Software Deployment

> IE University - BCSAI - SDD - 2025

Introduction

Software deployment is the critical process of releasing software to users in a controlled, repeatable, and reliable manner. It encompasses everything from packaging code into deployable artifacts to monitoring applications in production. Modern deployment practices have evolved from manual, error-prone processes to highly automated pipelines that enable teams to release multiple times per day with confidence. This transformation has been driven by the rise of cloud computing, containerization, and infrastructure as code, along with cultural shifts toward continuous integration and continuous deployment (CI/CD). Understanding deployment is essential for building reliable systems that can evolve quickly while maintaining stability and security.

This guide covers the fundamentals of software deployment, exploring automation strategies, deployment pipelines, infrastructure management, and best practices for releasing software to production environments safely and efficiently.

Recommended Resources

Books:

"The Phoenix Project" by Gene Kim, Kevin Behr, and George Spafford tells the story of IT transformation through a compelling narrative that illustrates DevOps principles, including deployment automation and continuous delivery.

"Continuous Delivery" by Jez Humble and David Farley is the definitive guide to building deployment pipelines and releasing software reliably. It covers everything from version control to production monitoring.

"Infrastructure as Code" by Kief Morris provides comprehensive guidance on managing infrastructure through code, covering patterns, practices, and tools for automating infrastructure provisioning and configuration.

Online:

The GitHub Actions documentation (docs.github.com/actions) offers excellent tutorials and examples for building CI/CD pipelines integrated directly with your repositories.

Microsoft's Azure documentation (learn.microsoft.com/azure) provides detailed guides for deploying applications to Azure cloud services, from web apps to containers and serverless functions.

Martin Fowler's articles on deployment patterns (martinfowler.com) explain various deployment strategies including blue-green deployments, canary releases, and feature toggles.

Practice:

The best way to learn deployment is hands-on practice. Set up a simple CI/CD pipeline for a personal project, experiment with different deployment strategies, and use infrastructure as code tools like Terraform or Bicep to provision cloud resources. Contributing to open source projects also exposes you to real-world deployment practices and tooling.

Table of Contents

---

1. Introduction to Software Deployment

Summary

1.1 What deployment is and why it matters
1.2 Manual vs automated deployment
1.3 Goals of modern deployment
1.4 How automation enables those goals

What is Software Deployment?

Deployment is the process of taking software from source code to a running product that users can access. In practice, it bundles code and dependencies (an artifact), prepares the target environment, moves the artifact, configures it, verifies it works, and exposes it to traffic. It's the bridge between writing code and delivering value.

Key activities:

- Packaging: build a deployable unit (container image, ZIP/JAR, binary)
- Environment setup: provision infra, networking, databases, security (often via IaC)
- Install and configure: ship the artifact, set env vars/secrets, run migrations
- Verify and activate: health checks, smoke tests, and routing traffic to the new version

Manual vs Automated Deployment

Manual = humans follow runbooks; Automated = pipelines do repeatable steps. The industry standard is automation because it's faster, safer, and consistent.

Differences at a glance:

- Consistency: manual varies; automated is identical each time
- Speed: manual is slow; automated is minutes
- Errors: manual is typo-prone; automated is reproducible
- Rollback: manual is stressful; automated is scripted and quick
- Auditability: manual leaves gaps; automated logs every step
- Scale: manual hits human limits; automated scales across many environments

Automation requires some upfront investment (pipelines, scripts, secrets), but quickly pays off—even for small teams.

Goals of Modern Deployment

What great deployment looks like:

- Reliability and consistency: same process in every environment, predictable outcomes
- Fewer manual errors: automate repetitive steps and validate early
- Fast iteration: small, frequent changes with rapid feedback
- Safe rollback: easy, rehearsed reversions; backward-compatible data changes
- Transparency: clear version-to-commit mapping, logs, and change history
- Infra parity: environments defined and reproduced with code, minimal drift

The Role of Automation

Treat automation as code: keep pipeline definitions, scripts, and infra in version control with reviews and history.

Core building blocks:

- CI: builds and tests on every change
- CD: packages and deploys to environments with verification
- IaC: declarative infra (Terraform/Bicep/etc.) for parity and reproducibility
- Config management: keep servers/apps in desired state (e.g., Ansible)
- Orchestration: run and update containers reliably (e.g., Kubernetes/ACA)

Typical flow: commit → CI builds/tests → package artifact → deploy to staging → smoke tests → approval (optional) → deploy to prod → health checks/monitoring → rollback if needed.

Benefits: speed, consistency, observability, and higher deploy frequency. Challenges: initial setup, ongoing maintenance, and avoiding “false confidence.” Start small (build + tests), add staging deploys, then automate production with monitoring and rollback.

---

2. The Software Deployment Process

Summary

Typical Deployment Stages

The end-to-end flow, simplified:

1. Source control: Git is the single source of truth; events (push/PR/tag) trigger pipelines.
2. Build: compile/transpile, resolve dependencies, and produce deterministic outputs in clean CI environments.

3. Test: unit/integration/static/security/perf checks; fail fast and report clearly.
4. Package: create immutable deployable artifacts (images/archives/binaries) and store them in an artifact repo.
5. Deploy: provision infra (IaC), install artifact, configure env vars/secrets, run migrations, restart services.
6. Verify: health checks and smoke tests; auto-fail and rollback on errors.
7. Monitor: metrics, logs, alerts, and UX monitoring to catch regressions quickly.

Environments (Development, Staging, Production)

- Development: local or shared; fast feedback; fake/sanitized data; simple config; instability expected.
- Staging (pre-prod): production-like; realistic data and config; final validation for functionality, performance, and integrations.
- Production: hardened, monitored, highly available; real users and data; strict access and auditing; practiced rollback.

Extras: QA, demo, performance, and security testing environments as needed. Aim for parity across environments (OS/runtime/deps/infra/deploy process) using IaC to minimize drift and surprises.

Versioning and Release Management

- Semantic Versioning (MAJOR.MINOR.PATCH + optional pre-release) communicates change impact.
- Git tags identify exact release commits; dashboards and build metadata show what runs where.
- Branching: trunk-based (simple, flags for unfinished work), GitHub Flow (PRs into always-deployable main), or Git Flow (heavier, clear release/hotfix separation).
- Cadence: continuous (every green build), scheduled (weekly/monthly), feature-based, plus hotfixes for urgent issues. Smaller, frequent releases reduce risk and speed feedback.
- Rollback: know current/previous versions and DB compatibility; keep changelogs/releases notes focused and actionable.

Artifacts and Packages

Artifacts are immutable, versioned bundles you deploy (“build once, deploy many”). Good artifacts are self-contained, portable, reproducible, and traceable (commit SHA, build time, test results).

