Introduction to CI/CD

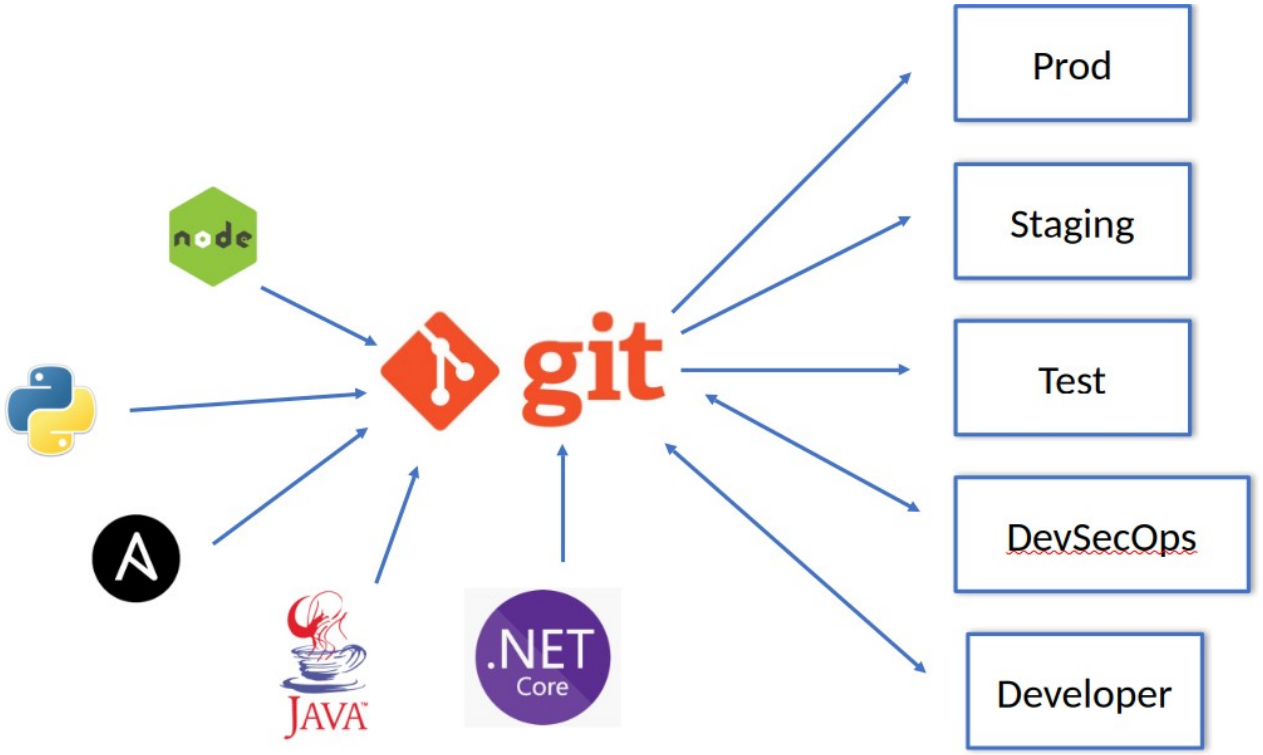
Module 4: Continuous Integration

12 Factor App

Factor	Description
I. Codebase	One codebase tracked in revision control, many deploys
II. Dependencies	Explicitly declare and isolate dependencies
III. Config	Store config in the environment
IV. Backing services	Treat backing services as attached resources
V. Build, release, run	Strictly separate build and run stages
VI. Processes	Execute the app as one or more stateless processes
VII. Port binding	Export services via port binding
VIII. Concurrency	Scale out via the process model
IX. Disposability	Maximize robustness with fast startup and graceful shutdown
X. Dev/prod parity	Keep development, staging, and production as similar as possible
XI. Logs	Treat logs as event streams
XII. Admin processes	Run admin/management tasks as one-off processes

