Agenda

Continuous Integration

Continuous Delivery

Pipeline As Code

GitHub Actions

Continuous Integration

What happens without Continuous Integration

Long Development Cycles

- Non-compiling code. Code may not be functional

High Build Failures/Bug Count

- Long living branches causing bigger merge effort
- Code missing from source control
- Security flaws found late
- Overall reduced quality

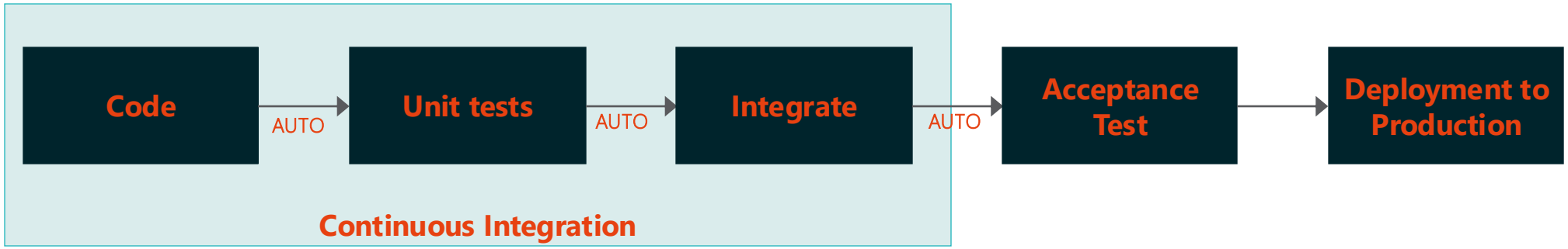
Less communication and collaboration

- Not following standards (code, docs, etc.)
- Few (or no) code reviews
- Testing done late (if done…) and mostly manual

What is Continuous Integration

CI is a mindset, team strategy and a capability. Is not a tool!

"… a **software development practice** where members of a team **integrate their work frequently**, usually each person integrates **at least daily** – leading to multiple integrations per day. Each integration is **verified by an automated build** (including test) to detect integration errors as quickly as possible. Many teams find that this **approach leads to significantly reduced integration problems** and allows a team to **develop cohesive software more rapidly**." (Martin Fowler)



Continuous Integration: Goals

1) Leverage team collaboration
2) Enable parallel development
3) Minimize integration debt
4) Act as a quality gate
5) Automate everything!

Continuous Integration: Main elements

Version Control

- Source code, configurations, scripts
- Infra, pipelines, everything as code!

Branching Strategy

- First, have one ☺
- Work with small batches
- Branch often, commit always

Automated Build

- Runs after every commit/merge
- Shift-left Testing
- Must run fast!

Continuous Integration: Are your team doing it?

Are all the developers on the team checking into trunk at least once a day? (**Trunk-based development** and **working in small batches**)

Does every change to trunk kick off a build process, including running a set of automated tests to detect regressions? (**Automated builds**)

When the build and test process fail, does the team fix the build within a few minutes, either by fixing the breakage or by reverting the change that caused the build to break? (**Focus on DevOps Metrics** [Deployment frequency, MTTR, Lead time for change])

Reference: Humble, Jez. Farley, David. (2010) Continuous delivery: Reliable software releases through build, test, and deployment automation

Continuous Integration: Practices

Mono-repos vs Multi-repos

All your code on a single repo or each component (service, microservice) on its own repo

Mono-repo: Improved Developer Testing, Reduce Code Complexity, Effective Code Reviews, Share of Common Components, Easy Refactoring

Multi-repo: Clear Ownership, Improved Scale

Branching Strategies

KISS, keep it stupid simple

Main/trunk is always buildable and (production) deployable

Keep feature/topic branches as short lived as possible

Merge to main/trunk as fast as possible

Continuous Integration: Practices

Branching policies

- Require a minimum number of reviewers on Git pull requests
- Check for linked work items
- Check for comment resolution
- Enforce a merge strategy
- Build validation
- Automatically include specific code reviewers