Common types:

- Container images (Docker) for consistent runtime and easy orchestration
- Archives (ZIP/JAR/WAR/TAR) for traditional/serverless packaging
- Platform-specific packages (wheels, gems, NuGet, npm)
- Cloud formats (e.g., Functions/Lambda zips or images)
- Compiled binaries (self-contained when needed)

Best practices:

- Store in a registry/repository with access controls, retention, vulnerability scanning, and audit logs
- Tag artifacts with commit SHAs and semantic versions
- Keep secrets out of artifacts; inject via environment/secret stores
- Optimize size (multi-stage builds, exclude dev deps) for faster deploys

3. Continuous Integration / Continuous Deployment (CI/CD)

What is Continuous Integration?

Continuous Integration (CI) is the practice of merging changes to the main branch frequently and validating every change with an automated build and test run. CI turns integration from a painful, infrequent event into a routine activity with fast feedback.

Key elements:

- Frequent commits and small pull requests
- Automated build in a clean environment on every push/PR
- Automated tests (unit first, then integration where feasible)
- Static checks: linting, formatting, type checks, SCA (dependency vulnerabilities)
- Fast feedback (aim for <10 minutes for core checks)

Success criteria:

- Broken builds are rare and fixed immediately
- Test signal is trusted (low flakiness)
- Artifacts are created once and reused downstream

What is Continuous Deployment?

Continuous Deployment (CD) extends CI by automatically promoting a green build through environments up to production. Two common modes:

- Continuous Delivery: automated to staging; production requires a human approval
- Continuous Deployment: fully automated to production when all checks pass

Prerequisites and enablers:

- Robust automated tests and health checks
- Environment-specific configs and secrets management
- Progressive delivery (canary, blue-green, feature flags) to reduce risk
- Clear rollback procedures and monitoring/alerts

CI/CD Pipeline Architecture

Common building blocks:

- Triggers: push, pull_request, tag/release, schedule
- Runners: hosted or self-hosted agents executing jobs
- Jobs/stages: build → test → package → scan → deploy → verify
- Artifacts/caches: share build outputs efficiently between jobs
- Environments: dev → staging → prod with promotion and approvals
- Secrets: injected at runtime via secure stores (never in code)
- Policies: concurrency limits, branch protections, required checks

Example (GitHub Actions, simplified):

```yaml
name: ci
on: [push, pull_request]
jobs:
   build-test:
      runs-on: ubuntu-latest
      steps:
         - uses: actions/checkout@v4
         - uses: actions/setup-node@v4
           with: { node-version: '18' }
         - run: npm ci
         - run: npm test --silent
         - run: npm run build
         - name: Save artifact
           uses: actions/upload-artifact@v4
           with: { name: web-build, path: dist }
```

Promotion to staging/production typically consumes the saved artifact, applies env-specific configuration, runs smoke checks, and gates rollout via approvals or automated policies.

Benefits and Trade-offs

Benefits:

- Faster feedback and delivery; smaller, safer releases
- Higher quality via consistent, repeatable automation
- Improved observability and auditability of changes
- Reduced manual toil; better developer focus and morale

Trade-offs (and mitigations):

- Upfront investment and maintenance (start small; iterate)
- Flaky tests slow teams (quarantine, deflake, stabilize; run critical tests first)
- Long pipelines (parallelize, cache, split paths by change scope)
- Secret management complexity (use managed secret stores and short-lived tokens)
- Risk of “green but wrong” (add smoke tests, synthetic checks, canaries, real-time alerts)

---

4. GitHub Actions for Deployment Automation

GitHub Actions Overview

GitHub Actions is a native CI/CD platform integrated directly into GitHub repositories. It automates workflows triggered by repository events (push, PR, release) using YAML definitions stored in `.github/workflows/`. It provides hosted runners (Linux/Windows/macOS), secrets management, a marketplace of reusable actions, and tight integration with GitHub's ecosystem (pull requests, issues, releases, environments).

Why use it:

- No separate CI/CD service to configure or maintain

- Built-in authentication and permissions via GitHub tokens
- Rich ecosystem of community and official actions
- Free tier for public repos; generous minutes for private repos
- Native support for GitHub features (environments, deployments, status checks)

Core Concepts (Workflows, Jobs, Steps, Runners)

Workflows: YAML files in `.github/workflows/` defining automation. Each workflow has triggers, jobs, and metadata.

Jobs: Independent units of work that run in parallel by default. Each job runs on a fresh runner (VM). Use `needs:` to create dependencies between jobs.

Steps: Sequential commands or actions within a job. Steps share the same runner and filesystem.

Runners: Virtual machines (GitHub-hosted or self-hosted) executing jobs. GitHub-hosted runners come with common tools pre-installed.

Actions: Reusable units (from Marketplace or custom) that encapsulate logic—e.g., `actions/checkout@v4` clones your repo, `actions/setup-node@v4` installs Node.js.

Structure at a glance:

```
Workflow
 ├─ Job 1 (runs on runner-1)
 │   ├─ Step 1
 │   ├─ Step 2
 │   └─ Step 3
 └─ Job 2 (runs on runner-2, may depend on Job 1)
     ├─ Step 1
     └─ Step 2
```

Workflow Triggers and Events

Common triggers:

- `push`: on commits to specified branches
- `pull_request`: on PR open/update/merge
- `workflow_dispatch`: manual trigger via UI or API
- `release`: when a release is published
- `schedule`: cron-based (e.g., nightly builds)
- `repository_dispatch`: external webhook trigger

Example trigger configuration:

```yaml
on:
  push:
    branches: [main, develop]
  pull_request:
```

```
    branches: [main]
  workflow_dispatch:  # manual trigger
```

Use path filters to run workflows only when relevant files change:

```
on:
  push:
    paths:
      - 'src/**'
      - 'package.json'
```

Building a Deployment Workflow

A typical deployment workflow:

1. Checkout code
2. Set up language runtime
3. Install dependencies
4. Run tests
5. Build artifact
6. Deploy to target environment
7. Verify deployment

Example deploying to Azure Web App:

```
name: Deploy to Azure
on:
  push:
    branches: [main]
jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - uses: actions/setup-node@v4
        with:
          node-version: '18'
          cache: 'npm'

      - run: npm ci
      - run: npm test
      - run: npm run build

      - name: Deploy to Azure Web App
        uses: azure/webapps-deploy@v2
        with:
          app-name: my-webapp
```

```
          publish-profile: ${{ secrets.AZURE_WEBAPP_PUBLISH_PROFILE }}
          package: ./dist
```

Best practices:

- Use caching (`cache: 'npm'`) to speed up installs
- Fail fast: run quick tests before slow builds
- Use matrix strategies for multi-platform/version testing
- Separate build and deploy jobs for clarity and artifact reuse
- Use environments for deployment approvals and protection rules

Secrets and Environment Variables

Secrets: encrypted values stored in repository/organization settings. Access via `${{ secrets.SECRET_NAME }}`. Never log or expose secrets in output.