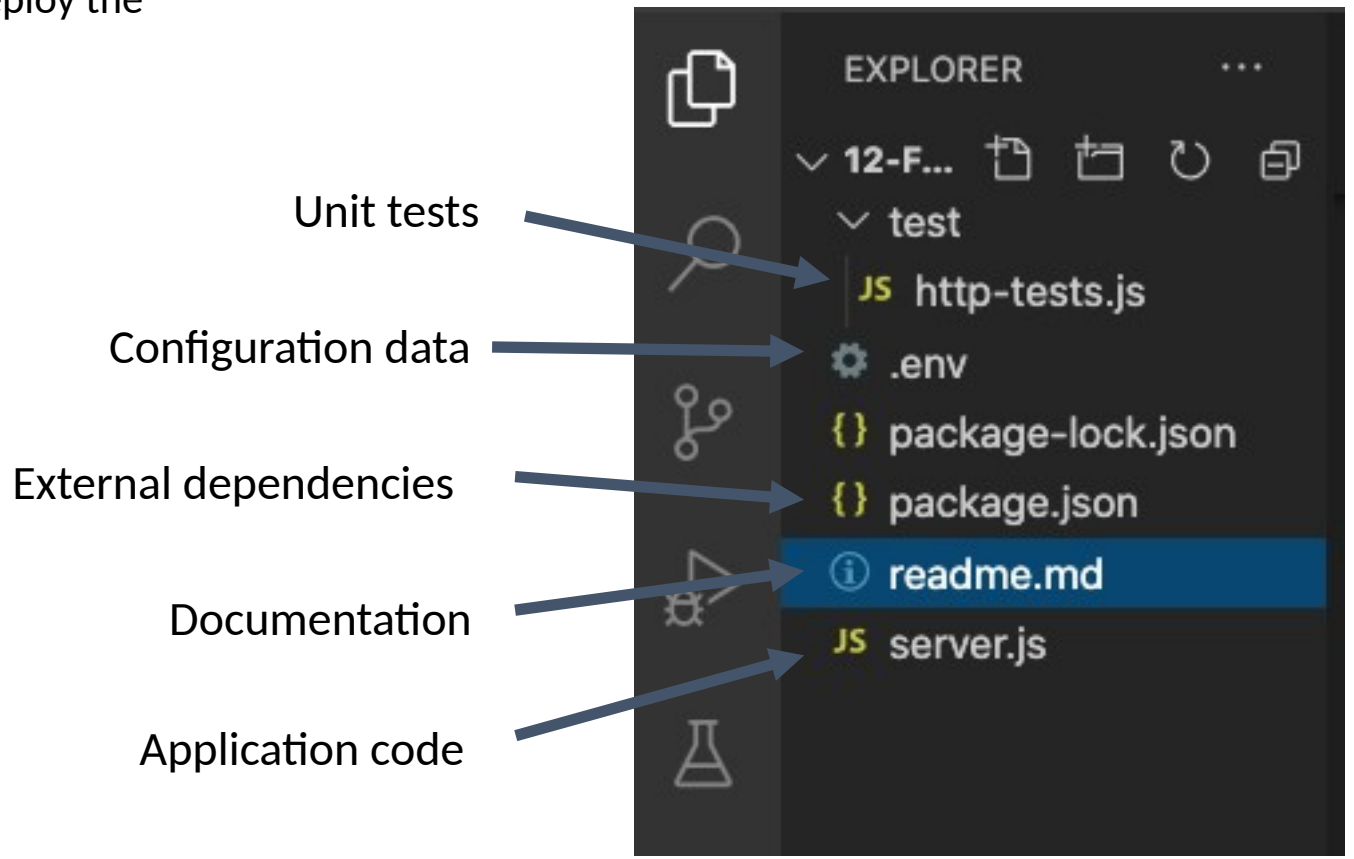
I. Codebase

One codebase needs to be defined for the enterprise. That code base is used to create, build, track, revise, control and conduct various deployments. Automation should be implemented in deployment so that everything can run in different environments without extra configuration work.



