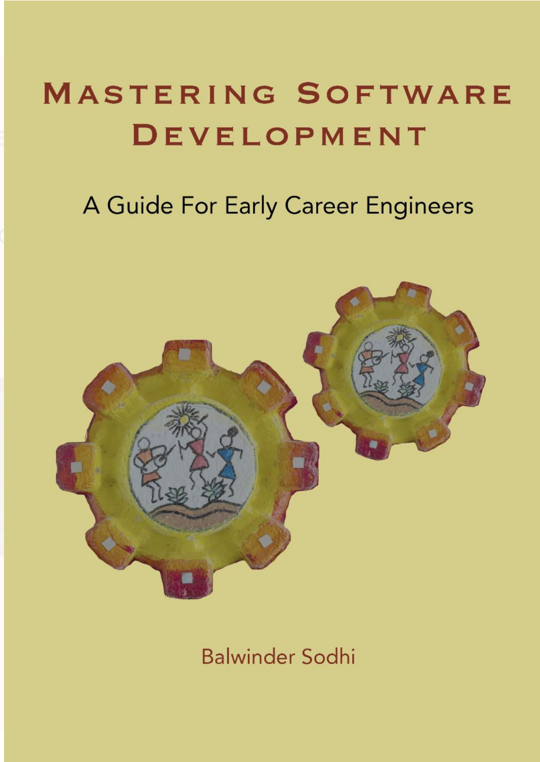


Deployment and Maintenance

Balwinder Sodhi

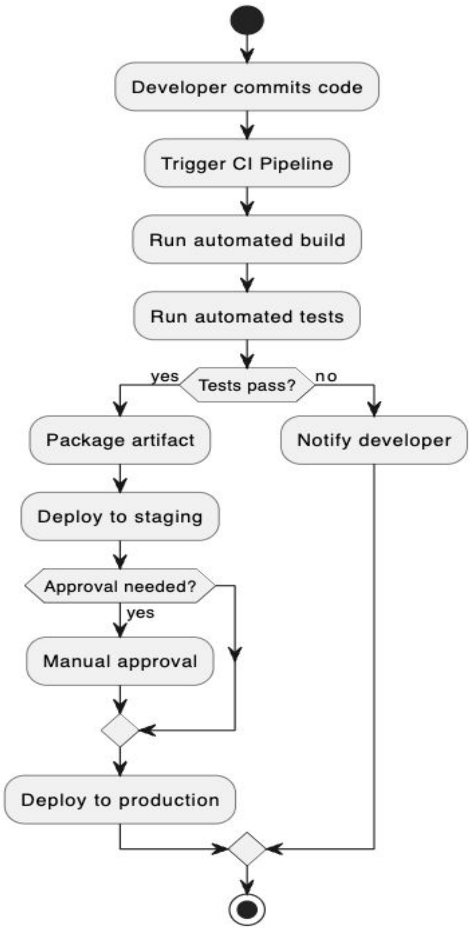
Automating the Release Pipeline: CI/CD

- What Is CI?
  - Continuous Integration
  - Frequent integration of code to the mainline
  - Automated builds + automated tests on every commit
  - Goal: detect issues early and maintain a stable codebase
- What Is CD?
  - Continuous Delivery / Continuous Deployment
  - Automated packaging + delivery of builds to staging/production
  - Goal: reduce manual steps, increase release reliability

Why should we care about CI/CD?

- Faster feedback loops
- Predictable and repeatable releases
- Higher code quality through early detection
- Improved collaboration between dev, QA, Ops
- Enables DevOps and trunk-based development

High-Level CI/CD Flow



Building a CI/CD Pipeline: Overview

A complete pipeline includes:

- Integration with version control
- Automated builds on triggers
- A comprehensive test suite
- An artifact repository for produced binaries
- Automated deployment stages
- Observability (logs, metrics, traces)

Version Control Integration

- Use Git (GitHub, GitLab, Bitbucket, Azure Repos)
- CI systems integrate via:
  - Webhooks
  - Branch rules
  - Pull/merge request triggers
- Enforce:
  - Protected main branches
  - Mandatory CI checks before merging
  - Commit statuses for visibility

Automated Builds

- Build triggers: Push, PR opened, nightly schedule
- Tools: Maven/Gradle, npm/yarn, Go build, Rust cargo, Docker build
- Build outputs:
  - Executables
  - JAR/WAR
  - Docker images
  - Static assets
- Build isolation using:
  - Containers
  - Sandboxed build runners

Automated Testing

Types of tests integrated into CI:

- Unit tests: Fast, isolated
- Integration tests: Test modules/services
- E2E tests: Full workflow validation
- API contract tests
- Security scans: SAST, dependency checks
- Performance tests (optional stage)

Best practice: run fast tests early, heavier tests later in pipeline.

Artifact Repository

Stores built outputs and versions:

- Artifactory, Nexus, GitHub Packages, AWS ECR, GCP Artifact Registry
- Benefits:
  - Immutable versioned artifacts
  - Support rollback
  - Separation of build and deploy
  - Audit trail of releases

Artifacts commonly stored:

- Docker images
- JAR/WAR binaries
- Helm charts
- Lambda bundles
- Mobile app bundles (APK/IPA)

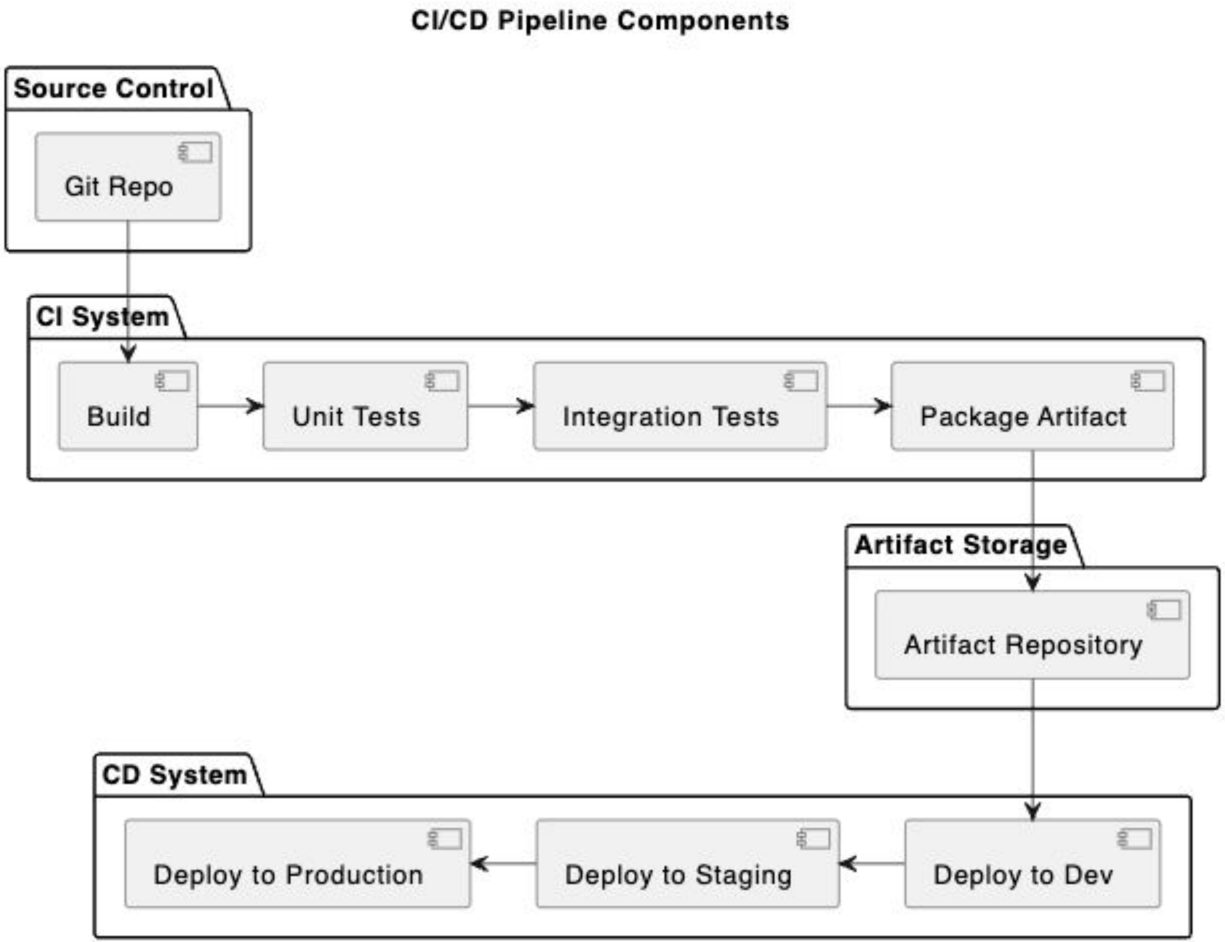
Deployment Automation

Typical deployment pipeline:

- Deploy to dev → staging → prod
- Use IaC (Terraform, CloudFormation) for infrastructure
- Deployment patterns:
  - Blue-green
  - Rolling
  - Canary
  - Shadow deployment

Deployment tools:

- ArgoCD, Spinnaker
- Kubernetes manifests + Helm
- Serverless deploy frameworks
- Mobile app store automation (Fastlane)


CI/CD Pipeline Components
Source Control
Git Repo
CI System
Build
Unit Tests
Integration Tests
Package Artifact
Artifact Storage
Artifact Repository
CD System
Deploy to Production
Deploy to Staging
Deploy to Dev

Tools & Technologies: Choosing CI/CD Platforms

- CI Platforms
  - GitHub Actions, GitLab CI
  - Jenkins, CircleCI
  - Azure DevOps
  - Buildkite
  - Google Cloud Build
  - AWS CodeBuild, CodePipeline
- CD/Deployment Tools
  - ArgoCD (GitOps)
  - Flux
  - Spinnaker
  - Octopus Deploy
  - AWS CodeDeploy

Selection Criteria

- Integration with code hosting
- Concurrent build capacity
- Secrets management
- Cloud/on-prem support
- YAML vs visual pipeline authoring
- Community ecosystem

CI/CD Best Practices

Reliability

- Keep pipelines fast (<10 min CI)
- Fail fast: early detection of issues
- Use caching (npm, Docker layers)

Security

- Scan dependencies
- Use signed artifacts
- Rotate secrets and use secret managers