Adding secrets:

- Repo → Settings → Secrets and variables → Actions → New repository secret

Common secrets:

- `AZURE_CREDENTIALS` or `AZURE_WEBAPP_PUBLISH_PROFILE`
- `AWS_ACCESS_KEY_ID` / `AWS_SECRET_ACCESS_KEY`
- `NPM_TOKEN` for private registry access
- Database connection strings
- API keys for third-party services

Environment variables: set at workflow, job, or step level using `env:`.

```
env:
  NODE_ENV: production
  API_URL: https://api.example.com
jobs:
  deploy:
    env:
      DEPLOY_REGION: us-west-2
    steps:
      - run: echo "Deploying to $DEPLOY_REGION"
```

Environments: GitHub feature for deployment targeting (production, staging) with protection rules (required reviewers, wait timers, branch restrictions). Configure in Repo → Settings → Environments.

```
jobs:
  deploy:
    environment: production  # requires approval if configured
    steps:
```

```
      - name: Deploy
        run: ./deploy.sh
```

GitHub Actions Marketplace

The Marketplace (github.com/marketplace?type=actions) offers thousands of reusable actions for common tasks:

Popular actions:

- `actions/checkout@v4`: clone your repository
- `actions/setup-node@v4`, `actions/setup-python@v4`: install runtimes
- `actions/cache@v4`: cache dependencies
- `actions/upload-artifact@v4` / `download-artifact@v4`: share build outputs
- `azure/webapps-deploy@v2`: deploy to Azure App Service
- `docker/build-push-action@v5`: build and push Docker images
- `aws-actions/configure-aws-credentials@v4`: authenticate with AWS

Using an action:

```
- uses: docker/build-push-action@v5
  with:
    context: .
    push: true
    tags: myregistry/myapp:latest
```

Creating custom actions:

- JavaScript/TypeScript actions (fastest startup)
- Docker container actions (custom environment)
- Composite actions (combine multiple steps into reusable unit)

Store custom actions in `.github/actions/` or separate repos and reference via `owner/repo@ref`.

---

5. Deploying to Azure

Azure Deployment Targets

Azure offers multiple compute options for different application types:

Azure App Service (Web Apps): Fully managed PaaS for web apps and APIs. Best for traditional web applications, REST APIs, and backends. Supports .NET, Java, Node.js, Python, PHP. Auto-scaling, deployment slots, built-in CI/CD.

Azure Container Apps: Serverless container hosting with auto-scaling to zero. Best for microservices, event-driven apps, and containerized workloads. Built on Kubernetes but abstracts cluster management.

Azure Kubernetes Service (AKS): Managed Kubernetes for full container orchestration control. Best for complex microservices, multi-tenant apps, or when you need Kubernetes features.

Azure Functions: Serverless compute triggered by events. Best for event-driven tasks, scheduled jobs, and API endpoints with sporadic traffic. Pay-per-execution model.

Azure Static Web Apps: Hosting for static sites with integrated API support via Functions. Best for SPAs (React/Vue/Angular), JAMstack sites, and portfolios.

Azure Container Instances (ACI): Simple container hosting without orchestration. Best for batch jobs, CI/CD agents, and simple containerized tasks.

Choose based on:

- Control needed (Functions < Web Apps < Container Apps < AKS)
- Scaling requirements (serverless vs always-on)
- Cost model (consumption vs reserved instances)
- Technology stack and containerization

Azure Web Apps

Web Apps provide a straightforward PaaS for deploying web applications without managing infrastructure.

Key features:

- Automatic OS patching and runtime updates
- Built-in load balancing and auto-scaling
- Deployment slots for blue-green deployments
- Integrated monitoring via Application Insights
- Custom domains and free SSL certificates

Deployment methods:

1. **Publish Profile**: Download from portal, use in GitHub Actions
2. **Azure CLI**: `az webapp up` or `az webapp deployment source config-zip`
3. **GitHub Actions**: Native integration via `azure/webapps-deploy` action
4. **Local Git**: Push to Azure Git endpoint
5. **ZIP deploy**: Upload ZIP via API or CLI

GitHub Actions example:

```
- name: Deploy to Azure Web App
  uses: azure/webapps-deploy@v2
  with:
    app-name: 'my-webapp'
    publish-profile: ${{ secrets.AZURE_WEBAPP_PUBLISH_PROFILE }}
    package: ./build
```

Deployment slots: Create staging slots for zero-downtime deployments:

```
az webapp deployment slot create --name my-webapp --resource-group rg --
slot staging
# Deploy to staging, test, then swap
az webapp deployment slot swap --name my-webapp --resource-group rg --slot
staging
```

Configuration:

- App Settings: environment variables accessible to your app
- Connection Strings: for databases (automatically encrypted)
- Configure via Portal, CLI, or ARM templates

Container Deployments (Azure Container Apps)

Container Apps run containerized workloads with automatic scaling, traffic splitting, and revision management—without managing Kubernetes.

Key features:

- Scale to zero (pay only when running)
- Traffic splitting for A/B testing and canary releases
- Dapr integration for microservices patterns
- Ingress with automatic HTTPS
- Container-native secrets and environment variables

Deployment workflow:

1. Build and push container image to registry (ACR, Docker Hub, GitHub Container Registry)
2. Create or update Container App with new image
3. Container Apps pulls image and creates new revision
4. Traffic routes to new revision (with optional gradual rollout)

GitHub Actions example:

```
- name: Build and push image
  run: |
    docker build -t myregistry.azurecr.io/myapp:${{ github.sha }} .
    docker push myregistry.azurecr.io/myapp:${{ github.sha }}

- name: Deploy to Container Apps
  uses: azure/container-apps-deploy-action@v1
  with:
    containerAppName: my-container-app
    resourceGroup: rg
    imageToDeploy: myregistry.azurecr.io/myapp:${{ github.sha }}
```

Azure CLI deployment:

```
az containerapp update \
  --name my-app \
  --resource-group rg \
  --image myregistry.azurecr.io/myapp:latest
```

Traffic splitting (canary deployment):

```
az containerapp ingress traffic set \
  --name my-app \
  --resource-group rg \
  --revision-weight latest=20 previous=80
```

Azure Functions (Serverless)

Functions execute code in response to triggers (HTTP, timer, queue, blob) without managing servers. Pay only for execution time.