I. One Codebase, Many Deploys

The codebase contains all the artifacts necessary to build, test and deploy the application



II. Dependencies

Explicitly declare and isolate dependencies

Isolating dependencies means clearly declaring and isolating the dependencies. An app is a standalone structure, which needs to install dependencies. Whatever dependencies are required are declared in the code configuration.

```
"dependencies": {
  "apollo-link": "^1.2.11",
  "apollo-link-ws": "^1.0.17",
  "apollo-server": "^2.4.0",
  "graphql": "^14.2.1",
  "graphql-tag": "^2.10.1",
  "lodash": "^4.17.11",
  "node-fetch": "^2.3.0",
  "subscriptions-transport-ws": "^0.9.16",
  "uuid": "^3.3.2",
  "ws": "^6.2.0"
},
"devDependencies": {
  "chai": "^4.2.0",
  "faker": "^4.1.0",
  "graphql-request": "^1.8.2",
  "mocha": "^5.2.0",
  "supertest": "^3.4.2"
}
```

node: package.json

```
ordereddict==1.1

argparse==1.2.1

python-

dateutil==2.2

matplotlib==1.3.1

nose==1.3.0

numpy==1.8.0

pymongo==3.3.0

psutil>=2.0
```

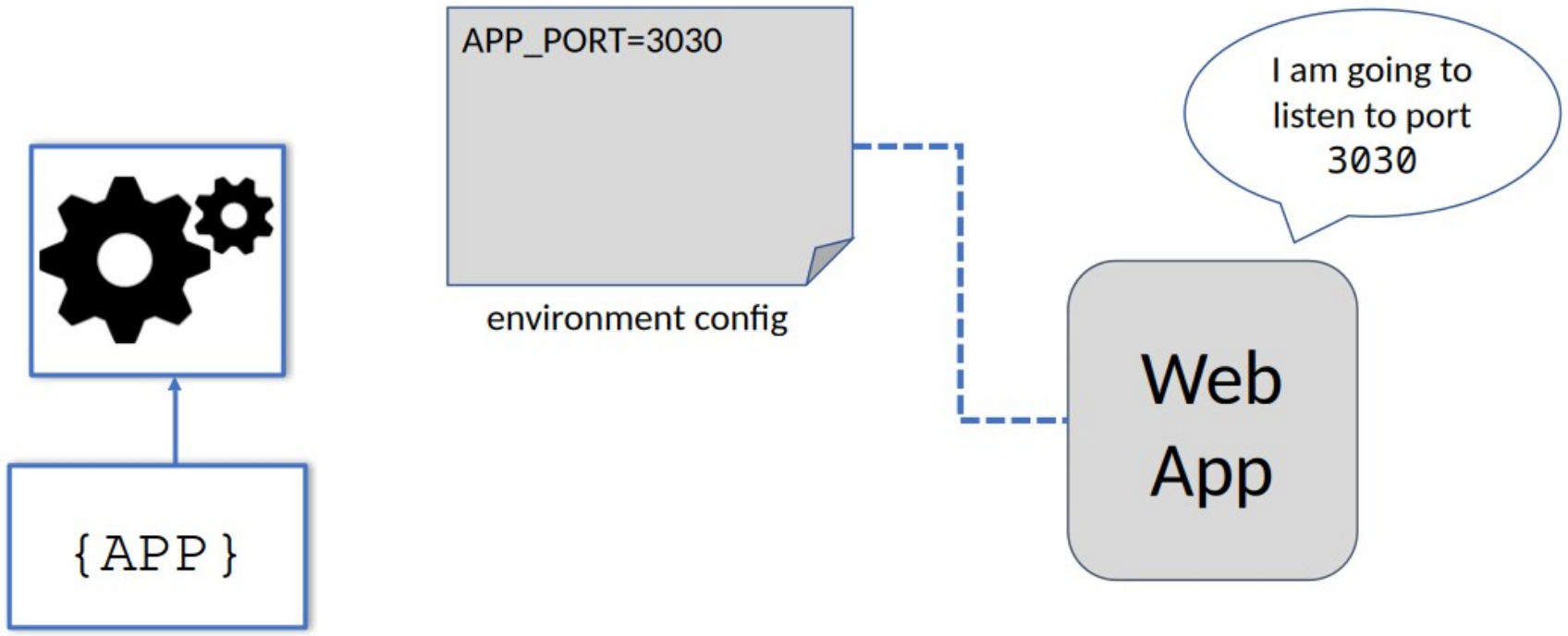
python: requirements.txt

III. Configuration

- The app configuration is anything likely to vary between deploys
  - Resource handles and references
  - Credentials
  - Environment dependent values like the hostname for the deploy
- There is no config information stored in the code
  - The code can be deployed into different environments without changes
- Litmus test:
  - If the codebase were made open source, no credentials would be compromised

III. Configuration

The concept of separation configuration from the app code can be applied to a variety of frameworks; for example, Docker Compose, Kubernetes or custom orchestration systems



III. Configuration

Examples of config files

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
 name: echocolor-red
spec:
 replicas: 1
 selector:
  matchLabels:
   app: echocolor
   color: red
 template:
  metadata:
   labels:
    app: echocolor
    color: red
  spec:
   containers:
    - name: echocolor-red
      image: reselbob/echocolor:v0.1
      ports:
       -
        containerPort: 3000
      env:
       - name: COLOR_ECHO_COLOR
         value: RED
```

Kubernetes Manifest