Maintainability

- Modular pipelines
- Clear naming conventions
- Versioned pipeline configuration

Operational Excellence

- Use metrics: build time, success rate
- Retry logic for flaky tests
- Automatic rollback strategies

Example: CI/CD Pipeline for a Modern Web Application

Stack

- Backend: Node.js + Express
- Frontend: React SPA
- Database: PostgreSQL
- Infra: Kubernetes
- Hosting: AWS EKS + S3 + CloudFront

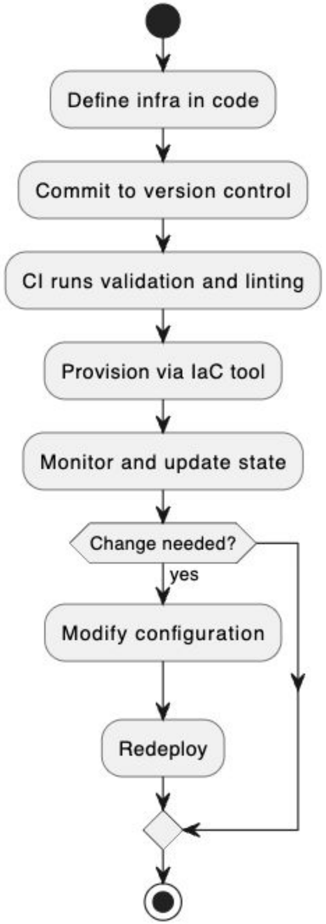
Pipeline Flow

1. Developer pushes to Git
2. CI triggers:
   - Install dependencies
   - Run lint + unit tests
   - Build frontend + backend
3. Build Docker images
4. Push images to ECR
5. Deploy to staging using ArgoCD
6. Run smoke tests
7. If all green → promote to production

Infrastructure as Code

Infrastructure as Code: Core Concept

- What is IaC?
  - Infrastructure provisioning through machine-readable configuration files
  - Environments become reproducible artifacts
  - Automates provisioning of servers, networks, storage, CI runners, K8s clusters, etc.
- Why it matters
  - Faster, more consistent provisioning
  - Enables DevOps automation
  - Simplifies multi-environment setup
- Key Idea Behind IaC
  - Treat infrastructure like code: versioned, testable, reviewable
  - Provisioning becomes deterministic
  - Enables ephemeral environments for feature branches
  - Automates rollback by reverting configs



Benefits of Adopting IaC

Speed & Consistency

- Automated provisioning
- Eliminates “snowflake servers”

Reliability

- Environments reproducible across dev, staging, prod
- Versioning ensures traceability

Scalability

- Cloud-native provisioning scales in minutes
- Supports autoscaling and dynamic infra

Security + Compliance

- Enforce policies via code
- Immutable infra → reduced drift

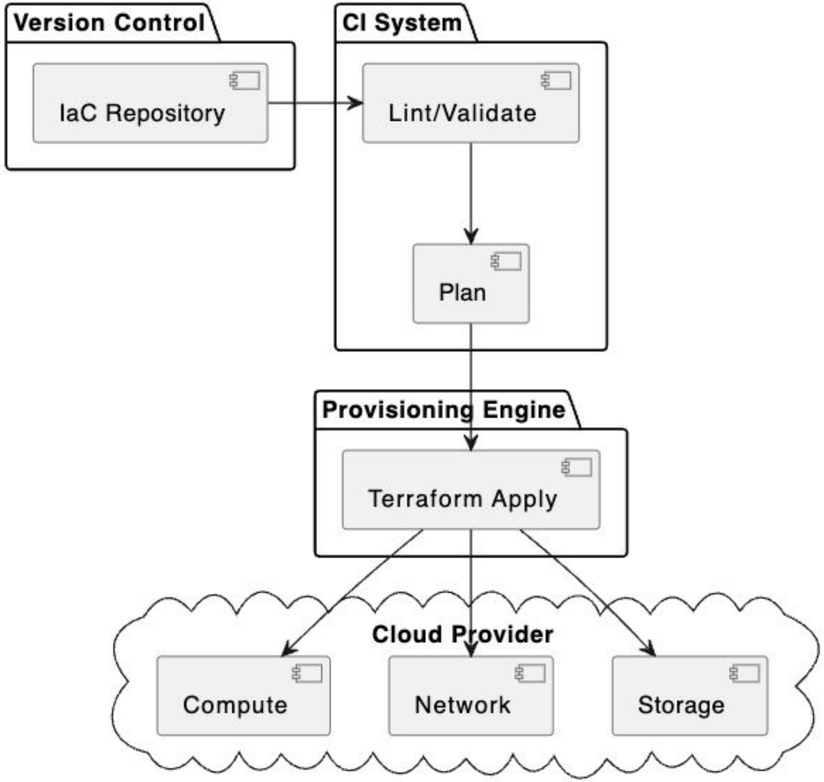
Declarative vs Imperative IaC

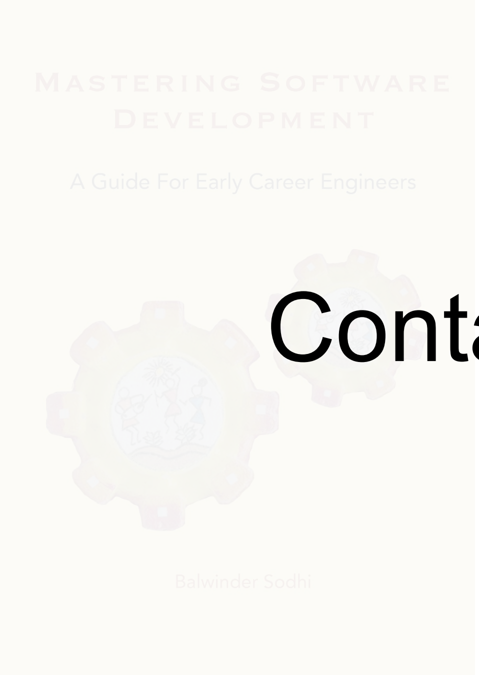
- Declarative (What)
  - Describe desired end state
  - Tool figures out how to reach it
  - Example: Terraform, Kubernetes Manifests
  - Pros: predictable, idempotent, audit-friendly
- Imperative (How)
  - Execute step-by-step commands
  - Example: Ansible (playbooks), Bash scripts
  - Pros: more control
  - Cons: harder to maintain at scale

Popular IaC Tools

- Terraform → Cloud-agnostic, state management
- CloudFormation → AWS-native
- Pulumi → IaC using general-purpose languages
- Ansible → Configuration management + provisioning
- Chef / Puppet → Policy-based config management
- Kubernetes Manifests / Helm → Declarative application infra

IaC Architecture Overview





Containerization & Orchestration

Containerization: Core Idea

- What is a container?
  - Lightweight runtime packaging application + dependencies
  - Uses OS-level virtualization
  - Predictable execution across dev/staging/prod
  - Much lighter than VMs
- What does it give us?
  - Reproducible builds
  - Fast deployment
  - Works well with IaC & CI/CD

Docker: Components

- Dockerfile → Describes how image is built
- Image → Immutable snapshot
- Container → Running instance of an image
- Registry → Stores images (DockerHub, ECR, GCR)

```
# Example Dockerfile for a
# Node.js application
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install --production
COPY . .
CMD ["node", "server.js"]
```

Kubernetes: Why We Need It

Challenges with many containers:

- Scheduling
- Health checks
- Rolling updates
- Networking
- Autoscaling

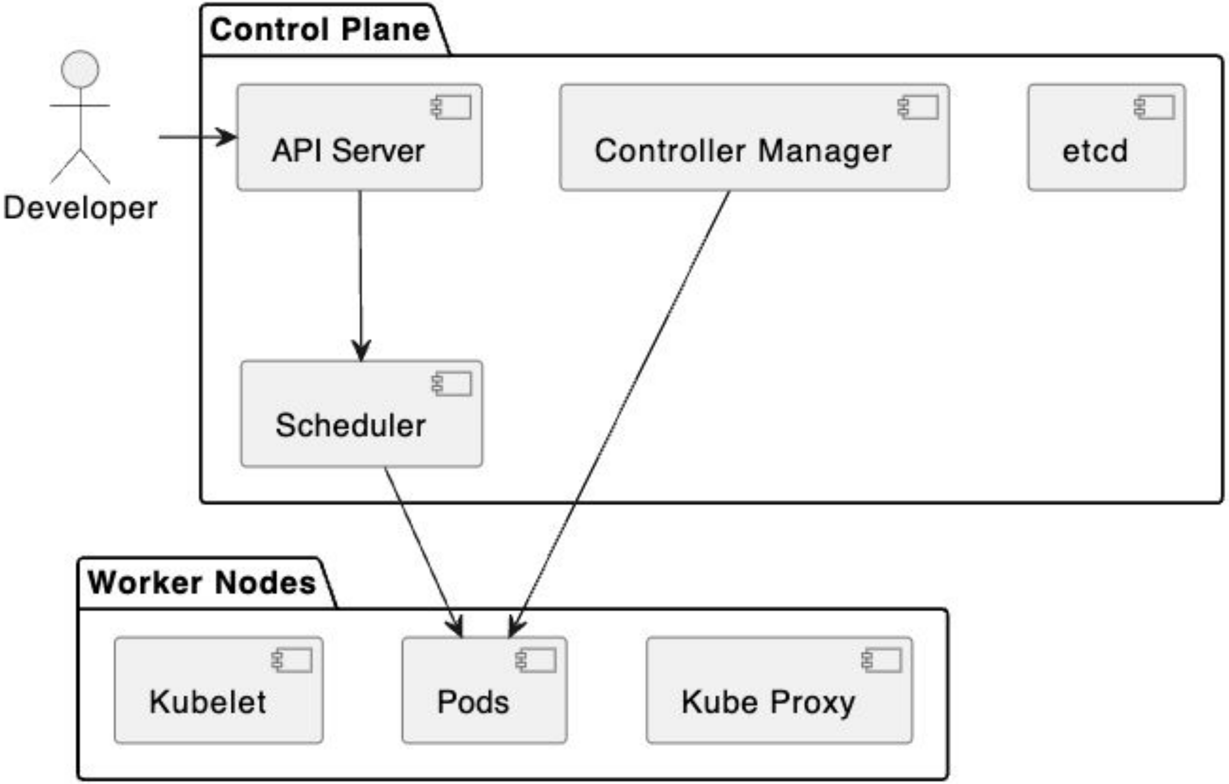
Kubernetes solves:

- Automated orchestration
- Declarative deployment model
- Self-healing
- Horizontal scaling
- Secrets and config mgmt

Kubernetes Building Blocks

- Pod → Smallest deployable unit
- Deployment → Replica management + rolling updates
- Service → Stable network endpoint
- ConfigMap / Secret → Externalized config
- Ingress → HTTP routing
- HPA → Autoscaling

High-level Kubernetes Architecture



Typical App Deployment Flow on Kubernetes

1. Developer pushes code
2. CI builds Docker image → pushes to registry
3. CD updates Deployment manifest
4. Kubernetes pulls image and performs rolling update
5. Service keeps endpoint stable
6. Observability tools monitor cluster health

Observability

Observability: Core Idea

Observability answers one question:

- **“Why is the system behaving this way?”**
- It requires:
  - Metrics
  - Logs
  - Traces

Not just monitoring "is it up?" but *understanding system’s behavior*.