Common triggers:

- HTTP: REST APIs and webhooks
- Timer: scheduled tasks (cron)
- Queue/Service Bus: async message processing
- Blob/Cosmos DB: data change events

Deployment methods:

1. **VS Code Extension**: right-click and deploy
2. **Azure Functions Core Tools**: `func azure functionapp publish <app-name>`
3. **GitHub Actions**: `azure/functions-action`
4. **ZIP deploy**: package and upload

GitHub Actions example:

```
- name: Deploy to Azure Functions
  uses: Azure/functions-action@v1
  with:
    app-name: my-function-app
    package: ./output
    publish-profile: ${{ secrets.AZURE_FUNCTIONAPP_PUBLISH_PROFILE }}
```

Best practices:

- Keep functions small and focused (single responsibility)
- Use durable functions for workflows and orchestrations
- Monitor with Application Insights
- Set appropriate timeout values

- Use Premium or Dedicated plans for production workloads requiring VNet integration or faster cold starts

Publish Profiles and Service Principals

Two primary authentication methods for Azure deployments:

Publish Profiles (simpler, less secure):

- XML file with deployment credentials
- Downloaded from Azure Portal (App Service → Deployment Center → Manage publish profile)
- Store in GitHub Secrets as `AZURE_WEBAPP_PUBLISH_PROFILE`
- Scoped to single resource
- Easier for beginners but harder to rotate and audit

Service Principals (recommended for production):

- Azure AD identity with specific permissions (RBAC)
- Can access multiple resources
- Supports federated credentials (OIDC) for keyless authentication
- Easier to audit and rotate

Creating a Service Principal:

```
az ad sp create-for-rbac \
  --name "github-actions-sp" \
  --role contributor \
  --scopes /subscriptions/{subscription-id}/resourceGroups/{resource-group}
\
  --sdk-auth
```

Output JSON goes into GitHub Secret `AZURE_CREDENTIALS`.

Using Service Principal in GitHub Actions:

```
- name: Azure Login
  uses: azure/login@v1
  with:
    creds: ${{ secrets.AZURE_CREDENTIALS }}

- name: Deploy
  run: az webapp up --name my-webapp --resource-group rg
```

Federated Credentials (OIDC, no secrets): Configure trust between GitHub and Azure AD, then:

```
- name: Azure Login
  uses: azure/login@v1
  with:
```

```
      client-id: ${{ secrets.AZURE_CLIENT_ID }}
      tenant-id: ${{ secrets.AZURE_TENANT_ID }}
      subscription-id: ${{ secrets.AZURE_SUBSCRIPTION_ID }}
```

This approach eliminates secret storage entirely—Azure validates GitHub's OIDC token directly.

Choosing an approach:

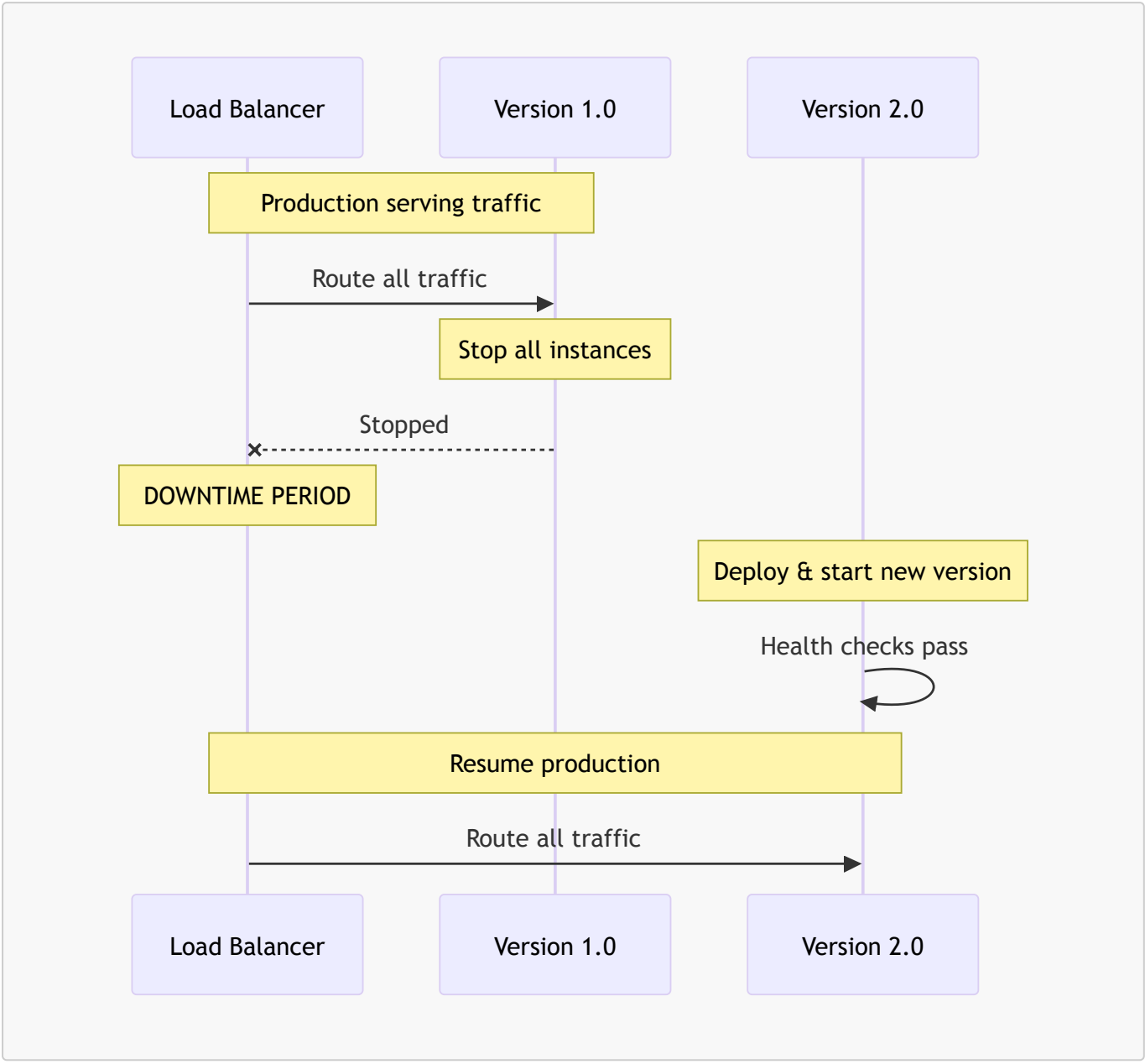
- **Learning/demos**: Publish profiles
- **Production**: Service principals with OIDC
- **Multiple resources**: Service principals with appropriate RBAC scopes

---

6. Deployment Strategies

Recreate Deployment

The simplest strategy: stop the old version, deploy the new one, start it up.