Git pull requests

- Promote code reviews by your peers
- Allow to define the flow of changes
- With automatics validation allow to increase (a lot) quality

Continuous Integration: Practices

Automated Builds

The way you build your product is consistent
Any change in the way you build is tracked
The build results are triaged for quality
Linking artifacts to build events for traceability
The code is valid before merging into main

Automated Builds: Common Activities

Coding Standards Checking

Download Dependent Packages

Build Code

Unit Testing

Code Coverage Analysis

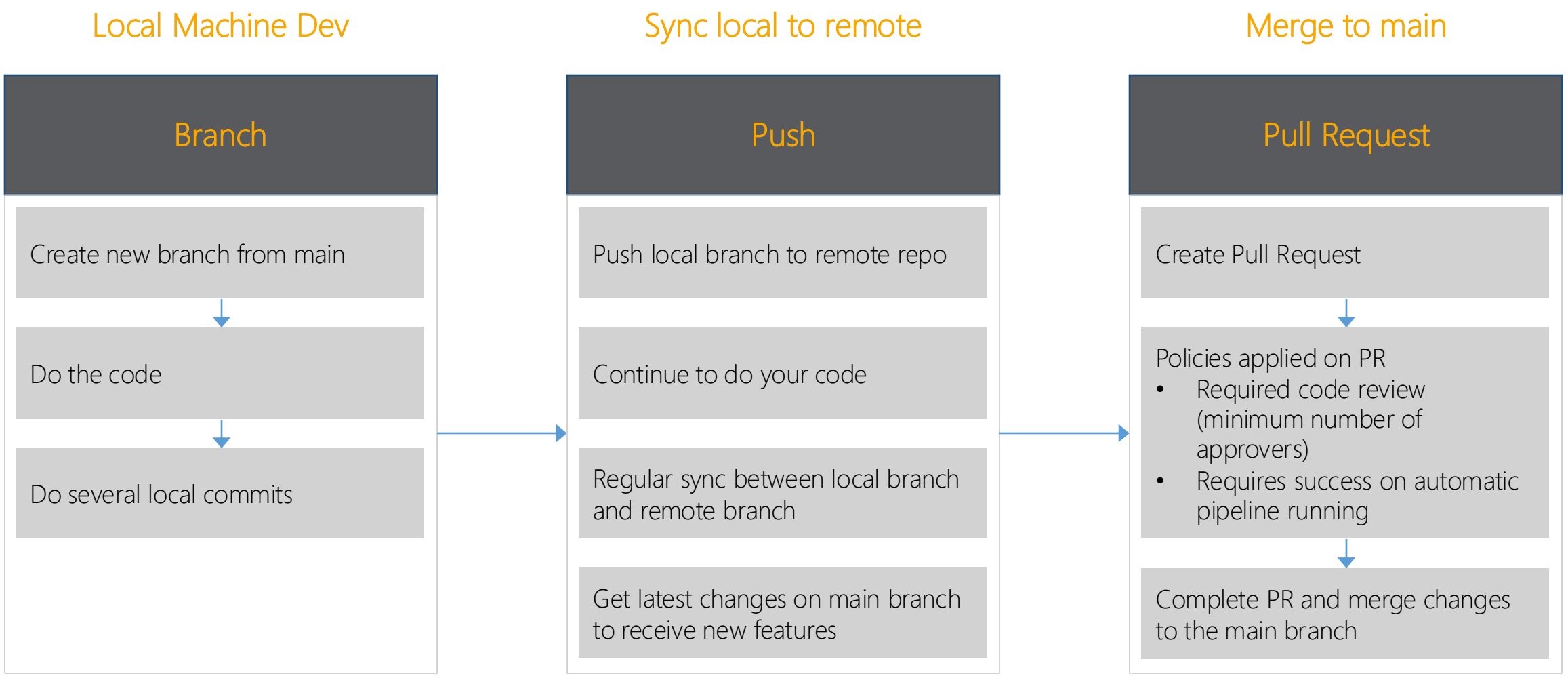
CredScan

Static Code Analysis

Open Source Component Scan

Create Deployable Package

Continuous Integration: E2E process



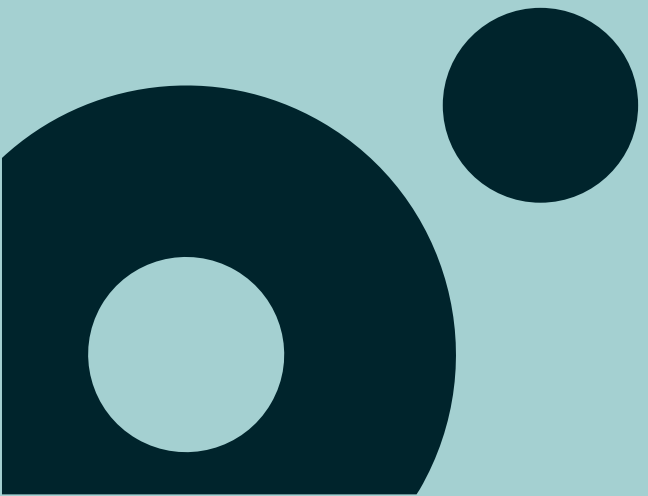
Agenda

Continuous Delivery

Practices

GitHub Actions

Continuous Delivery



What happens without Continuous Delivery

Manual and Larger Deployments

Higher risk and take a lot longer

Off-Business Hours

Manual steps document

Performed by designated operation team

Harder to rollback if failed

Larger feedback takes longer to implement

How much pain do you suffer to put a new version on production?

What is Continuous Delivery

"**Continuous delivery** is a **software engineering approach** in which teams **produce software in short cycles**, helping to ensure that the software can be **released quickly, reliably, at any time**, following a **repeatable and sustainable process**. It aims at **building, testing, and releasing software with greater speed and frequency**. The approach helps **reduce the cost, time, and risk of delivering changes** by allowing for more incremental updates to applications in production. A straightforward and repeatable deployment process is important for continuous delivery." (Martin Fowler)

Continuous Deployment vs Continuous Delivery

Continuous Deployment

MANUAL

Continuous Delivery

AUTO

What happens with Continuous Delivery

Fast, repeatable, predictable configurable deployments

Lower Risk and higher quality

Early feedback

Faster collaboration, everyone is involved, and anyone can initiate deployments

Adapt and react a lot quickly