```
version: '3'
services:
 customer:
  build: ./customer
  ports:
   - "4000:3000"
  networks:
   - westfield_mall
  depends_on:
   - jaeger
 burgerqueen:
  build: ./burgerqueen
  networks:
   - westfield_mall
  depends_on:
   - jaeger
 hobos:
  build: ./hobos
  networks:
   - westfield_mall
 iowafried:
  build: ./iowafried
  networks:
   - westfield_mall
 payments:
  build: ./payments
  networks:
    - westfield_mall
 jaeger:
  image: jaegertracing/all-in-one:latest
  ports:
   - "6831:6831/udp"
   - "6832:6832/udp"
   - "16686:16686"
  networks:
   - westfield_mall
networks:
 westfield_mall:
```

Docker Compose

The Role of Git Repositories

- CI is based on distributed version control systems, most commonly Git
- Common Git hosting platforms
  - GitHub
  - GitLab
  - Bitbucket
- These platforms provide:
  - Centralized repositories
  - Branch management
  - Pull / Merge Requests
  - Integration with CI tools
  - Access control and audit history
- A Git repository acts as the single source of truth for the codebase

Branching and CI

- CI relies on frequent merging of branches into a shared integration branch
- Core CI principle: *“Integrate early and integrate often”*
- Each merge triggers:
  - Automated builds
  - Automated tests
  - Quality checks (linting, security scans, etc.)
- Ensures problems are detected as close to the change as possible

Branches

- What Is a branch?
  - A branch is an independent line of development that allows developers to:
  - Work on features in isolation
  - Fix bugs without disrupting stable code
  - Experiment safely