Process:

1. Stop all instances of the current version
2. Deploy new version
3. Start new instances
4. Verify health

Pros:

- Simple to implement and understand
- Clean state (no mixed versions)
- No extra infrastructure needed

Cons:

- Downtime during deployment (seconds to minutes)
- All-or-nothing: if new version fails, must rollback manually
- Not suitable for high-availability services

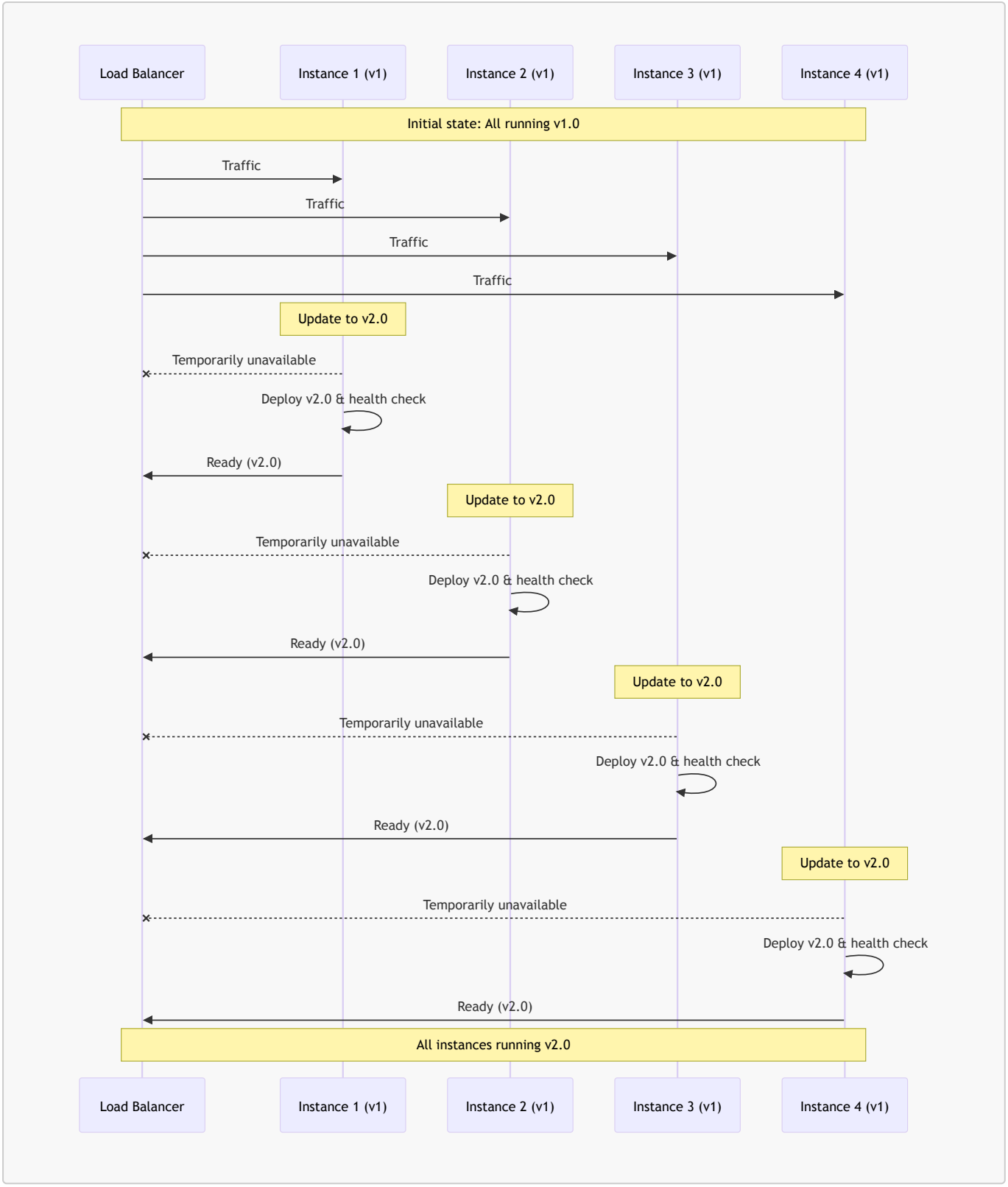
When to use: Internal tools, development environments, maintenance windows, or applications where brief downtime is acceptable.

Example (Docker Compose):

```
docker-compose down
docker-compose pull
docker-compose up -d
```

Rolling Update

Gradually replace instances with new version, maintaining some capacity throughout.



Process:

1. Deploy new version to subset of instances (e.g., 25%)
2. Wait for health checks to pass
3. Route traffic to new instances
4. Repeat for remaining instances in waves
5. Old version is phased out gradually

Pros:

- Zero or minimal downtime
- Gradual rollout reduces blast radius
- Can pause or rollback mid-deployment
- No extra infrastructure cost

Cons:

- Mixed versions running simultaneously (compatibility required)
- Slower than recreate
- Requires load balancer or orchestrator
- Rollback is slower (must roll back each wave)

When to use: Production services requiring high availability, when you have multiple instances behind a load balancer.

