versioned**

Automation isn't just about creating infrastructure—it's about creating resilient infrastructure every time you deploy.

Resiliency in Infrastructure Automation



Examples of Regions

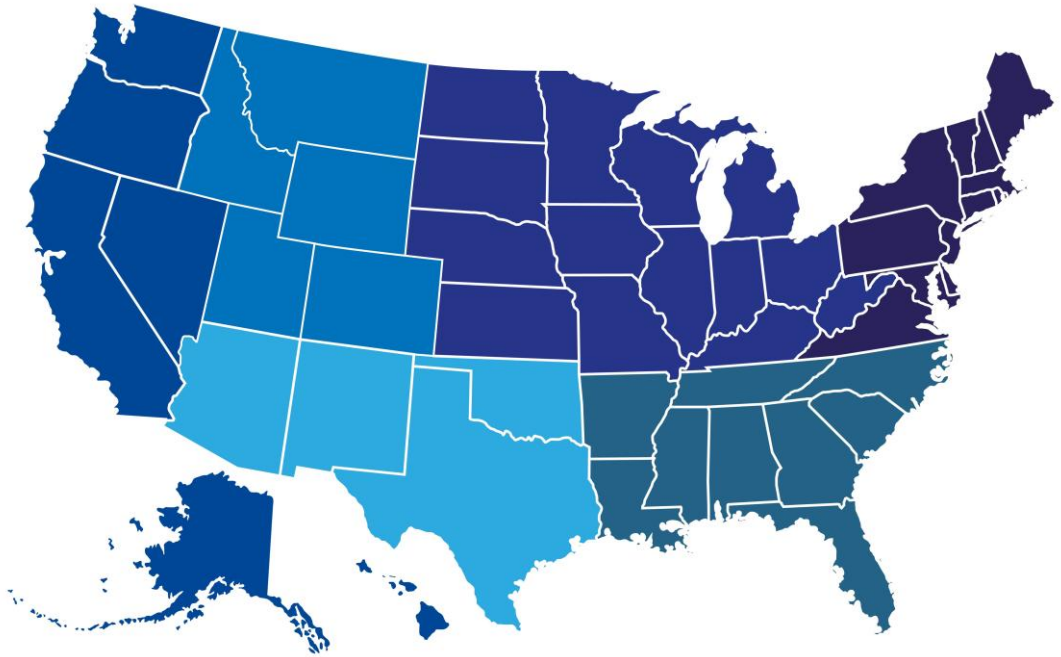
- us-east-1 (N. Virginia)
- us-west-1 (N. California)

  Deploying infrastructure in **multiple regions** gives the highest level of HA because each region is completely independent.

Availability Zones

- Each region contains several AZs (e.g., us-east-1a, us-east-1b, us-east-1c)
- Deploying an application across **multiple AZs in the same region** provides strong resiliency with simpler networking and lower latency

Resiliency in Infrastructure Automation



Cloud providers divide infrastructure into **regions** and **availability zones (AZs)**. These choices directly affect resiliency and uptime.

Multi-region = **maximum HA** but more complex.

Multi-AZ = **high HA**, easier to automate, widely used in real product

Resiliency

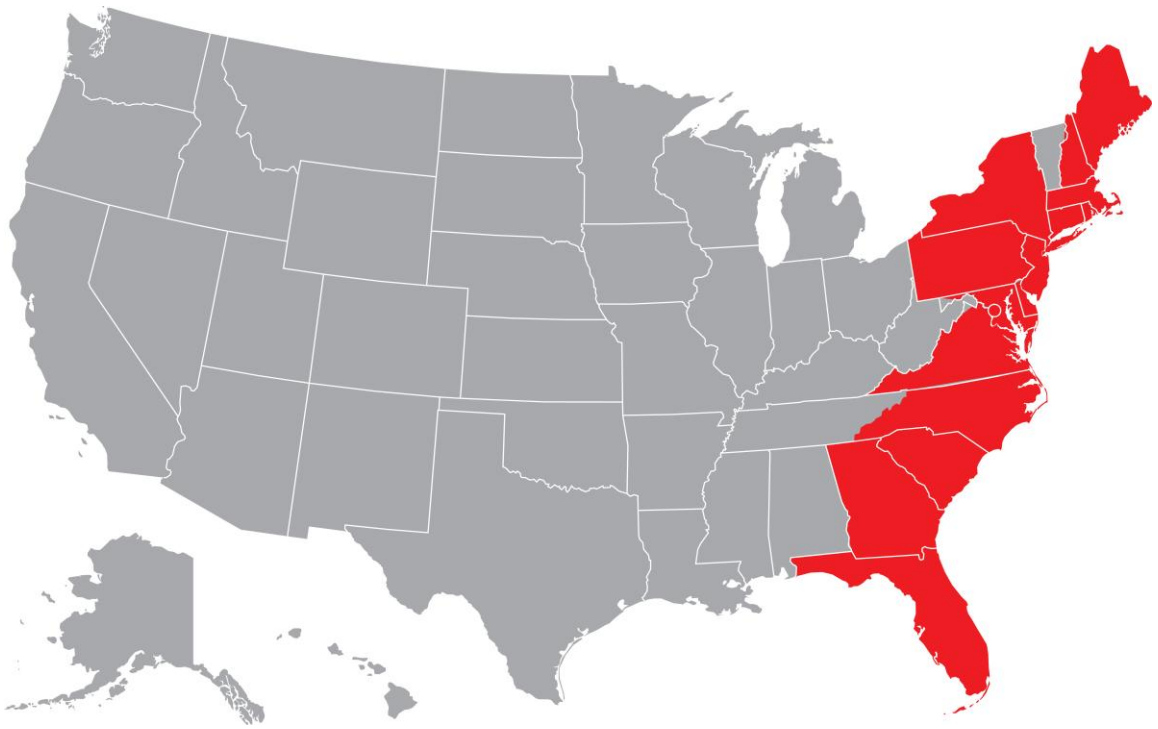
Resiliency is the ability of your infrastructure to remain available even when parts of the system fail. **High Availability (HA)** designs ensure that no single failure takes down your application.