The Three Pillars of Observability

1. Metrics
    - Numeric measurements over time
    - Good for KPIs, SLOs, alerts
2. Logs
    - Detailed event records
    - Useful for debugging
3. Traces
    - End-to-end request tracking across services
    - Critical for microservices

Key Metrics to Monitor

- Application metrics
  - Latency
  - Error rates
  - Throughput
  - Queue length
- Infrastructure metrics
  - CPU/memory usage
  - Disk I/O
  - Network traffic
- Business metrics
  - Signups, session lengths, other application specific items such as purchases

Log Aggregation & Analysis

Tools:

- ELK (Elasticsearch, Logstash, Kibana)
- Loki + Grafana
- Datadog Logs
- Splunk

Best Practices:

- Structure logs (JSON)
- Include correlation IDs
- Avoid logging secrets
- Centralize ingestion

Distributed Tracing

- Tracks a request across multiple services
- Shows bottlenecks and latency sources
- Tools: Jaeger, Zipkin, OpenTelemetry

Key Concepts:

- Spans
- Trace IDs
- Context propagation

Alerting & Notification

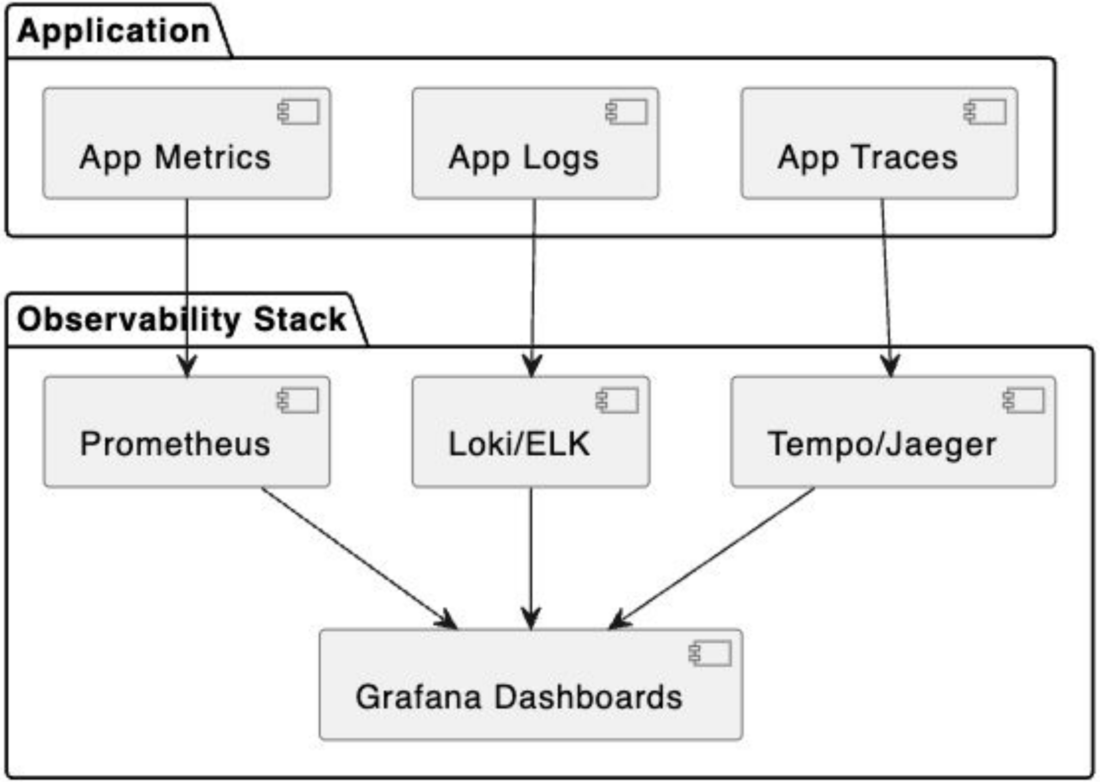
Trigger alerts based on:

- Error rate spikes
- Latency thresholds
- Saturation (CPU/Memory)
- SLO violations

Avoid alert fatigue:

- Use severity levels
- Use deduplication & grouping
- Always include runbook links

High-level Observability Architecture



Example Scenario: Monitoring with Prometheus + Grafana

- Prometheus Responsibilities
  - Scrape metrics from services
  - Store time-series data
  - Provide PromQL for analysis
- Grafana Responsibilities
  - Query Prometheus
  - Build custom dashboards
  - Visualize latency, error rates, CPU usage
- Typical Setup
  - Application exposes /metrics endpoint
  - Prometheus scrapes every N seconds
  - Grafana dashboard visualizes trends
  - Alerts fire when thresholds cross limits

Observability Best Practices

- Choose consistent metrics naming
- Implement RED (Rate, Errors, Duration) or USE (Utilization, Saturation, Errors)
- Use OpenTelemetry for unified instrumentation
- Capture correlation IDs across hops
- Store logs cost-effectively
- Regularly audit dashboards and alerts

Version Control (Git): Mastering Collaboration and Code Management

We will talk about:

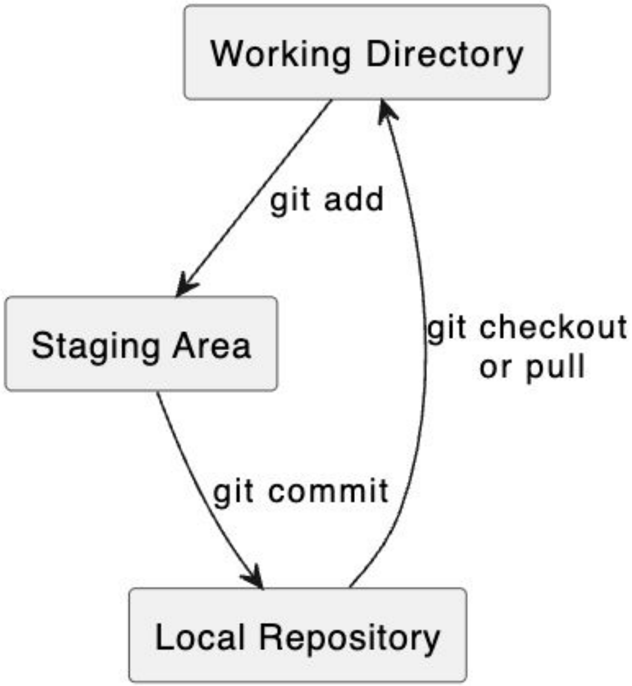
- Core Git concepts
- Branching strategies
- Advanced Git techniques
- Real-world collaboration workflow with GitHub

Why Version Control Matters

- Tracks every change to your codebase
- Enables safe experimentation via branching
- Facilitates collaboration across teams
- Provides a single source of truth for production-ready code
- Integrates deeply with CI/CD, code review, automation

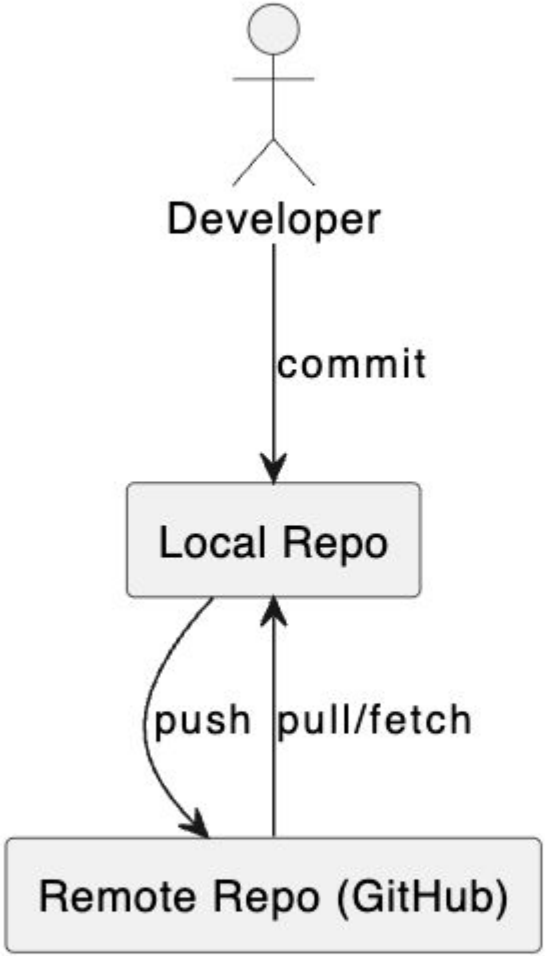
Core Git Concepts

- Repository
  - A directory tracked by Git
  - Contains a .git folder storing history and metadata
- Commits
  - Snapshots of the project at a point in time
  - Each commit has an author, timestamp, and SHA hash
- Working Directory, Staging Area, Local Repo
  - Working directory: your uncommitted changes
  - Staging: commit preparation area
  - Local repo: commit history



Remote Repositories

- Hosted on GitHub, GitLab, Bitbucket
- Facilitate collaboration and CI
- Local repo syncs with remote via `fetch`, `pull`, `push`