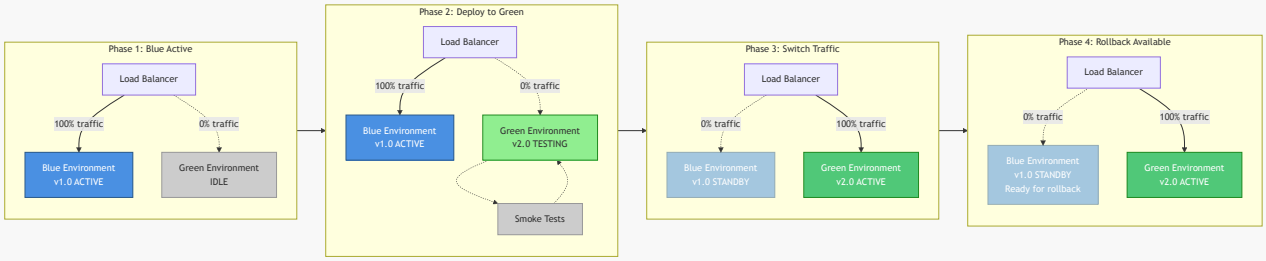
Example (Kubernetes):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 4
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1  # max instances down during update
      maxSurge: 1        # max extra instances during update
```

Kubernetes updates 1-2 pods at a time, ensuring at least 3 are always available.

Blue-Green Deployment

Run two identical environments (blue = current, green = new). Switch traffic instantly once green is validated.



Process:

1. Blue environment serves production traffic
2. Deploy new version to green environment

3. Test green thoroughly (smoke tests, validation)
4. Switch traffic from blue to green (via load balancer or DNS)
5. Monitor green; keep blue ready for instant rollback
6. After confidence period, decommission blue (or make it the next "green")

Pros:

- Zero downtime
- Instant rollback (switch back to blue)
- Full testing in production-like environment before traffic switch
- Clean separation of versions

Cons:

- Requires double infrastructure (2x cost during deployment)
- Database migrations tricky (must be backward-compatible)
- More complex to set up

When to use: Critical production services, when instant rollback is essential, when you can afford double infrastructure temporarily.

Implementation:

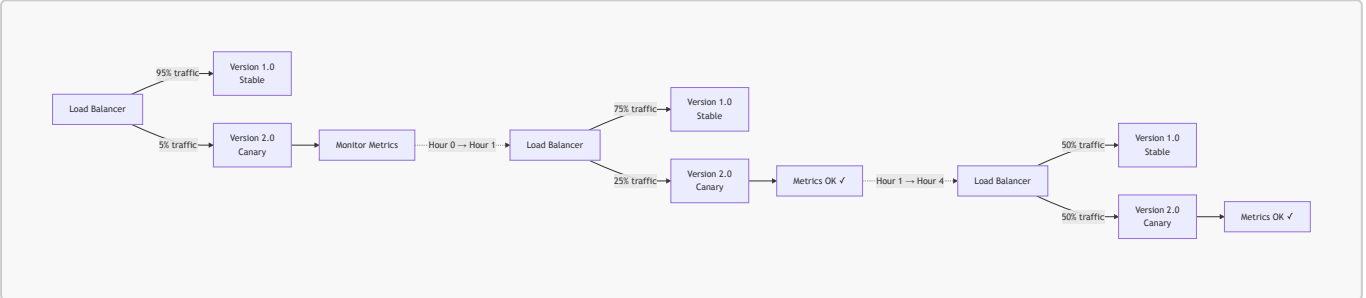
- Azure App Service: deployment slots with swap
- AWS: Elastic Beanstalk environments or Route 53 weighted routing
- Load balancers: switch target groups/backends

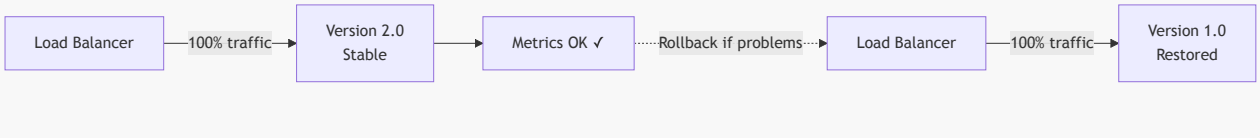
Example (Azure Web App slots):

```
# Deploy to staging slot
az webapp deployment source config-zip \
  --resource-group rg --name my-app --slot staging --src app.zip

# Test staging

# Swap staging → production
az webapp deployment slot swap \
  --resource-group rg --name my-app --slot staging
```

Canary Release

Release new version to a small subset of users first, gradually increasing traffic if metrics look good.





Process:

1. Deploy new version alongside old
2. Route small % of traffic to new version (e.g., 5%)
3. Monitor metrics (errors, latency, user behavior)
4. If healthy, gradually increase traffic (10% → 25% → 50% → 100%)
5. If issues detected, rollback immediately
6. Retire old version once canary serves 100%

Pros:

- Minimal blast radius (issues affect few users initially)
- Data-driven rollout based on real metrics
- Early detection of production issues
- Can target specific user segments

Cons:

- Requires sophisticated traffic routing
- Needs robust monitoring and automated decision-making
- Mixed versions in production
- More complex rollback scenarios

When to use: High-traffic applications, when you want data-driven deployment decisions, when partial failures are acceptable.

Traffic progression example:

```
Hour 0: 5% canary, 95% stable   (monitor closely)
Hour 1: 10% canary, 90% stable  (if metrics OK)
Hour 2: 25% canary, 75% stable
Hour 4: 50% canary, 50% stable
Hour 8: 100% canary             (full rollout)
```

Implementation:

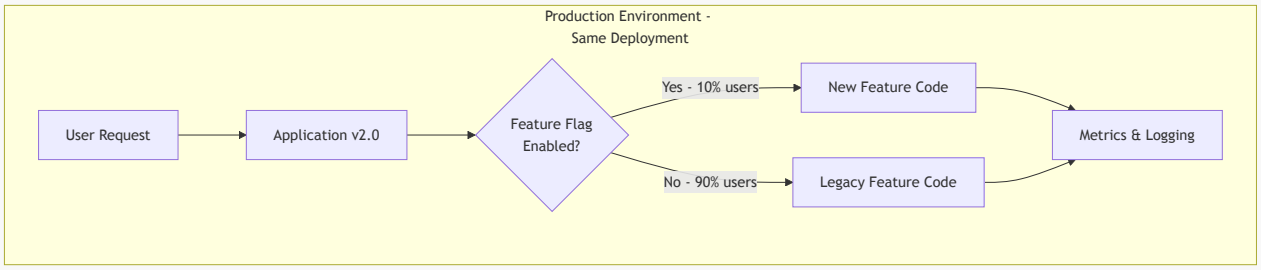
- Kubernetes: Flagger, Argo Rollouts
- Service mesh: Istio, Linkerd
- Azure Container Apps: traffic splitting
- Feature flags + server-side logic

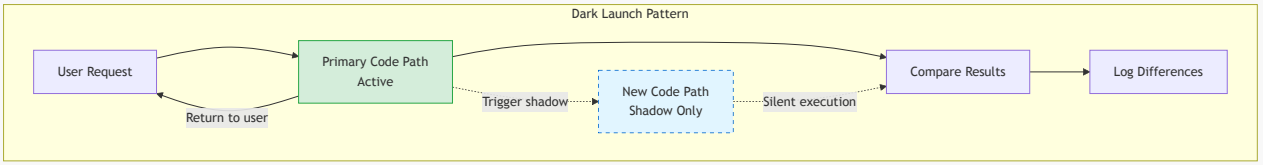
Example (Azure Container Apps):

```
# Route 10% to latest revision, 90% to previous
az containerapp ingress traffic set \
  --name my-app --resource-group rg \
  --revision-weight latest=10 previous=90
```

Feature Flags and Dark Launches

Control feature visibility at runtime without redeploying. Related to but distinct from deployment strategies.





Feature Flags (Feature Toggles): Conditional code that enables/disables features dynamically.

Use cases:

- Deploy code before feature is ready (dark launch)
- A/B testing different implementations
- Emergency kill switch for problematic features
- Gradual rollout to user segments (beta users, regions)
- Ops toggles for performance tuning

Example:

```
if (featureFlags.isEnabled('new-checkout')) {
  return <NewCheckoutFlow />;
} else {
  return <LegacyCheckoutFlow />;
}
```

Best practices:

- Keep flags short-lived (remove after full rollout)
- Use a flag management system (LaunchDarkly, Unleash, ConfigCat)
- Don't nest flags deeply (complexity)
- Document flag purpose and owner
- Clean up old flags regularly

Dark Launch: Deploy new code in production but don't expose to users yet. Use flags to enable only for internal users or test traffic.

Benefits:

- Deploy and test in production without user impact
- Decouple deployment from release
- Validate production performance before release
- Mimic real load with shadow traffic

Example:

```
def process_order(order):
    # Old code path (active)
    result = legacy_process(order)

    # New code path (dark launch, metrics only)
    if feature_flags.is_enabled('new-processor', user='internal'):
        new_result = new_process(order)
        log_comparison(result, new_result)  # compare but don't use

    return result
```

Choosing the Right Strategy

Selection criteria:

Strategy	Downtime	Cost	Complexity	Rollback Speed	Best For
Recreate	Yes	Low	Low	Slow	Dev/test, maintenance windows
Rolling	No	Low	Medium	Medium	Standard production apps
Blue-Green	No	High	Medium	Instant	Mission-critical, instant rollback needed
Canary	No	Medium	High	Fast	High-traffic, data-driven decisions

Decision guide:

- **Learning/prototyping**: Recreate
- **Standard web app**: Rolling update

- **Can't afford any downtime**: Blue-green
- **Large user base, risk-averse**: Canary
- **Need A/B testing**: Feature flags + canary
- **Database changes involved**: Blue-green or rolling with backward-compatible migrations

Combining strategies:

- Blue-green for infra + feature flags for features
- Rolling update + canary traffic routing
- Feature flags for fast rollback instead of redeployment

The trend is toward more sophisticated strategies (canary + flags) as tooling matures, but simple strategies still have their place based on application requirements and team maturity.

---

7. Infrastructure as Code (IaC)

What is Infrastructure as Code?

Managing and provisioning infrastructure through machine-readable definition files rather than manual processes or interactive configuration tools.

Core concept: Treat infrastructure like software code—version it, review it, test it, and deploy it automatically.

Traditional approach (manual):

1. Log into cloud portal
2. Click through UI to create resources
3. Configure settings manually
4. Document steps (often forgotten)
5. Repeat manually for different environments

IaC approach:

1. Write infrastructure definition in code
2. Commit to version control
3. Run automated tool to provision/update
4. Same code creates dev, staging, production
5. Changes tracked in git history

Example: Instead of clicking in Azure Portal to create a web app, write:

```
resource "azurerm_app_service" "example" {
  name                = "my-app"
  location            = "eastus"
  resource_group_name = azurerm_resource_group.example.name
  app_service_plan_id = azurerm_app_service_plan.example.id
}
```

Run `terraform apply` → infrastructure created automatically.

Benefits of IaC

1. Repeatability and consistency

- Same code produces identical infrastructure every time
- No configuration drift from manual changes
- Environments (dev/staging/prod) are consistent

2. Version control

- Infrastructure changes tracked in git
- Review changes via pull requests
- Rollback to previous infrastructure state
- Audit trail of who changed what and when

3. Speed and efficiency

- Provision entire environments in minutes
- Automate repetitive tasks
- Scale resources programmatically
- Parallel resource creation

4. Documentation as code

- Infrastructure definition IS the documentation
- Always up-to-date (unlike docs)
- Onboarding: read the code to understand architecture

5. Disaster recovery

- Recreate entire infrastructure from code
- No manual reconstruction guesswork
- Test disaster recovery regularly (spin up/down)

6. Collaboration and team knowledge

- Infrastructure changes go through code review
- Team members understand infrastructure
- No "only one person knows how it works"

7. Cost management

- Spin down non-production environments after hours
- Template smaller/larger resource sizes per environment
- Track infrastructure costs via code changes

Trade-offs:

- Learning curve for IaC tools
- Initial setup time investment

- Need discipline to avoid manual changes (always use code)

IaC Tools Overview

Terraform (HashiCorp):

- **Multi-cloud**: AWS, Azure, GCP, and 1000+ providers
- **Declarative**: describe desired state, Terraform figures out how
- **HCL language**: HashiCorp Configuration Language
- **State management**: tracks what's deployed
- **Plan before apply**: preview changes
- **Industry standard** for multi-cloud

Bicep (Microsoft):

- **Azure-specific**: deeply integrated with Azure
- **Declarative**: ARM template alternative with cleaner syntax
- **Transpiles to ARM**: compiles to ARM JSON templates
- **Native Azure support**: day-0 support for new Azure features
- **VS Code extension**: excellent autocomplete, validation
- **Best for Azure-only** shops

Pulumi:

- **Use real programming languages**: TypeScript, Python, Go, C#
- **Imperative + declarative**: familiar programming constructs
- **Multi-cloud**: like Terraform
- **Good for**: teams preferring general-purpose languages

AWS CloudFormation / Azure Resource Manager (ARM):

- **Cloud-native**: built into cloud platform
- **JSON/YAML templates**: verbose but comprehensive
- **Free**: no extra tooling cost
- **Good for**: single-cloud, simple use cases

Ansible, Chef, Puppet:

- **Configuration management**: primarily for server configuration
- **Can do IaC**: but less common for cloud resources
- **Procedural**: describe steps, not desired state

Comparison:

Tool	Multi-cloud	Language	Learning Curve	Best For
Terraform	YES	HCL	Medium	Multi-cloud, industry standard

Tool	Multi-cloud	Language	Learning Curve	Best For
Bicep	NO (Azure only)	Bicep	Low-Medium	Azure-specific projects
Pulumi	YES	TypeScript/Python/Go	Medium	Devs wanting familiar languages
ARM/CloudFormation	NO	JSON/YAML	Medium-High	Cloud-native, simple setups

Terraform Fundamentals

Core workflow:

1. **Write**: Define infrastructure in `.tf` files
2. **Init**: `terraform init` (download providers)
3. **Plan**: `terraform plan` (preview changes)
4. **Apply**: `terraform apply` (create/update resources)
5. **Destroy**: `terraform destroy` (tear down)

Key concepts:

- **Providers**: plugins for cloud platforms (azurerm, aws, google)
- **Resources**: infrastructure components (VMs, databases, networks)
- **State**: Terraform's record of what's deployed (stored in `terraform.tfstate`)
- **Variables**: parameterize configurations
- **Outputs**: export values for other tools or display

Example (Azure Web App):

```
# Configure provider
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~> 3.0"
    }
  }
}

provider "azurerm" {
  features {}
}

# Resource group
resource "azurerm_resource_group" "example" {
  name     = "rg-myapp-prod"
  location = "East US"
}
```

```
# App Service Plan
resource "azurerm_service_plan" "example" {
  name                = "asp-myapp-prod"
  resource_group_name = azurerm_resource_group.example.name
  location            = azurerm_resource_group.example.location
  os_type             = "Linux"
  sku_name            = "B1"
}

# Web App
resource "azurerm_linux_web_app" "example" {
  name                = "app-myapp-prod"
  resource_group_name = azurerm_resource_group.example.name
  location            = azurerm_resource_group.example.location
  service_plan_id     = azurerm_service_plan.example.id

  site_config {
    application_stack {
      node_version = "18-lts"
    }
  }
}

# Output the URL
output "app_url" {
  value = "https://${azurerm_linux_web_app.example.default_hostname}"
}
```

Commands:

```
terraform init      # Download azurerm provider
terraform plan      # Show what will be created
terraform apply     # Create resources (prompts for confirmation)
terraform destroy   # Delete all resources
```

State management:

- Local state: `terraform.tfstate` file (not for teams)
- Remote state: Azure Storage, AWS S3, Terraform Cloud (for teams)
- Locks prevent concurrent modifications

Variables for environments:

```
variable "environment" {
  description = "Environment name"
  type        = string
  default     = "dev"
}
```

```
resource "azurerm_resource_group" "example" {
  name     = "rg-myapp-${var.environment}"
  location = "East US"
}
```

Use: `terraform apply -var="environment=prod"`

Azure ARM Templates and Bicep

Microsoft's domain-specific language for deploying Azure resources. Cleaner alternative to ARM templates.

Core workflow:

1. **Write**: Define infrastructure in `.bicep` files
2. **Build**: `az bicep build` (compile to ARM JSON, optional)
3. **Deploy**: `az deployment group create` (provision resources)
4. **Validate**: `az deployment group validate` (check before deploying)

Example (Same Azure Web App):

```
// Parameters
param location string = 'eastus'
param appName string = 'myapp'
param environment string = 'prod'
param sku string = 'B1'

// Variables
var resourceGroupName = 'rg-${appName}-${environment}'
var appServicePlanName = 'asp-${appName}-${environment}'
var webAppName = 'app-${appName}-${environment}'

// Resource Group (if deploying at subscription level)
// For resource group deployment, RG already exists

// App Service Plan
resource appServicePlan 'Microsoft.Web/serverfarms@2022-03-01' = {
  name: appServicePlanName
  location: location
  sku: {
    name: sku
  }
  kind: 'linux'
  properties: {
    reserved: true  // required for Linux
  }
}

// Web App
resource webApp 'Microsoft.Web/sites@2022-03-01' = {
```

```
  name: webAppName
  location: location
  properties: {
    serverFarmId: appServicePlan.id
    siteConfig: {
      linuxFxVersion: 'NODE|18-lts'
      alwaysOn: true
    }
  }
}

// Output
output appUrl string = 'https://${webApp.properties.defaultHostName}'
```

Deploy:

```
# Create resource group
az group create --name rg-myapp-prod --location eastus

# Deploy Bicep file
az deployment group create \
  --resource-group rg-myapp-prod \
  --template-file main.bicep \
  --parameters environment=prod appName=myapp

# What-if (preview changes)
az deployment group what-if \
  --resource-group rg-myapp-prod \
  --template-file main.bicep
```

Bicep advantages:

- Simpler syntax than ARM JSON
- Better IDE support (VS Code extension is excellent)
- Type safety and validation
- Day-0 support for new Azure features
- Modules for reusable components

Modules (reusable components):

```
// webapp-module.bicep
param name string
param location string
// ... define web app resource

// main.bicep
module webApp 'webapp-module.bicep' = {
  name: 'webAppDeployment'
  params: {
```

```
    name: 'myapp'
    location: 'eastus'
  }
}
```

Integrating IaC with CI/CD

Integrate infrastructure provisioning with application deployment for full automation.

Workflow:

1. Code change triggers CI/CD pipeline
2. Pipeline provisions/updates infrastructure (IaC)
3. Pipeline deploys application to infrastructure
4. Automated tests verify deployment
5. Promote to next environment or rollback

Benefits:

- Infrastructure and app deployed together
- Infrastructure changes reviewed like code
- Consistent environments across pipeline
- No manual infrastructure setup

GitHub Actions example (Terraform):

```yaml
name: Deploy Infrastructure and App

on:
  push:
    branches: [main]

env:
  ARM_CLIENT_ID: ${{ secrets.AZURE_CLIENT_ID }}
  ARM_CLIENT_SECRET: ${{ secrets.AZURE_CLIENT_SECRET }}
  ARM_SUBSCRIPTION_ID: ${{ secrets.AZURE_SUBSCRIPTION_ID }}
  ARM_TENANT_ID: ${{ secrets.AZURE_TENANT_ID }}

jobs:
  infrastructure:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v3
        with:
          terraform_version: 1.6.0

      - name: Terraform Init
        run: terraform init
```

```
        working-directory: ./terraform

      - name: Terraform Plan
        run: terraform plan -out=tfplan
        working-directory: ./terraform

      - name: Terraform Apply
        run: terraform apply -auto-approve tfplan
        working-directory: ./terraform

      - name: Export Outputs
        id: tf_outputs
        run: |
          echo "app_name=$(terraform output -raw app_name)" >>
$GITHUB_OUTPUT
        working-directory: ./terraform

    outputs:
      app_name: ${{ steps.tf_outputs.outputs.app_name }}

  deploy:
    needs: infrastructure
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Deploy to Azure Web App
        uses: azure/webapps-deploy@v2
        with:
          app-name: ${{ needs.infrastructure.outputs.app_name }}
          package: ./app
```

GitHub Actions example (Bicep):

```
name: Deploy Infrastructure with Bicep

on:
  push:
    branches: [main]

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Azure Login
        uses: azure/login@v1
        with:
          creds: ${{ secrets.AZURE_CREDENTIALS }}

      - name: Deploy Bicep
```

```
          uses: azure/arm-deploy@v1
          with:
            scope: resourcegroup
            resourceGroupName: rg-myapp-prod
            template: ./infra/main.bicep
            parameters: environment=prod appName=myapp
            failOnStdErr: false

        - name: Deploy Application
          uses: azure/webapps-deploy@v2
          with:
            app-name: app-myapp-prod
            package: ./app
```

Best practices:

- **Separate jobs**: infrastructure first, then application deployment
- **State management**: use remote state (Azure Storage for Terraform)
- **Secrets**: store cloud credentials in GitHub Secrets
- **Plan on PR**: run `terraform plan` on pull requests for review
- **Apply on merge**: only apply changes when merged to main
- **Environment protection**: require approvals for production deployments
- **Drift detection**: scheduled job to detect manual changes

Terraform remote state (Azure Storage):

```
terraform {
  backend "azurerm" {
    resource_group_name  = "rg-terraform-state"
    storage_account_name = "sttfstate"
    container_name       = "tfstate"
    key                  = "prod.terraform.tfstate"
  }
}
```

Pull request workflow:

- PR opened → `terraform plan` runs, posts plan as comment
- Team reviews infrastructure changes
- PR merged → `terraform apply` runs automatically
- Infrastructure versioned alongside application code

This completes the deployment automation cycle: code → CI → infrastructure provisioning → application deployment → production.