Key points

- Infrastructure continues running even if one server or AZ fails
- Traffic automatically routes to healthy instances
- Self-healing components (like ASGs) replace failed resources
- Reduces downtime and improves reliability for production workloads

Resilient infrastructure is predictable, fault-tolerant, and easier to automate.

Auto Scaling Groups

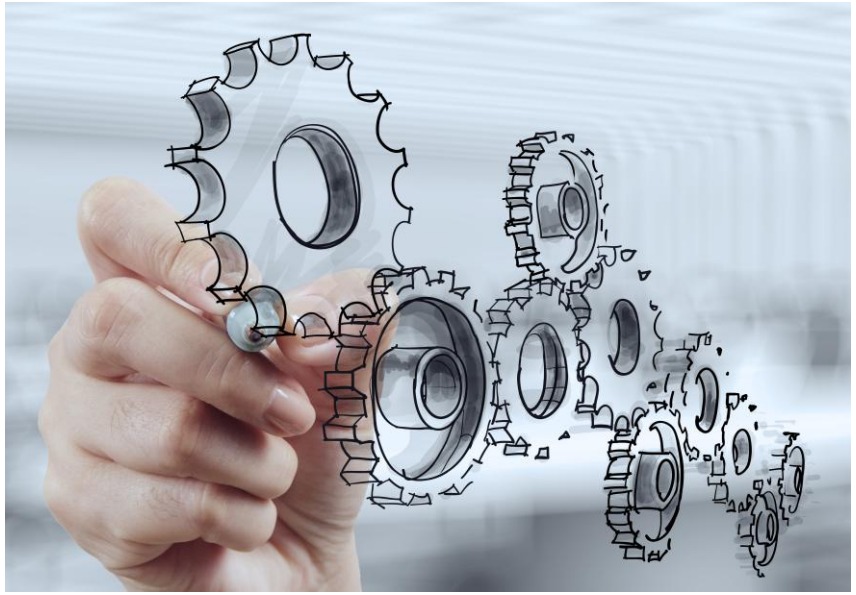


Auto Scaling Groups (ASGs) improve availability by distributing instances across multiple Availability Zones.

Key points

- ASGs launch instances in **two or more AZs** to avoid single points of failure
- If an instance or AZ becomes unhealthy, the ASG automatically replaces it
- Load balancers route traffic only to healthy instances
- Scaling policies ensure the right number of instances based on demand

ASG Terraform



Terraform makes multi-AZ design easy by defining subnets, mappings, and resources as code. This ensures consistent architecture every time you deploy.

Key points

- Create **multiple subnets**, each in a different AZ
- Reference those subnets in ASGs, load balancers, and route tables
- Use data sources to dynamically fetch AZ names
- Define your architecture once and apply it identically across environments

Terraform turns high-availability patterns into simple, reusable templates.

ASG for AMIs - not Containers

Auto Scaling Groups are designed to launch **EC2 instances from AMIs**, not container images. If your application is packaged as a container, the EC2 instances in the ASG must still run an operating system and a container runtime (like Docker) before the app can start.

- ASGs expect a full machine image (AMI) to boot from
- Containers require a runtime such as Docker or containerd
- You can install Docker and run your container using **user data**
- User data runs only once—on the first boot of each instance.

ASGs + AMIs form the "machine-level" layer; containers run *on top* of that layer.

ASG for AMIs - not Containers

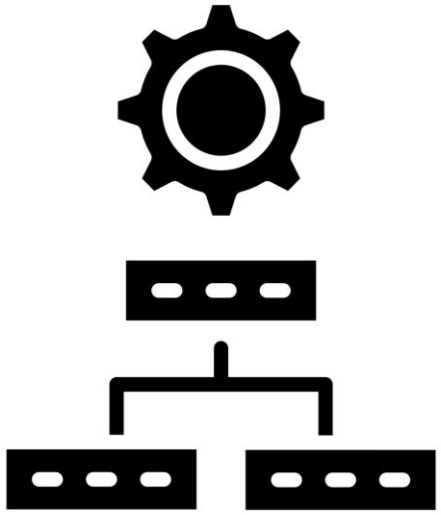
User data allows you to install software and start your application when an instance boots—but it has limitations.

Key points

- User data is **not dynamic**: it only runs on first launch
- If your script fails, the instance may come up broken
- Updating the application means updating the entire user data script
- Harder to maintain and test compared to immutable AMIs
- Not ideal for complex deployments or production-grade reliability

User data works for simple setups but breaks down as deployments evolve, making immutable images (via Packer) a cleaner approach.

Stateless Applications



When applications do not store session data or state locally, they become **stateless**. This makes it much easier to achieve high availability.

Key points

- Any instance can handle any request—no dependency on a specific server
- Traffic can shift instantly between instances without user impact
- ASGs can replace unhealthy instances with no downtime
- Scaling up or down becomes seamless

Ephemeral, stateless apps are the foundation of resilient, cloud-native architectures.