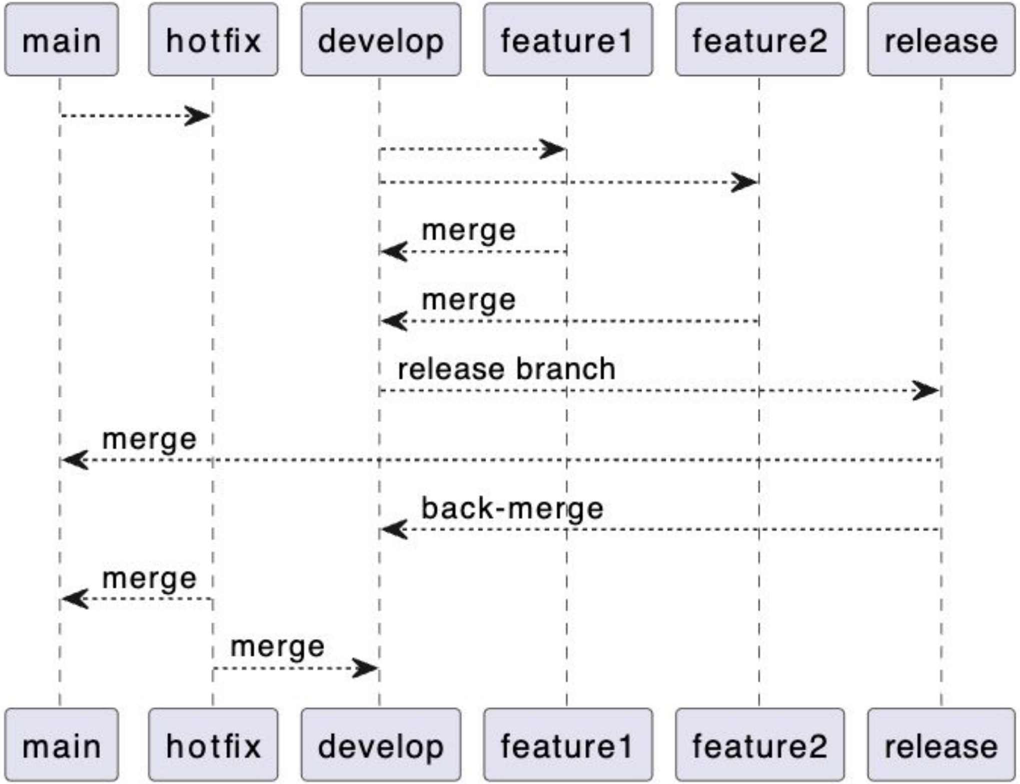
Branches

- Lightweight pointers to commits
- Enable isolated development
- Common branches:
    - `main/master` → stable production
    - `develop` → integration branch
    - `feature/bugfix/hotfix` branches

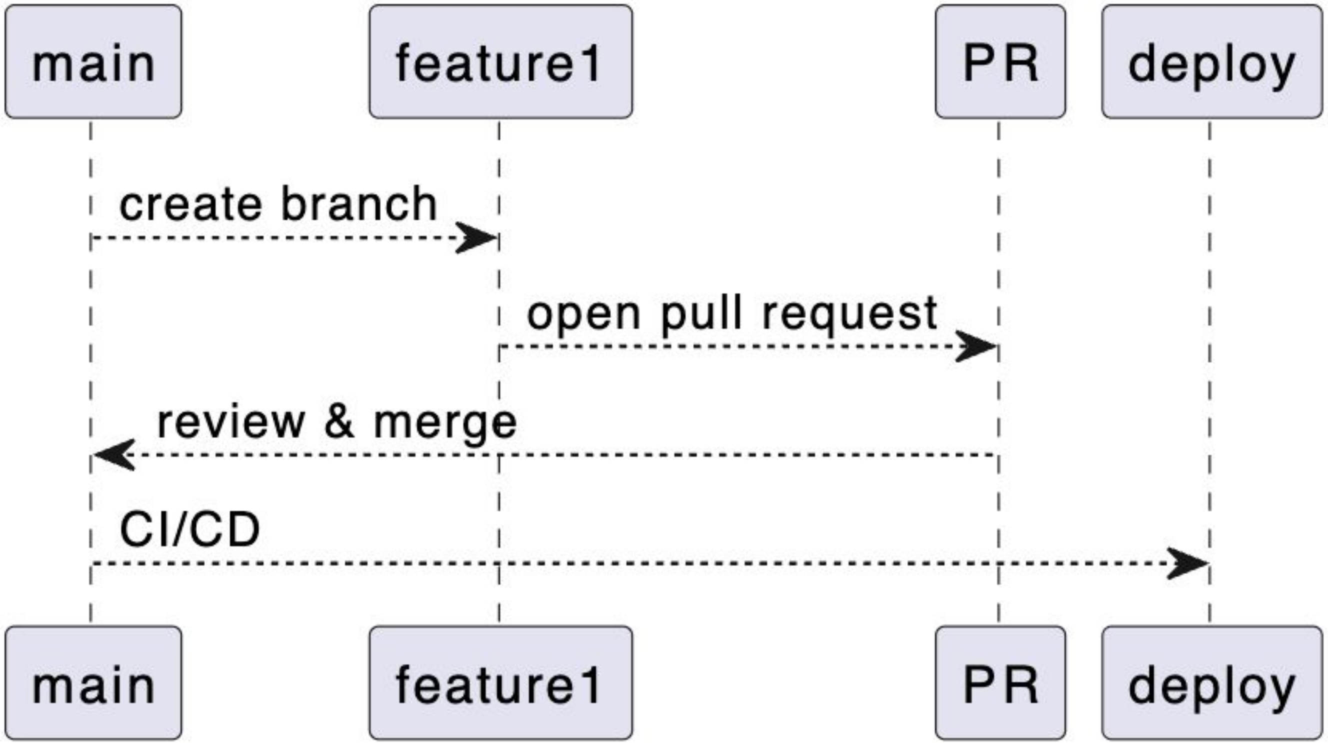
Branching Strategies Overview

1. Git Flow
   - Structured, ideal for release-driven teams
   - main, develop, feature, release, hotfix branches
2. GitHub Flow
   - Simple, continuous delivery friendly
   - All development on feature branches → PR → main
3. Trunk-Based Development
   - Very short-lived branches
   - Frequent merges to main
   - Works well with strong CI/CD pipelines