- Branches enable parallel development, critical in modern software
  - Deals with the issues raised by industrial strength software
  - And allows for workflows that support microservices and other modern architectures

Branching Strategies

- Main / trunk-based development
  - Simplest strategy
- Structure
  - Main (or trunk) branch
  - Short-lived feature branches (hours or days)
- Characteristics
  - Developers frequently merge into main
  - Feature flags are used to hide incomplete work
  - CI runs on every commit
- Benefits
  - Minimal merge conflicts
  - Fast feedback loops
  - Encourages small, incremental changes
- Commonly used by
  - High-performing DevOps teams
  - Organizations practicing Continuous Deployment

Main / Trunk-Based Development

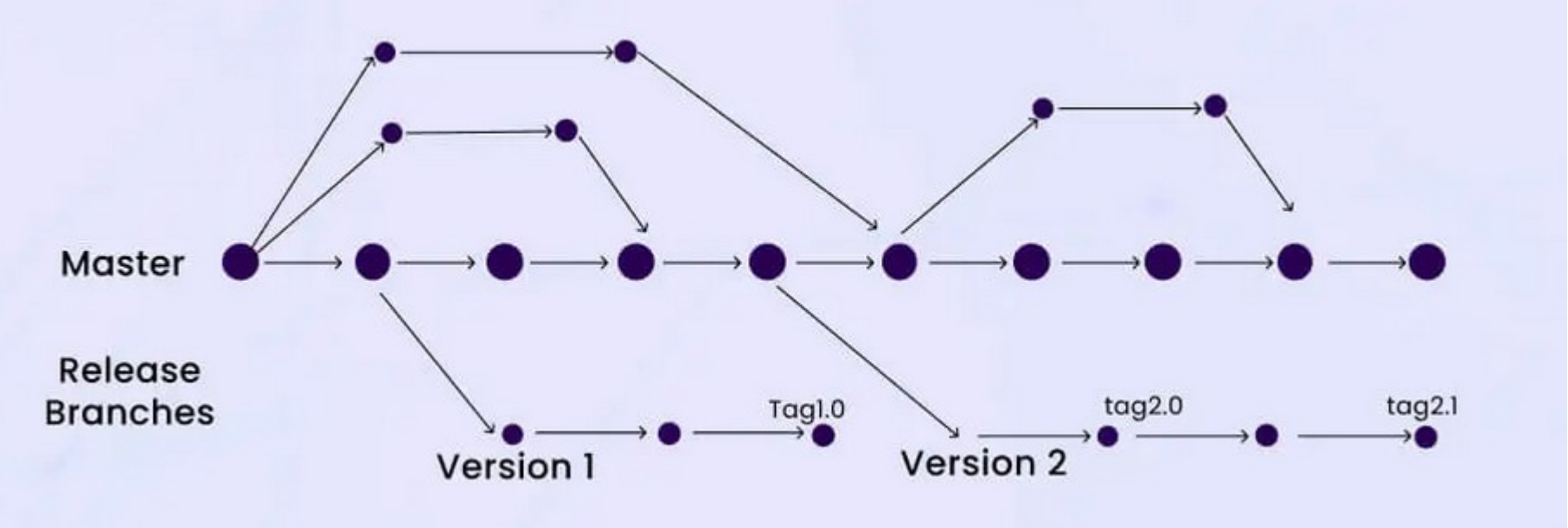


Image Credit: https://www.moontechnolabs.com/blog/trunk-based-development/

Branching Strategies

- Feature branch workflow
  - Each new feature or bug fix is developed in its own branch
- Structure
  - Main branch
  - Feature/* branches
- Workflow
  - Developer creates a feature branch
  - Code changes are committed
  - A Pull Request (PR) is opened
  - CI runs automatically on the PR
  - After review and success, the branch is merged
- CI Integration
  - Tests run on feature branches
  - Additional checks run before merging
- Benefits
  - Code review support
  - Clear separation of work
  - Easy collaboration

Feature Branching

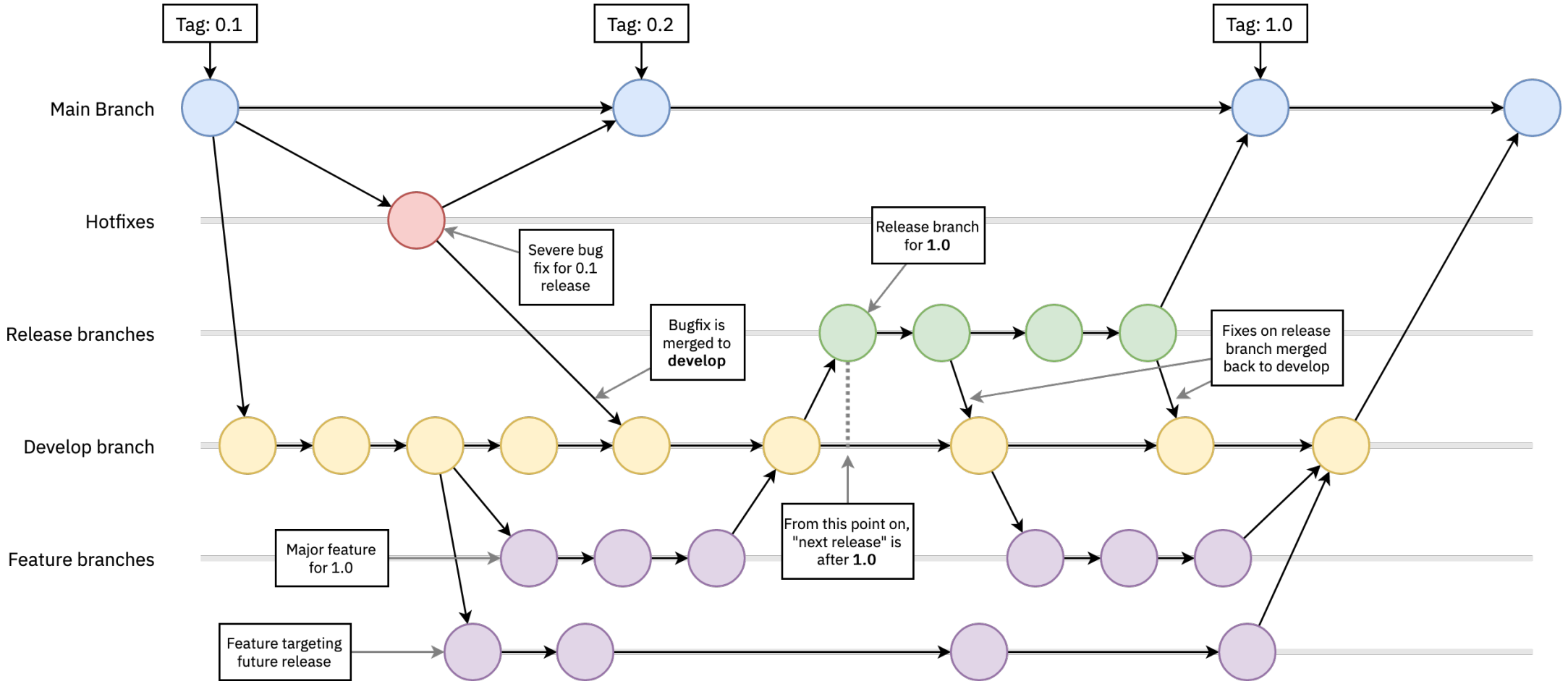


Image Credit: https://peerdh.com/blogs/programming-insights/git-branching-strategies-for-modern-development