Database Resiliency

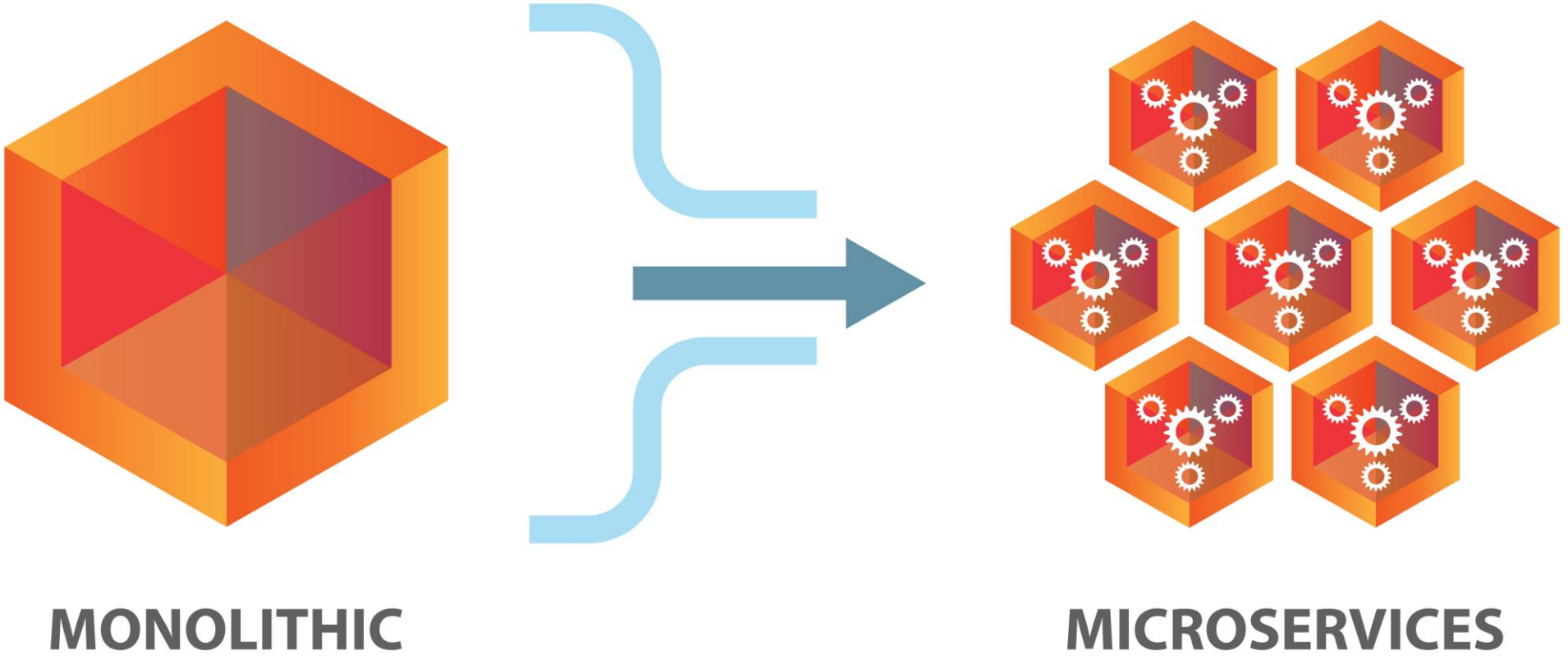
Unlike stateless apps, databases maintain persistent state and cannot be freely replaced or shuffled between AZs.

Key points

- Databases often use **hot–warm architectures**: one writable primary (hot) and one or more read-only replicas (warm)
- Failover promotes a replica to primary if the main database fails
- Ensures data consistency while maintaining availability
- Much more complex than stateless application HA

Applications scale horizontally and can be ephemeral; databases scale carefully and require replication strategies.

Microservices vs Monolithic



Microservices vs Monolithic

Modern infrastructure is shifting from large monolithic applications to smaller, independent microservices. This allows teams to deploy applications in highly available, scalable, and fault-tolerant patterns.

Monolithic

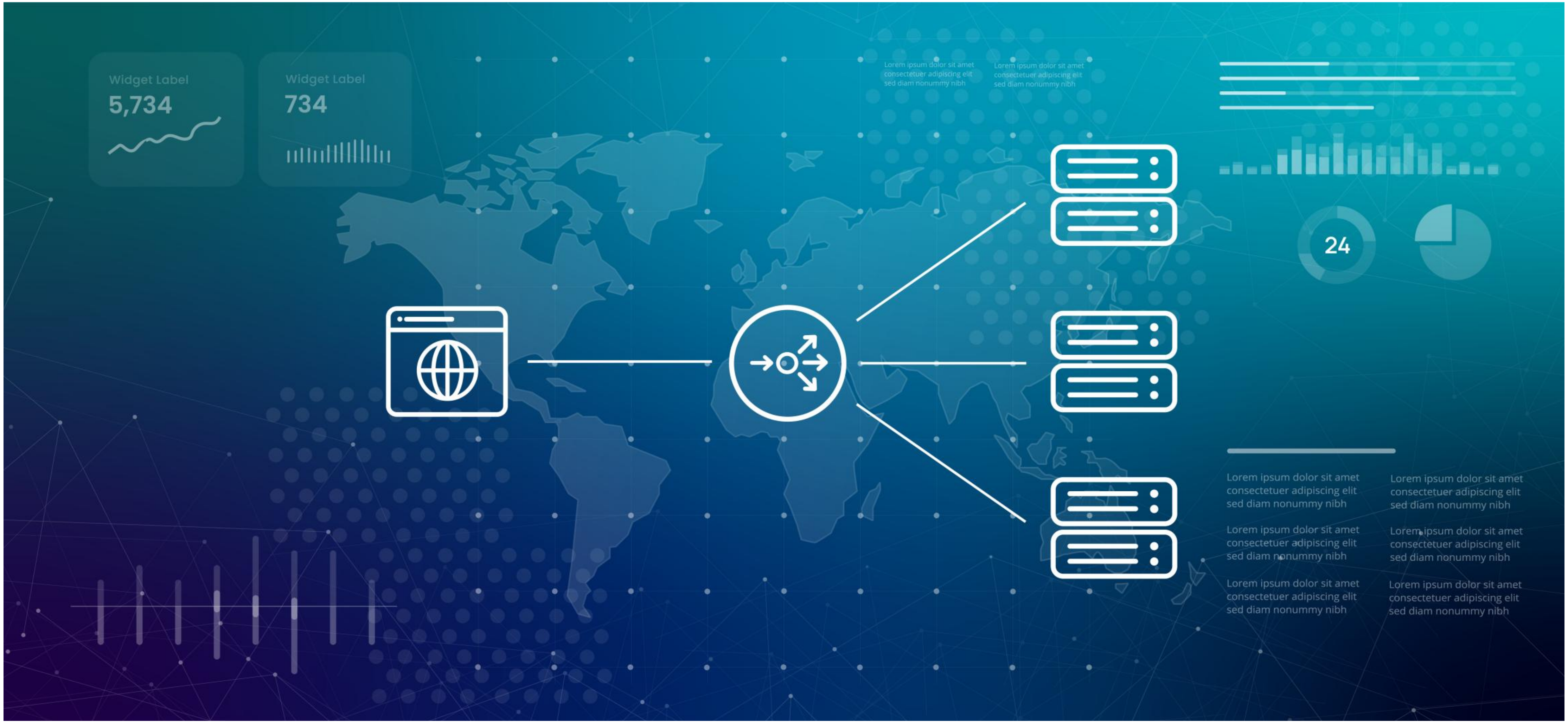
- One large application running on one or few servers
- Harder to scale or deploy without downtime
- Failures impact the entire system