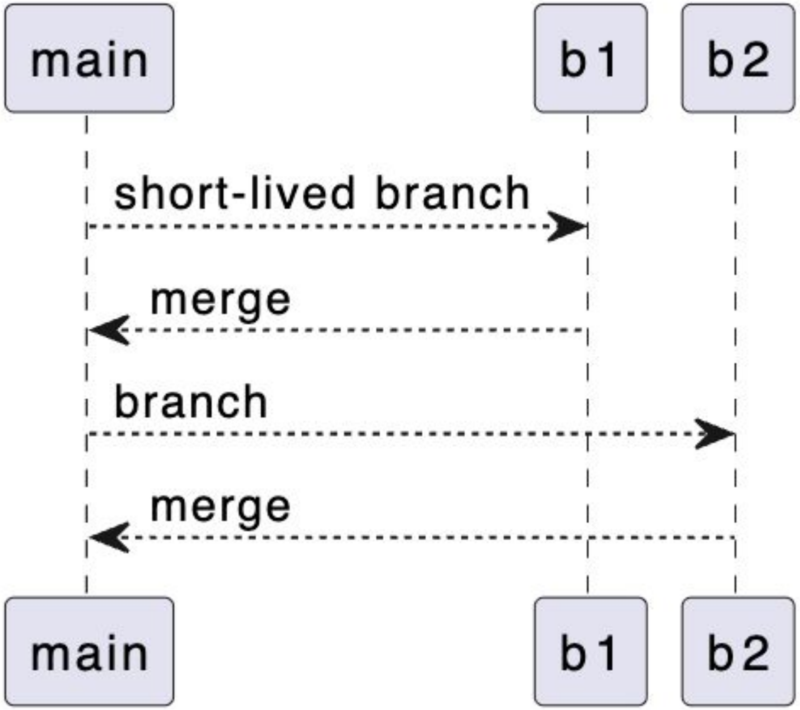
Git Flow Diagram



GitHub Flow Diagram

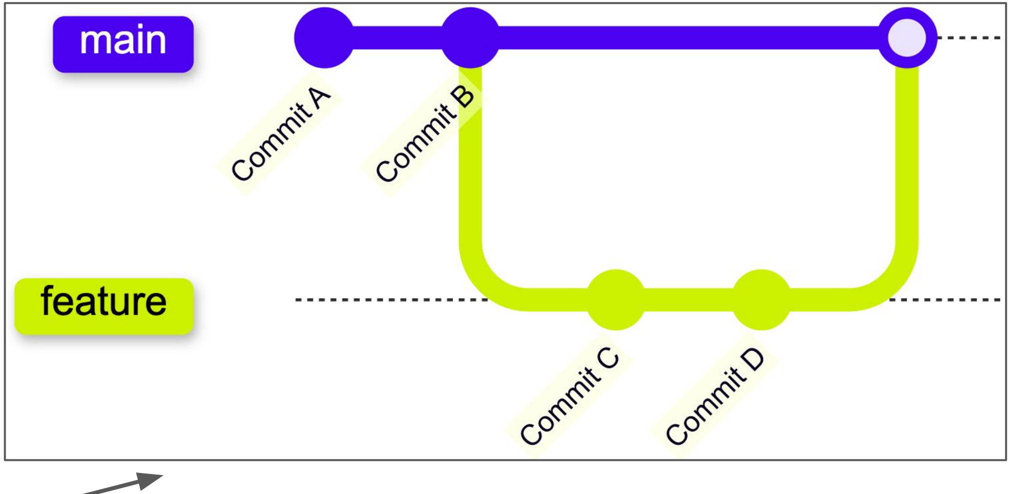


Trunk-Based Development Diagram

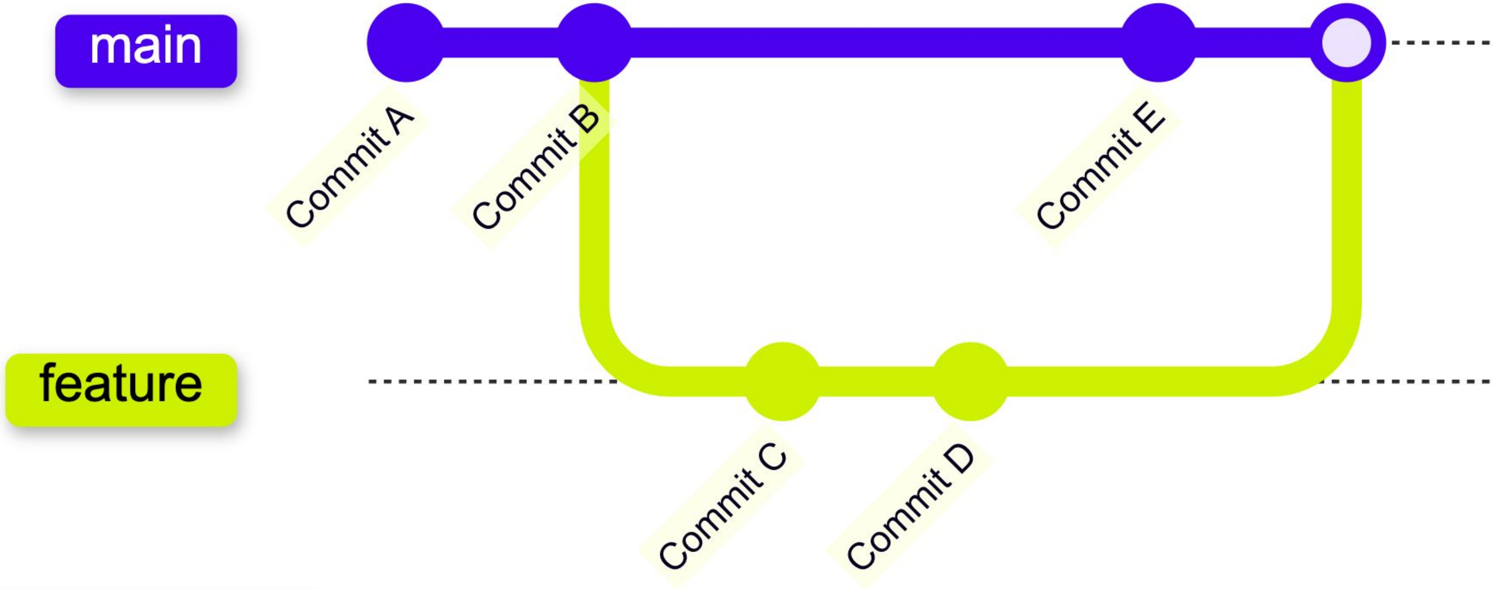


Merging vs Rebasing

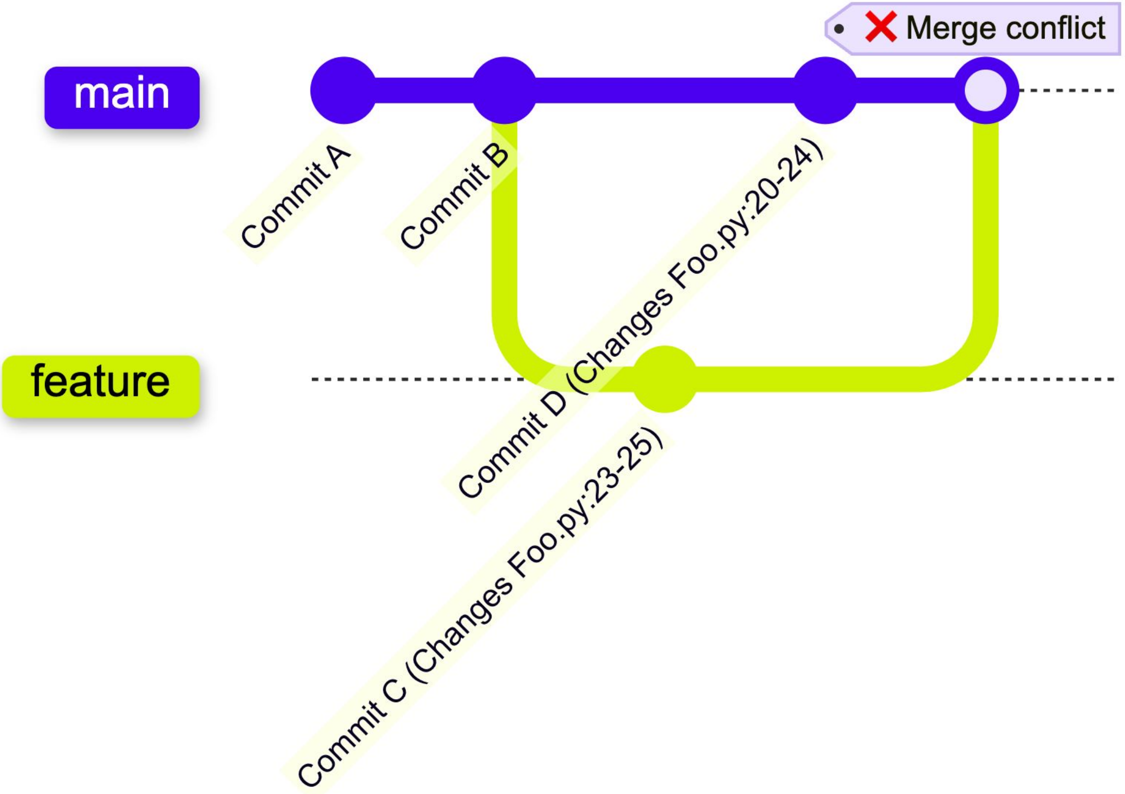
- Merge
  - Preserves full branch history
  - Creates merge commits
  - Great for shared branches
- Rebase
  - Rewrites history
  - Cleaner linear history
  - Avoid rebasing shared branches
- Fast-Forward Merge
  - Happens when main has no new commits since branch creation
  - Simply advances the pointer
  - Clean, linear history



Diverging commit E on `main` Prevents Fast-forwards Merge



Scenario Leading to a Merge Conflict



Resolving Merge Conflicts

- Occurs when two branches modify the same lines
- Conflict markers appear in files
- Best practices:
    - Resolve logically and test
    - Communicate with teammates
    - Keep branches short-lived

Stashing

- Save uncommitted work temporarily
- Useful when switching context
- Commands:
    - git stash push
    - git stash list
    - git stash apply

Cherry-Picking

- Apply a specific commit from another branch
- Useful for hotfixing or selective backports
- Avoid excessive use to prevent duplicate commits

Hooks for Automation at Commit-Time

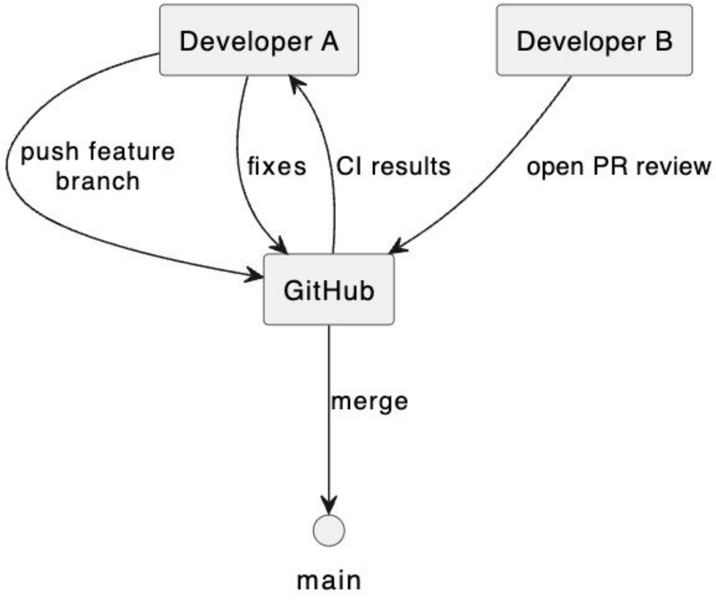
- Local scripts that trigger on Git events
- Common hooks:
  - `pre-commit` → lint, format
  - `commit-msg` → enforce message conventions
  - `pre-push` → run tests

Example Scenario: Team Collaboration (End-to-End)

Context: Team of 3 engineers building a feature using Git + GitHub.

Steps:

1. Create issue in GitHub
2. Branch from main: feature/user-auth
3. Push and open PR
4. CI runs tests, lint, builds
5. Code review + comments
6. Squash and merge
7. Deploy via CI/CD



GitHub Pull Requests

- Small, focused changes
- Clear description with before/after context
- Linked issues
- Tests added/updated
- Screenshots for UI changes
- Lint + static analysis clean

Common Real-World Problems

- Diverged branches
- Accidental commits on main
- Force-push misuse
- Binary file conflicts
- Large file history bloat
- Mitigation
  - Enable protected branches
  - Require PR reviews
  - Use Git LFS for large files
  - Use `git reflog` to recover lost changes

CI/CD Integration

- Pull request triggers automated pipelines
- Ensures `main` stays green
- Typical tasks:
  - Build
  - Unit tests
  - Static analysis
  - Vulnerability scanning
  - Auto-deploy on merge

Deployment Strategies – Choosing a Product Release Approach

What Are Deployment Strategies?

- Key ideas
  - Methods used to deliver new versions of software into production.
  - Aim to balance speed, risk, cost, and user experience.
  - Modern systems require strategies that support high availability, zero-downtime updates, and fast rollback.
- Considerations
  - System architecture (monolith vs microservices).
  - Traffic distribution abilities (load balancer, gateway).
  - CI/CD maturity.
  - Business constraints (compliance, SLAs).

Why Deployment Strategies Are Important

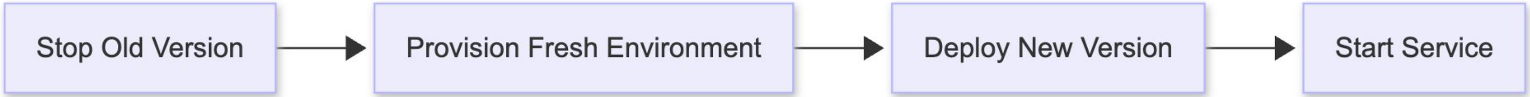
- Modern applications require high availability, frequent releases, and fast recovery from bad deployments.
- Deployment method impacts:
  - Risk of failure
  - User experience (downtime, errors)
  - Operational cost
  - Observability + rollback capabilities
- Engineering concerns
  - Traffic routing and load balancing.
  - Stateful vs stateless components.
  - Data schema evolution.
  - Monitoring and alerting capabilities.
  - Compliance or regulatory constraints.