Pull Requests and CI

- A Pull Request (PR) (or Merge Request) is a request to merge code from one branch into another
- PRs can occur automatically to
  - Build the code
  - Run tests
  - Enforce quality gates
- A PR is typically blocked from merging unless
  - All CI checks pass and all automatic tests pass
  - Required reviews are completed
  - This ensures that only validated code enters shared branches

Triggers

- CI pipeline automation is triggered by Git events
- Typical CI triggers include
  - Push to a branch
  - Opening or updating a Pull Request
  - Merging into main
- Tight integration between Git repositories and CI orchestration tools enables automation without manual intervention
  - Developer pushes code → GitHub webhook (event) → CI pipeline starts

Best Practices

- Recommended Practices
  - Keep branches short-lived
  - Merge frequently
  - Automate tests early
  - Avoid long-running feature branches
  - Use branch protection rules
  - Require CI success before merge
- Anti-Patterns
  - “Big bang” merges
  - Long-lived feature branches
  - Manual testing before integration
  - Skipping CI checks

Best Practices

- Protect Important Branches
  - Protect main (and sometimes develop) with rules:
  - Require at least 1 or 2 approvals before merging
  - Require successful CI checks before merge
  - Disallow force-pushes
- Integrate CI/CD Pipelines
  - Every commit should:
    - *Be tested automatically*
    - *Possibly trigger builds, deployments, or staging environment updates*
  - Ensures stable code reaches production

Questions