Microservices

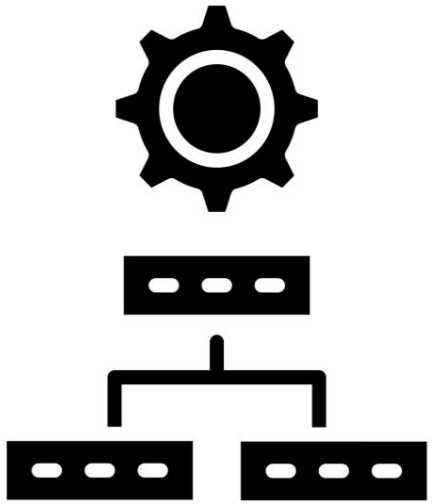
- Many small, independent services
- Easy to scale horizontally and run stateless
- Perfect for HA setups, auto scaling, and ephemeral infrastructure

Microservices align naturally with modern automation, CI/CD pipelines, and cloud-native resiliency patterns.

Load Balancing



Load Balancing



A load balancer distributes incoming traffic across multiple backend instances to improve availability, performance, and reliability.

Key points

- Routes traffic only to healthy instances
- Prevents overload on a single server
- Enables zero-downtime deployments and rolling updates
- Essential component in any HA architecture (especially with ASGs)

Load balancers make it possible for multiple instances to behave like one unified service.

Load Balancing Strategies

Load balancers use different algorithms to decide which instance should receive each request.

Common strategies

- **Round Robin** — Each request is sent to the next instance in rotation
- **Least Connections** — Sends traffic to the instance with the fewest active connections
- **IP Hash / Session Affinity** — Uses client IP or metadata to route the same user to the same instance
- **Weighted Distribution** — Some instances receive more traffic based on assigned weights
- **Health Checks** — Automatically removes unhealthy instances from rotation

POP QUIZ:

What is the main benefit of deploying across multiple AWS regions?

A. Faster EC2 boot times
B. Lower cost for EC2 instances
C. Automatic scaling without configuration
D. Better Failover solution

POP QUIZ:

What is the main benefit of deploying across multiple AWS regions?

A. Faster EC2 boot times
B. Lower cost for EC2 instances
C. Automatic scaling without configuration
D. **Better Failover solution**

POP QUIZ:

Which of the following best describes an Availability Zone (AZ)?

A. A physically isolated datacenter inside a region
B. A separate AWS account
C. A virtual private cloud
D. A network of edge locations

POP QUIZ:

Which of the following best describes an Availability Zone (AZ)?

A. **A physically isolated datacenter inside a region**
B. A separate AWS account
C. A virtual private cloud
D. A network of edge locations

POP QUIZ:

What problem does a load balancer primarily solve in HA architectures?

A. Encrypting data at rest
B. Selecting the fastest AZ
C. Distributing traffic across multiple healthy instances
D. Automatically building AMIs

POP QUIZ:

What problem does a load balancer primarily solve in HA architectures?

A. Encrypting data at rest
B. Selecting the fastest AZ
C. **Distributing traffic across multiple healthy instances**
D. Automatically building AMIs

POP QUIZ:

In the context of Terraform and ASGs, why is immutable infrastructure preferred?

A. It is cheaper than mutable instances
B. AMIs automatically delete logs
C. Every deployment starts from a known, clean state
D. It removes the need for CI/CD

POP QUIZ:

In the context of Terraform and ASGs, why is immutable infrastructure preferred?

A. It is cheaper than mutable instances
B. AMIs automatically delete logs
C. **Every deployment starts from a known, clean state**
D. It removes the need for CI/CD

POP QUIZ:

Why is user data not ideal for frequently changing applications?

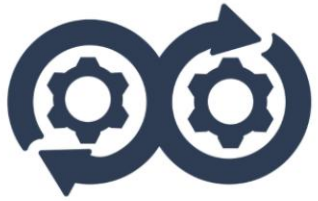
A. It only runs when the instance is first created
B. It only works in the first AZ of a region
C. It always fails on the first run
D. It requires a load balancer

POP QUIZ:

Why is user data not ideal for frequently changing applications?

A. **It only runs when the instance is first created**
B. It only works in the first AZ of a region
C. It always fails on the first run
D. It requires a load balancer

Lab: Terraform Infra



Packer

Packer

Packer is an automation tool used to create **machine images** (like AMIs in AWS) in a **repeatable, consistent, and automated** way.

- Creates **Golden AMIs** that already include your dependencies, configurations, and application.
- Builds images **without needing to configure instances manually** after they launch.
- Can run on your machine or inside **CI/CD pipelines** (like GitHub Actions).
- Produces **immutable**, versioned images — essential for reliable Auto Scaling Groups.