Core Dimensions of Deployment Design

1. Downtime tolerance
    - Mission-critical systems: aim for zero-downtime or near zero.
2. Infrastructure elasticity
    - Do you have the ability to run parallel environments?
    - Cloud-native systems have more flexibility.
3. Risk controls
    - Ability to pause, roll back, progressively release.
4. Operational observability
    - Metrics, logs, traces, automated canary analysis.
5. Delivery frequency
    - Frequent deployers need techniques like trunk-based development, feature flags, canaries.

Big-Bang / Recreate Deployment

- How it works
  - Shut down the old version.
  - Deploy new version on fresh instances.
  - Start service again.
- Best for
  - Internal tools
  - Low-traffic batch systems
  - Systems where maintenance windows are acceptable
- Operational considerations
  - Requires coordinated downtime notifications.
  - Useful when infrastructure is hard to version or update incrementally.
  - Simplifies rollback: you redeploy the old version.



Rolling Deployment

- How it works
  - Update a subset of instances in waves.
  - Load balancer drains traffic from nodes being updated.
  - Continue until all instances are running the new version.
- Strengths
  - Very little user disruption.
  - No need to double infrastructure.
  - Works well with Kubernetes: rolling update is built-in.
- Weaknesses
  - Mixed-version behavior during rollout → may cause subtle bugs.
  - Rollback isn't instantaneous. Must roll forward or roll back through waves.
- Best for
  - Stateless microservices
  - Services where requests don’t depend on long-lived sessions

Blue/Green Deployment

How it works:

- Full duplication of production environment.
- Deploy new version to Green while Blue serves traffic.
- Run tests in Green: smoke tests, performance, synthetic checks.
- Switch load balancer routing from Blue → Green.

Common patterns:

- Used with database where schema changes follow expand → deploy → contract pattern.

Strengths:

- Fastest rollback of all strategies.
- Simplifies validation in production-like environment.
- Zero downtime if switching is atomic.

Weaknesses:

- Expensive: two full environments.
- Database schema must be carefully designed to allow dual versions.

Canary Deployment 1/2

Goal: Reduce risk by gradually exposing the new version to real users.

Process:

- Route 1–5% traffic to canary.
- Monitor key metrics: latency, RPS, error rate, saturation.
- If stable → ramp up to 10%, 25%, 50%, 100%.
- If regression detected → rollback immediately.

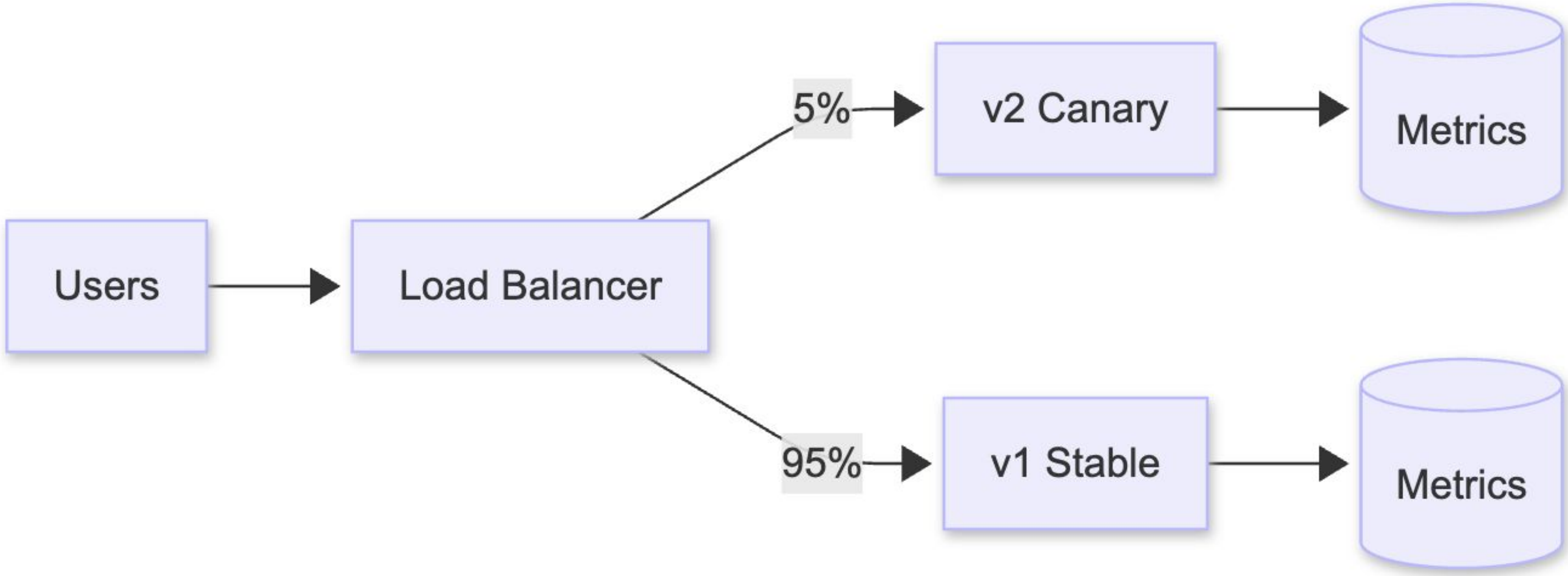
Strengths:

- Best real-world safety net.
- Enables automated analysis tools (e.g., Kayenta in Spinnaker).
- Minimizes blast radius of issues.

Weaknesses:

- Observability must be strong.
- Requires traffic shaping capabilities.

Canary Deployment 2/2



Automated Canary Analysis

- Key metrics monitored
  - Success rate / error rate
  - Latency distribution (p95, p99)
  - CPU, memory, network throughput
  - Business KPIs: drop in conversions, increase in failures
- Approach
  - Statistical tests (e.g., Mann-Whitney U) compare canary vs baseline.
  - Automated pass/fail thresholds trigger promotion or rollback.
- Best for
  - Large distributed systems
  - High-stakes deployments

A/B Testing vs Canary

<u>Canary</u>

- Goal: *safe deployment*.
- Evaluates stability, correctness.

<u>A/B Testing</u>

- Goal: *product learning*.
- Compares behaviors across user segments.

<u>Risks</u>

- A/B testing can inadvertently become a deployment method if not controlled.
- Canary affects entire system reliability; A/B impacts business KPIs.

Feature Flags / Feature Toggles

- Concept: Deploy code to production with features dark-launched behind flags.
- Strengths
  - Separate deployment from release.
  - Rollout control per user segment.
  - Kill switch for quick rollback.
  - Enables trunk-based development → reduces merge conflicts.
- Challenges
  - Feature-flag debt: stale flags clutter code.
  - Must design flags carefully (boolean, dynamic, multivariate).
  - Requires flag management system (LaunchDarkly, Unleash, homegrown).

Shadow Deployment

- How it works
  - Production traffic is mirrored to a new version.
  - Responses from the new version are ignored.
  - Useful for validating:
    - ML models (accuracy, drift)
    - Performance regression
    - New caching strategies
    - Protocol/serialization changes
- Strengths
  - Zero-risk evaluation before release.
  - Works well in ML and AI-driven services.
- Weaknesses
  - Doubles system load.
  - Hard to compare outputs if behavior diverges.

Database Deployment Strategies

- Challenges
  - Applications may run multiple versions simultaneously (rolling, canary).
  - Database must remain compatible across versions.
- Patterns
  - Expand → Migrate → Contract
    - Add new schema elements (non-breaking).
    - Deploy app using new schema.
    - Migrate data.
    - Remove old schema.
  - Dual write + background migration
    - Write to old + new schema.
    - Read from old until new is valid.
    - Switch reads to new version.
  - Versioned schema
    - Each version independently reads/writes a version-specific structure.

Advanced Rollback Techniques

1. Instant rollback (Blue/Green)
   a. Traffic switch back to previous version.
2. Rolling rollback
   a. Update nodes back to previous stable version in batches.
3. Config-level rollback
   a. Feature flags disabled → instant kill switch.
4. Data migration rollback
   a. Hardest to revert.
   b. Requires reversible migration strategy or compensating migration.

Key rule: Rollback must always be cheaper than deploy.

Deployment Strategy Decision Matrix

Requirement	Best Strategy
Zero downtime	Blue/Green, Rolling
Fast rollback	Blue/Green, Feature Flags
Low cost	Rolling
Observability-driven risk reduction	Canary
Large experiments	A/B testing
Validate performance before release	Shadow
Frequent deployments (daily)	Trunk-based + Feature Flags

A Modern Release Strategy For a SaaS product

- Use **feature flags** to control exposure.
- Use **canary deployment** for backend services.
- Use **blue/green** for major infrastructure upgrades.
- Use **shadow mode** for ML model updates.
- Use **trunk-based development** to avoid long-lived branches.
- Use **expand-contract DB migrations** for safe schema changes.

Management of Configuration Data

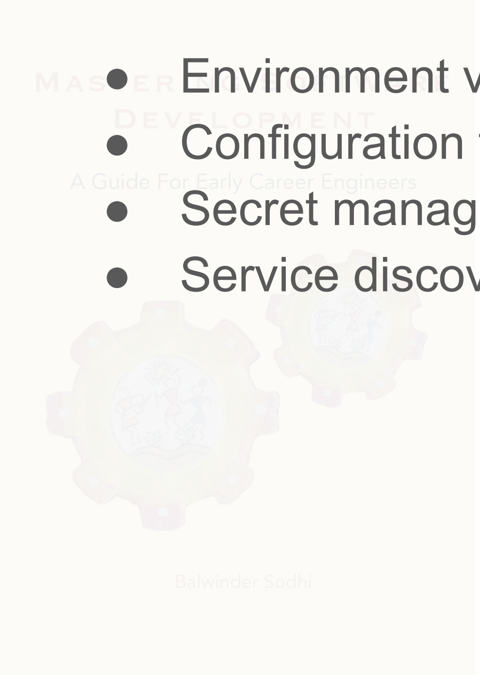
What Counts as Configuration?

- Database connection strings
- API endpoints and credentials
- Feature flags
- Logging parameters
- Third-party service keys
- Environment-specific tuning (caches, thread pools, timeouts)

Why Bother About Configuration Management?

- Applications behave differently across dev, test, staging, and production.
- Configuration management ensures:
  - Consistent behavior across environments
  - Safe handling of secrets
  - Clean separation between code and configuration
- Avoids hard-coding values, reduces errors, improves deployability.

Approaches to Storing Configuration

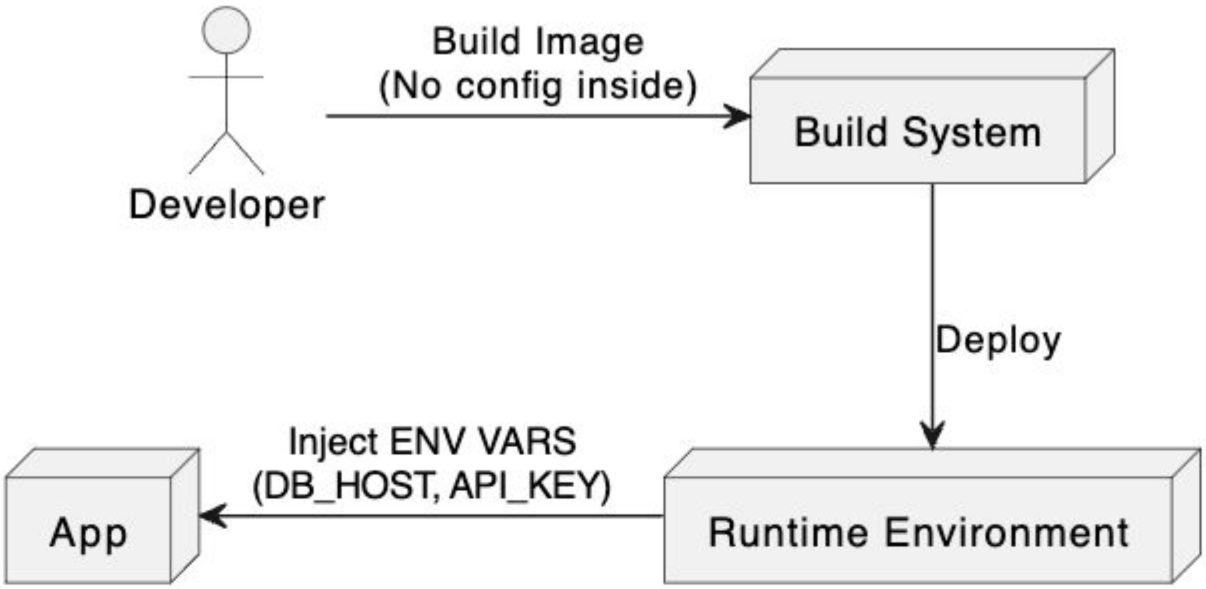


- Environment variables
- Configuration files (YAML, JSON, TOML, HOCON)
- Secret managers (HashiCorp Vault, AWS/GCP/Azure)
- Service discovery / dynamic configs (Consul, etcd, Zookeeper)

Environment Variables

- Key–value pairs injected into the runtime environment
- Common in container orchestration (Docker, Kubernetes)
- Good for:
    - Small values
    - Credentials
    - Feature toggles
- Supported by all languages and platforms.

Environment Variables Flow



Pros & Cons of Environment Variables

Pros

- Easy to override per environment
- No risk of accidental check-in
- Portable; widely supported

Cons

- Hard to manage large structured configs
- Debugging missing/incorrect env vars can be tricky
- Risk of leakage via logs / process lists (old systems)

Configuration Files

- Store structured config in JSON, YAML, TOML, INI, XML, etc.
- Loaded at runtime.
- Good for:
  - Readable, hierarchical settings
  - Large configuration surfaces
- Can be versioned for traceability (but avoid storing secrets).
- Pros
  - Human-readable
  - Great for complex/hierarchical data
  - Can maintain “defaults” checked into source control
- Cons
  - Easy to accidentally commit secrets
  - Harder to override in containerized deployments
  - Requires file access at runtime

Configuration Files: Example (YAML)

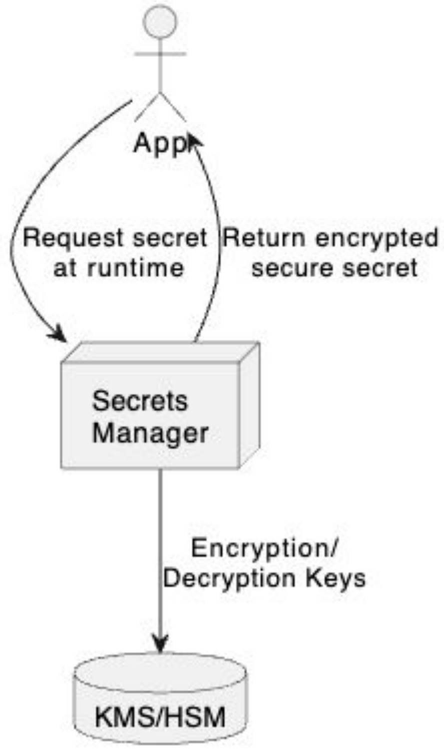
```yaml
app:
  logLevel: INFO
  cache:
    enabled: true
    ttlSeconds: 120
database:
  url: jdbc:postgresql://staging-db/app
  poolSize: 10
```

Secrets Management: Why Not Store Secrets in Git

- Once committed, secrets cannot be "unseen"
- Developers may clone logs, screenshots, or dumps containing secrets
- Leaks happen through:
  - CI/CD logs
  - Diagnostic printing
  - Debug builds
  - External dependency repos

Secrets Managers (Vault, AWS Secrets Manager, etc.)

- Systems designed to store, rotate, audit, and encrypt secrets.
- Features:
  - Automatic key rotation
  - Fine-grained access control
  - Audit logs
  - Temporary tokens / dynamic credentials
- Applications fetch secrets securely at runtime.



Secret Injection Patterns

- *Pull model:* app fetches secrets at startup (Vault, AWS SDK).
- *Push model:* CI/CD injects secrets as environment variables.
- *Sidecar container:* secret agent auto-updates files used by the app.
- *Encrypted config files:* decrypted at runtime (SOPS, KMS-integrated tools).
- Handling Secrets in Local Development
  - Use .env files stored locally but excluded from Git
  - Static mock secrets for dev (e.g., “local-test-key”)
  - Use secret manager dev instances where possible
  - Avoid onboarding developers to real production secrets

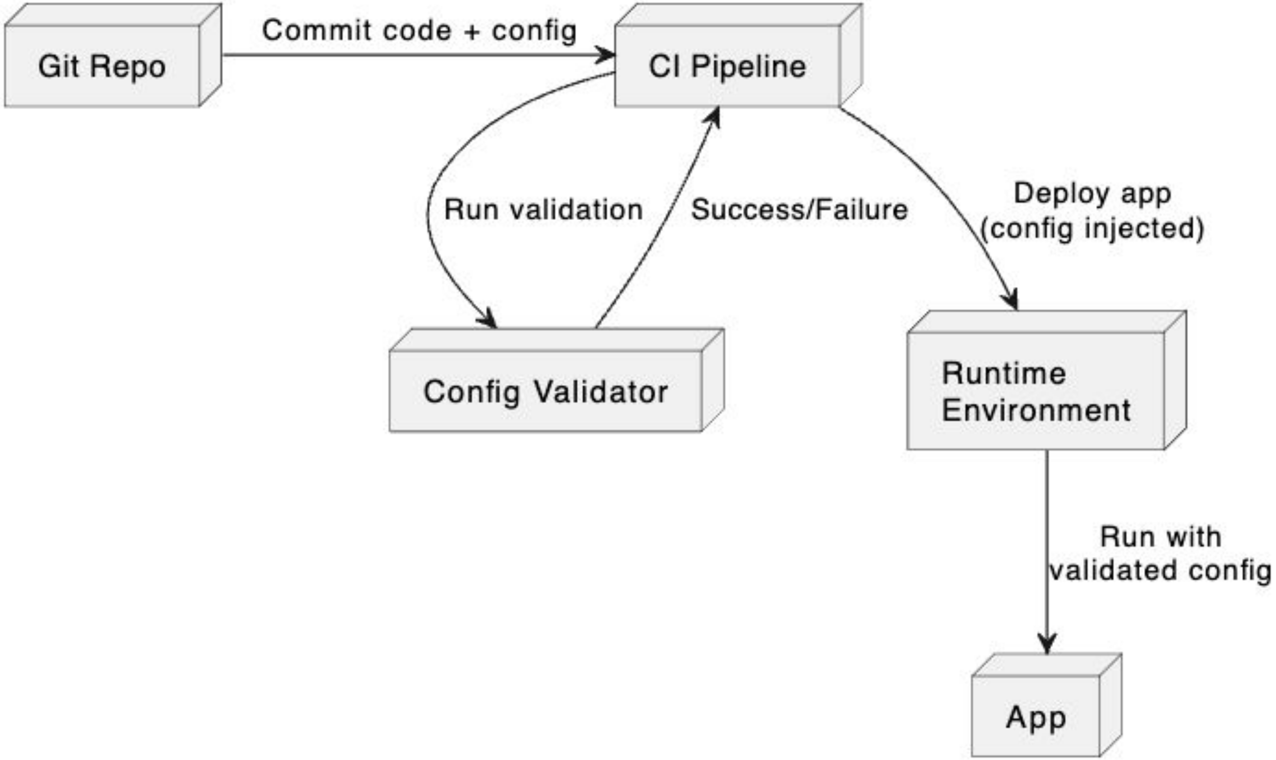
Best Practices

- Keep configuration out of code
- Separate defaults (in repo) from sensitive overrides (external)
- Store configs in environment
- Provide clear schema/validation for configs
- Fail fast when required configs are missing
- Use managed secret systems instead of environment variables for high-sensitivity values
- Rotate secrets regularly
- Avoid long-lived tokens
- Encrypt in transit and at rest
- Use least-privilege access (IAM roles, service accounts)

Managing Configuration Across Environments

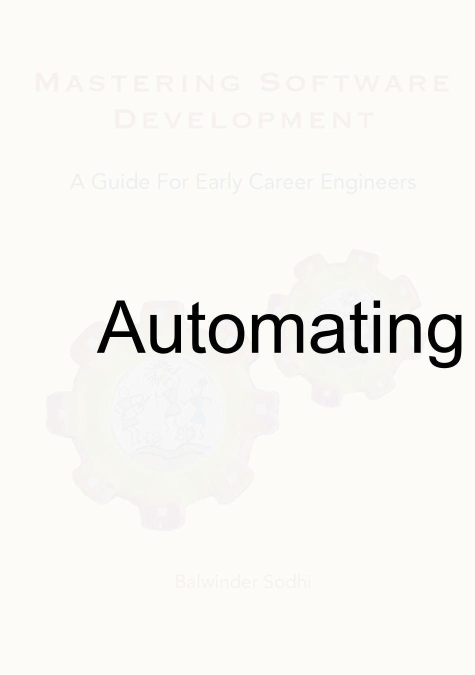
- Maintain configuration per environment:
  - development.yaml
  - staging.yaml
  - production.yaml
- Use merge strategies (defaults + overrides)
- Use feature flags instead of branching code for env-specific behavior.
- Schema validation (JSON Schema, TypeSafe Config, pydantic, etc.)
- Validation steps:
  - Check presence
  - Check type
  - Check value ranges
- CI pipeline should validate configs before deployment.