Packer

- Ensures every EC2 instance is **identical** (no configuration drift).
- Launches are **fast**, because everything is baked into the AMI.
- Greatly improves **resiliency**, **rollback**, and **security**.
- A cornerstone of **immutable infrastructure**.
- These are operating system images, they are not container images

How Packer Creates AMIs

Packer builds AMIs in a simple, predictable way:

- It **launches a temporary EC2 instance** from a base image (like Amazon Linux 2023).
- It **installs your application and all required dependencies** using scripts or provisioners.
- Once the instance is fully configured, Packer **takes a snapshot of that machine**.
- That snapshot becomes a **new AMI** (Amazon Machine Image).
- Packer then **terminates the temporary instance**, leaving you with a clean, ready-to-use Golden AMI.

Packer

Packer Provisioners

Packer doesn't just build AMIs — it **provisions** them using different tools. A *provisioner* is how Packer installs software, copies files, and configures the temporary VM before snapshotting it into an AMI.

Common Provisioners

- **Shell (most common)** — run Bash commands or scripts
- **Ansible** — run playbooks to configure the machine
- **Chef / Puppet** — infrastructure configuration tools
- **File** — upload files into the instance
- **PowerShell** — Windows automation
- **Windows Shell** — run commands on Windows images

Packer starts a clean VM → runs provisioners to configure it → snapshots the final result into a **Golden AMI**.

Basics of Images and Containers

To run a containerized app directly on EC2, the AMI must include:

- Docker installed
- Your container image pre-pulled
- A service that starts the container on boot

Why do this?

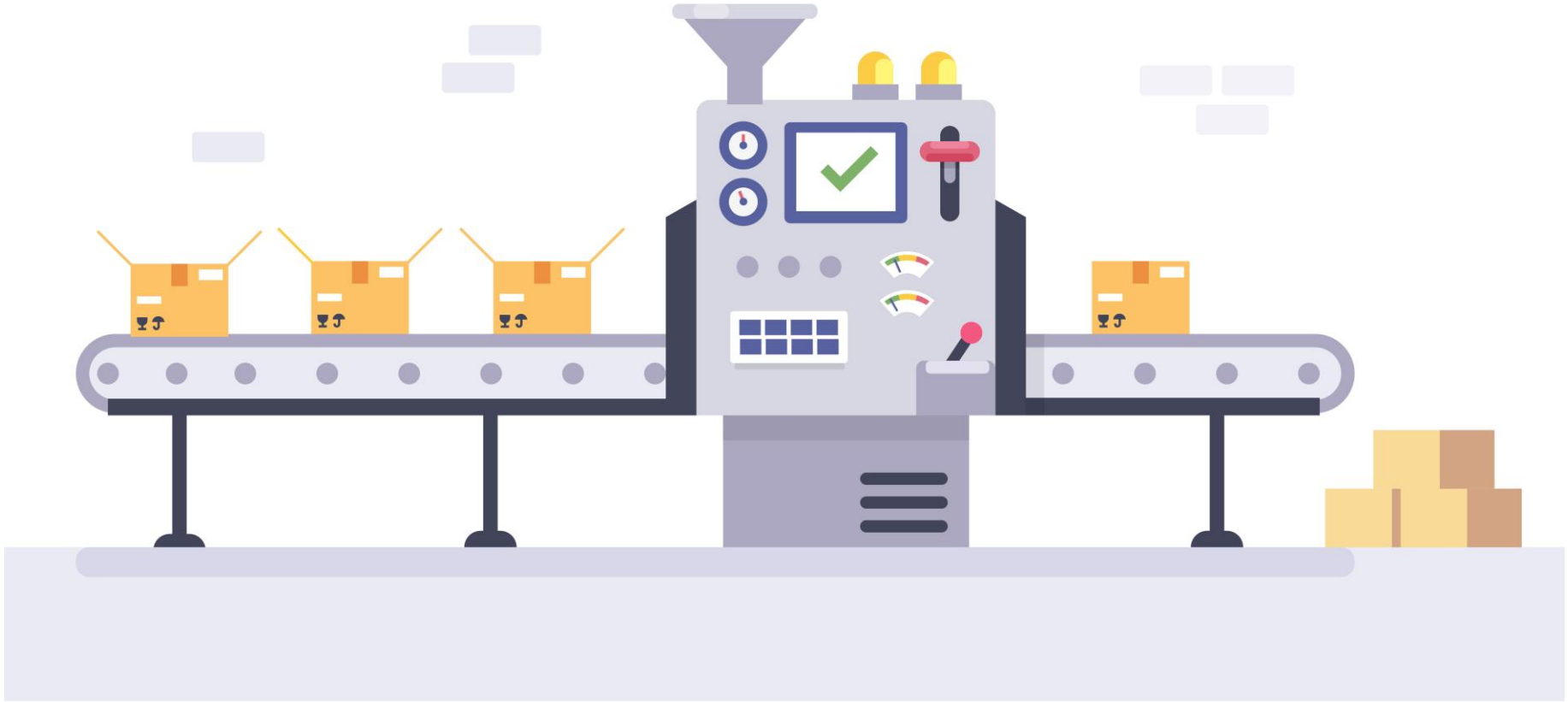
- Faster launches (no setup at boot)
- More reliable than user data
- Ensures every instance runs the exact same version

Launching the instance instantly starts your application.

Continuous Integrations / Delivery

This pipeline takes your application code from a simple **git push** through testing, image building, and finally produces a **fully baked AMI** ready for deployment. Each workflow step automates a stage—from building and pushing the Docker image to generating a production-ready AMI—ensuring consistent, repeatable delivery.

CI CD



Golden Image

POP QUIZ:

What is the primary purpose of Packer in an infrastructure pipeline?

A. To run Terraform plans
B. To build machine images (AMIs) in an automated, repeatable way
C. To configure running EC2 instances with running AMIs
D. To deploy container images to ECS

POP QUIZ:

What is the primary purpose of Packer in an infrastructure pipeline?

A. To run Terraform plans
B. **To build machine images (AMIs) in an automated, repeatable way**
C. To configure running EC2 instances with running AMIs
D. To deploy container images to ECS

POP QUIZ:

How does Packer create a Golden AMI?

A. By copying the latest public AMI from AWS, uploading code and snapshotting it
B. By asking Terraform for the most recent AMI
C. By downloading an AMI from Docker Hub, uploading an app and snapshotting it
D. By launching a temporary EC2 instance, provisioning it, and snapshotting it

POP QUIZ:

How does Packer create a Golden AMI?

A. By copying the latest public AMI from AWS, uploading code and snapshotting it
B. By asking Terraform for the most recent AMI
C. By downloading an AMI from Docker Hub, uploading an app and snapshotting it
D. **By launching a temporary EC2 instance, provisioning it, and snapshotting it**

POP QUIZ:

why is installing Docker and pulling your app image done *inside* the Packer build instead of user data preferred?

A. Because Packer is cheaper than user data
B. User data does not support installing Docker
C. User data only runs on the first boot
D. Docker cannot run in user data

POP QUIZ:

why is installing Docker and pulling your app image done *inside* the Packer build instead of user data preferred?

A. Because Packer is cheaper than user data
B. User data does not support installing Docker
C. **User data only runs on the first boot**
D. Docker cannot run in user data

POP QUIZ:

Why is using Packer considered an “immutable infrastructure” pattern?

A. Because it prevents Terraform from creating new resources
B. Because instances are configured on launch rather than after launch
C. Because each deployment produces a new AMI that replaces old server state
D. Because it makes EC2 instances impossible to modify

POP QUIZ:

Why is using Packer considered an “immutable infrastructure” pattern?

A. Because it prevents Terraform from creating new resources
B. Because instances are configured on launch rather than after launch
C. **Because each deployment produces a new AMI that replaces old server state**
D. Because it makes EC2 instances impossible to modify

POP QUIZ:

What does Packer's "provisioner" block do?

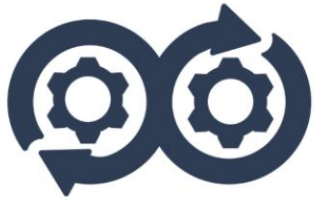
A. Manages load balancing policies
B. Configures the base image by running scripts/commands on the temporary instance
C. Controls how Terraform builds AMIs by running scripts to upload code in AMI
D. Stores secrets for the image

POP QUIZ:

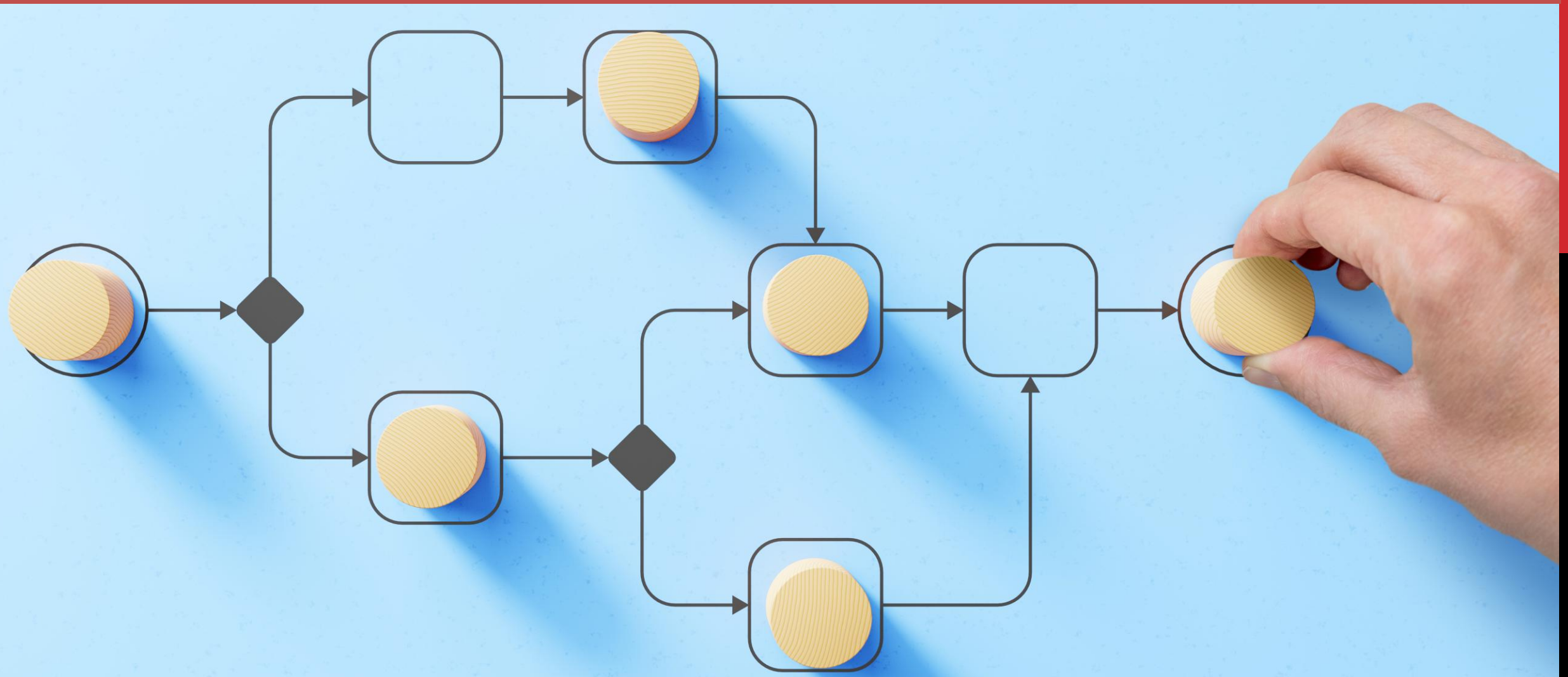
What does Packer's "provisioner" block do?