CI/CD with Config Validation



Tooling & Framework Support

- Kubernetes ConfigMaps & Secrets
- Spring Boot Profiles
- Django settings modules
- Node.js Dotenv + Config libraries
- Terraform + Vault integration
- SOPS for encrypted YAML/JSON
- Helm value files for K8s deployments



Automating Database Changes: Database Migrations

What is a Database Migration?

- A migration is a versioned, incremental change to the database structure or static reference data that it stores.
- Types of migrations
    - Schema migrations (DDL).
    - Data migrations (moving/transformation).
    - Seed/reference data migrations.

Core idea: Code and database evolve together.

Motivation for DB Migrations

Key motivations:

- Schemas evolve as features grow; manual SQL updates don’t scale.
- Need for predictable, repeatable, version-controlled schema changes.
- Essential for CI/CD pipelines and multi-environment deployments.

Outcomes:

- Safer deployments.
- Traceability of every DB-level change.
- Empower teams to collaborate on schema evolution.

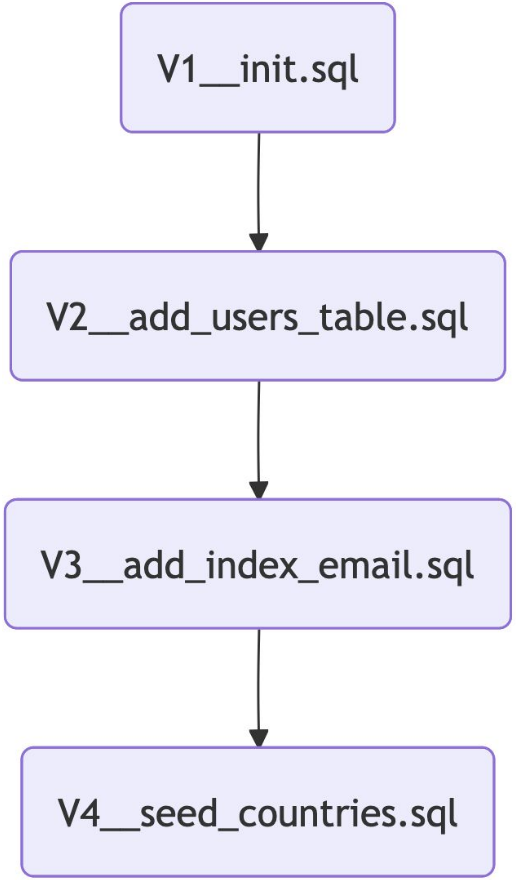
Migration Tools Ecosystem Examples

- Popular frameworks
  - Flyway (Java ecosystem, CLI)
  - Liquibase (Declarative XML/JSON/YAML/SQL)
  - Alembic (Python / SQLAlchemy)
  - Rails ActiveRecord Migrations
  - Django Migrations
  - EF Core Migrations
- Common capabilities
  - Versioning
  - Up/Down scripts
  - Rollbacks (if supported)
  - Checksums and integrity verification

How Migration Versioning Works

Key rules:

- Ordered execution by version.
- Never modify a previously executed migration.
- Allow repeatable migrations (in some tools).



Writing Migrations: Up/Down Pattern

- Up migration (apply change)
  - Create/alter tables
  - Add/remove indexes
  - Insert/update static data
- Down migration (rollback change)
  - Reverse the Up step
  - Used for local development or controlled rollback scenarios
- Example:

```sql
-- Up
ALTER TABLE orders ADD COLUMN priority INT;
-- Down
ALTER TABLE orders DROP COLUMN priority;
```

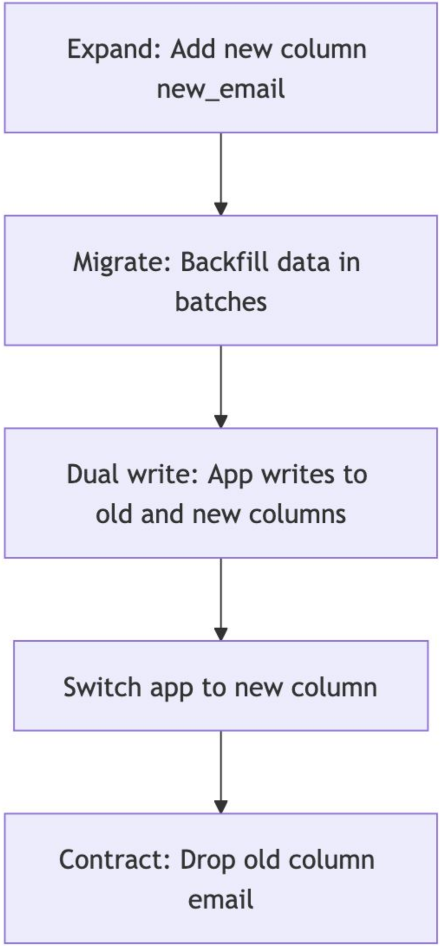
Idempotency and Safety

- Principles
  - Migrations should be deterministic.
  - Avoid relying on environment-specific state.
  - Use safe operations when possible:
    - CREATE TABLE IF NOT EXISTS
    - Online index creation for large tables
- Checks
  - Validate database state before applying migrations.
  - Use checksums to detect drift.

Managing Data Migrations

Considerations:

- Data migrations can be expensive and risky.
- Break large data changes into smaller batches.
- Use background workers for operational data migrations (e.g., double-write strategy).
- Prefer declarative transformations where possible.
- **Pattern:** Expand → Migrate → Contract
  - Add new column/table.
  - Migrate data gradually.
  - Remove old schema parts.



Schema Drift and Detecting Differences

Causes:

- Hotfixes made directly on production DB.
- Legacy systems with inconsistent environments.
- Manual migrations executed improperly.

Solutions:

- Drift detection tools (Liquibase/Flyway).
- Strict CI enforcement: schema must match latest migration.
- Automated DB state validation during deployment.

Rollbacks & Forward-Only Migrations

Two schools of thought:

- Rollback-friendly
  - Every migration has a Down script.
  - Good for early-stage projects.
- Forward-only (common in high-scale systems)
  - No Down scripts; rollback is done by new forward migrations.
  - Safer for systems with high traffic and data residency constraints.

Rule: *Never rely on DDL rollbacks in production unless you understand the risks.*

Patterns for Zero-Downtime Migrations

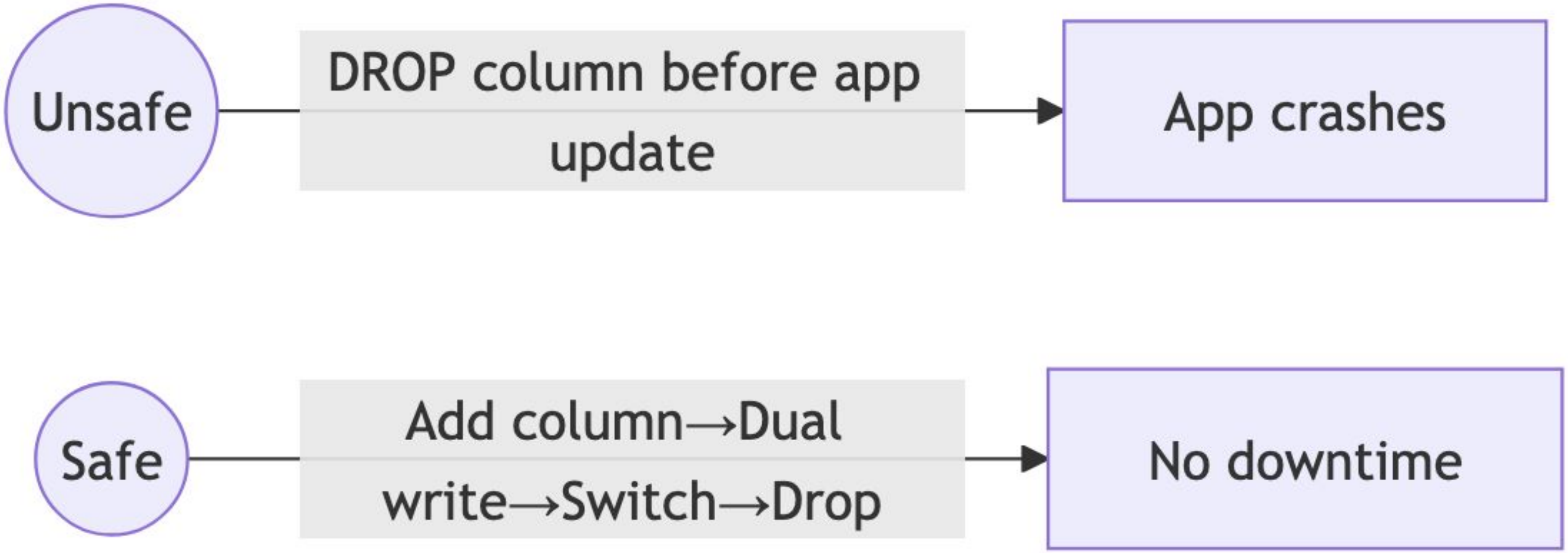
Avoid breaking changes:

- Never rename/remove columns without an expand–migrate–contract cycle.
- Avoid locks on large tables: use online DDL.
- Add new columns with defaults avoided (since defaults rewrite entire table on some DB engines).

Zero-downtime workflow:

- Always deploy migration before deploying code that depends on it.

Unsafe vs Safe Migration



Practical Tips & Best Practices

- Store migrations in version control next to application code.
- Keep migration scripts small and focused.
- Test schema changes with realistic datasets.
- Document assumptions (e.g., expected data volumes).
- Use feature flags when coordinating data migrations with app logic.
- Monitor migration time in production.