A. Manages load balancing policies
B. **Configures the base image by running scripts/commands on the temporary instance**
C. Controls how Terraform builds AMIs by running scripts to upload code in AMI
D. Stores secrets for the image

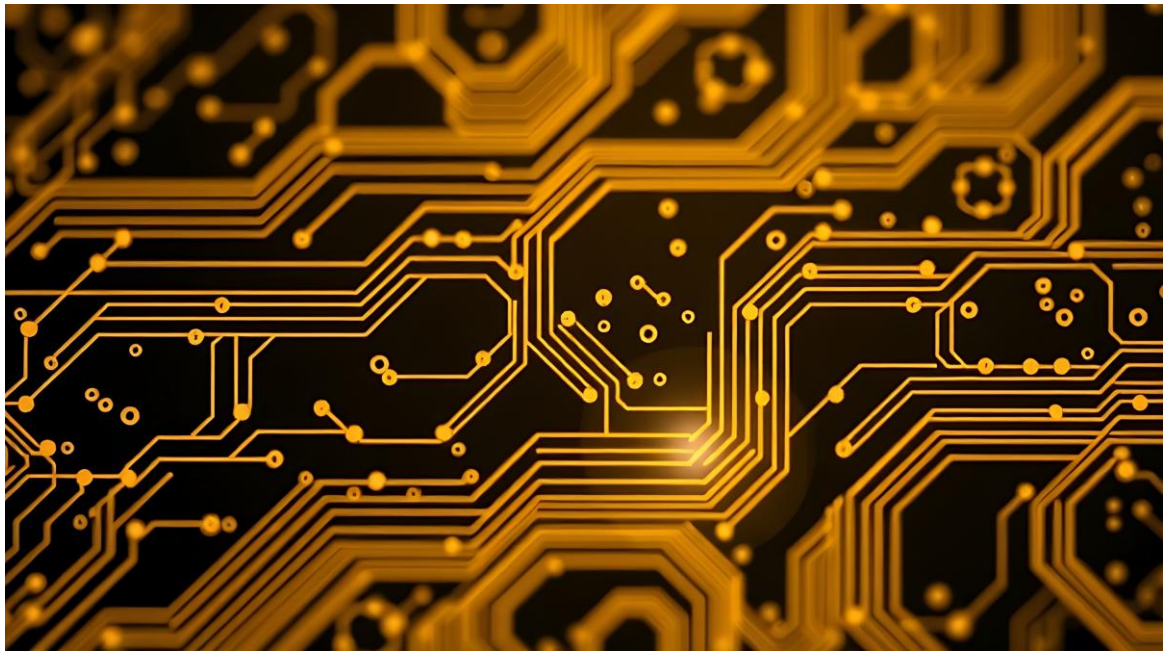
Lab: Packer Golden AMI



Terraform Custom AMI



Terraform Custom AMI



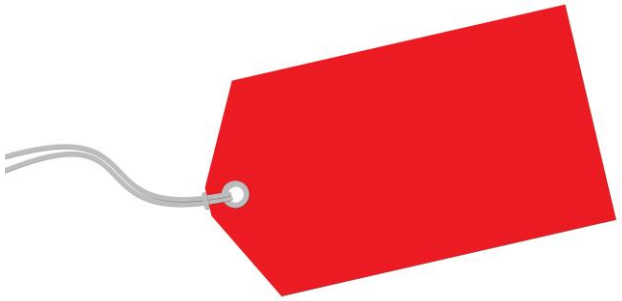
Now that we can build Golden AMIs automatically, we need a way to deploy them into real infrastructure.

Terraform will launch EC2 instances (or an ASG) using the AMI created by your GitHub Actions + Packer pipeline.

Why Terraform?

- Declaratively defines your infrastructure
- Automatically creates the networking, security groups, and Auto Scaling Groups
- Makes deployments consistent and repeatable
- Works perfectly with immutable AMI workflows

Tagging Each Image

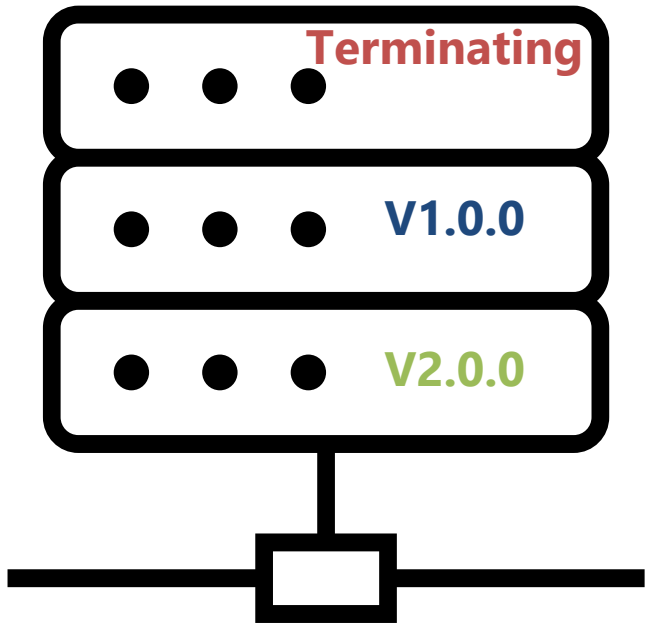


- Packer tags each AMI with a timestamp or version
- Terraform uses a data source filter to look up the most recent AMI
- No need to hardcode an AMI ID
- Every deployment automatically uses the newest Golden AMI

Simply run the terraform pipeline again. If the AMI uses a filter to look for the latest AMI, it will detect that a new AMI is created and update the image.

We can run the Terraform code manually or even integrate it with Hashi Corp cloud with drift detection and automatic deployment. This would achieve **Continuous Deployment.**

Upgrading AMI of ASG



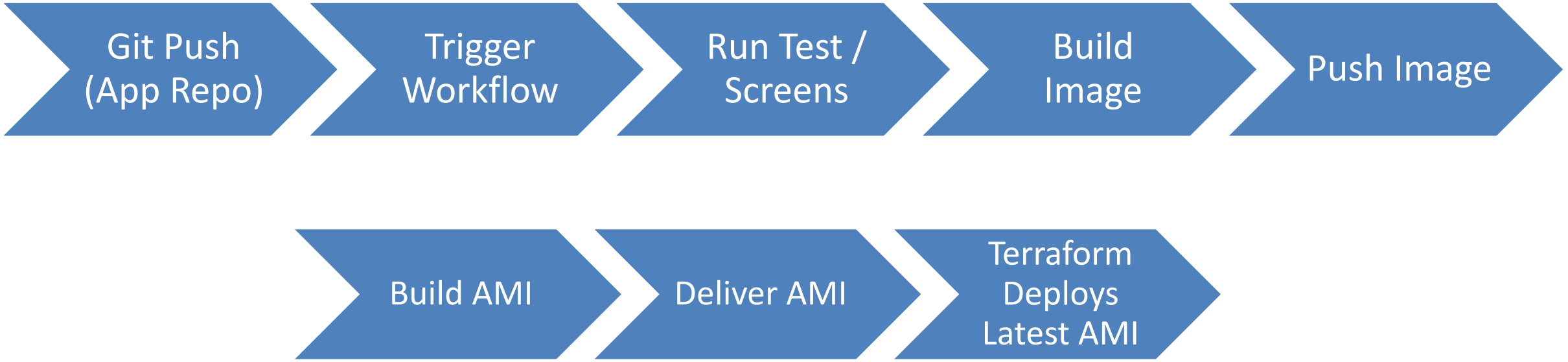
Instead of replacing all instances at once, the ASG gradually brings up new instances and retires old ones, keeping the application available.

How it works:

- ASG detects a new AMI or Launch Template version
- Launches a small number of new instances first
- Waits for them to pass health checks
- Terminates old instances in batches
- Continues until all instances run the new version

You get *zero-downtime* deployments with safe, incremental rollout.

Integration / Delivery / Deployment



Separate Repositories

One that only builds Docker Image

One that builds AMI (uses the Docker Image)

We could have other repositories for deployment

POP QUIZ:

How does Terraform automatically deploy the newest AMI created by Packer?

A. Terraform reads the AMI ID from the env file you provide
B. Terraform uses a data source with filters to select the most recent AMI
C. Terraform triggers a Lambda function to fetch the AMI ID
D. Terraform stores AMI IDs in S3 and downloads them

POP QUIZ:

How does Terraform automatically deploy the newest AMI created by Packer?

A. Terraform reads the AMI ID from the env file you provide
B. **Terraform uses a data source with filters to select the most recent AMI**
C. Terraform triggers a Lambda function to fetch the AMI ID
D. Terraform stores AMI IDs in S3 and downloads them

POP QUIZ:

What enables Continuous Deployment when using Packer + Terraform?

A. Hardcoding the AMI ID in main.tf
B. Using Terraform workspaces
C. Using AMI filters so every Terraform apply uses the newest Golden AMI
D. Running Packer and Terraform in the same workflow file

POP QUIZ:

What enables Continuous Deployment when using Packer + Terraform?

A. Hardcoding the AMI ID in main.tf
B. Using Terraform workspaces
C. **Using AMI filters so every Terraform apply uses the newest Golden AMI**
D. Running Packer and Terraform in the same workflow file

POP QUIZ:

What happens in an Auto Scaling Group when you update the AMI in the launch template?

A. ASG performs a rolling update, replacing instances gradually
B. ASG immediately updates all instances at the same time
C. ASG requires manual intervention to update instances
D. ASG updates only new instances, not existing ones

POP QUIZ:

What happens in an Auto Scaling Group when you update the AMI in the launch template?

A. **ASG performs a rolling update, replacing instances gradually**
B. ASG immediately updates all instances at the same time
C. ASG requires manual intervention to update instances
D. ASG updates only new instances, not existing ones

POP QUIZ:

What is the main benefit of having Terraform reference AMIs dynamically rather than by hardcoding?

A. Faster Terraform plan output
B. Automatic adoption of new Golden AMIs without modifying code
C. Compatibility with ECS
D. Prevents Terraform drift

POP QUIZ:

What is the main benefit of having Terraform reference AMIs dynamically rather than by hardcoding?

A. Faster Terraform plan output
B. **Automatic adoption of new Golden AMIs without modifying code**
C. Compatibility with ECS
D. Prevents Terraform drift

POP QUIZ:

After a new Golden AMI is produced, what must happen for the ASG to actually replace running instances?

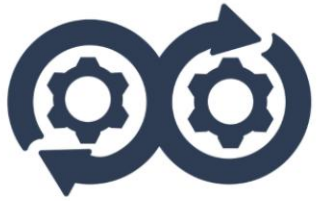
A. Re-running the Terraform apply so the launch template updates
B. Restarting Packer and build a new AMI to launch latest
C. Stopping the ASG manually so that it can update instances
D. Creating a new OIDC provider

POP QUIZ:

After a new Golden AMI is produced, what must happen for the ASG to actually replace running instances?

A. **Re-running the Terraform apply so the launch template updates**
B. Restarting Packer and build a new AMI to launch latest
C. Stopping the ASG manually so that it can update instances
D. Creating a new OIDC provider

Lab: Terraform Custom AMI



Congratulations

Today you learned how to automate infrastructure at a production level:

- How **regions, AZs, ASGs, and load balancers** create resilient architectures
- Why **immutable deployments** are safer and more consistent
- How **Packer builds Golden AMIs** ready to run your application on boot
- How Terraform can **automatically deploy the latest AMIs**
- How ASGs perform **rolling updates** to keep services online

You're now working with the same patterns used in real enterprise CI/CD pipelines.