Deploy during any business hours

Change delivered without significant delay

Outcomes of Continuous Delivery

According to the 2018 State of DevOps Report, presented by DORA, high performing DevOps companies when compared to low performers achieve:

46x deployment frequency

2555x faster lead time for changes

7x lower change failure rate

2604x faster mean time to recover

Continuous Delivery Practices

Automated Deployments

One Build Only

Move forward

Buildable and Deployable Trunk

Feature Flags

Shift-Right with Modern Deployments

Practices: Automated Deployments

Deploy process is scripted/automated using automated deployment tool (ex. GitHub Actions, Jenkins, CircleCI, …)

Processes are configurable, repeatable, idempotent and reliable

All artifacts needed for automation (besides CI produced packages) must be stored and maintained using a source control system

The process must be the same for all environments

Automated process must include quality check gates

- Automatic gates: Smoke tests, infra validation, external service check, …
- Manual gates: Human approvals

Practices: One Build Only

Continuous integration process must create a package to be delivered on all needed environments

Between environments you may only change configurations (database connection strings, certificates, services url's, …)

This approach guarantees that you delivery exactly what you tested and have direct link with your codebase on source control

Practices: Move forward

If the application breaks after a deployment, fix it, commit the fix and redeploy

Rollback must be really a last (last, last) resort

“Definition of Done” must be around of having your work deployed and working as expected, on production environment

Don’t do quick fixing on target environment. Any change must come from your CD process

Having this in mind (as a goal to your time) will automatically bring you better DevOps metrics

Practices: Buildable and Deployable Trunk

Source control as single source of truth (again)

Keep branching strategy simple

Main/trunk branch always buildable and deployable (you need to feel comfortable to start a CI/CD process anytime)

Short-lived branches (small batches)

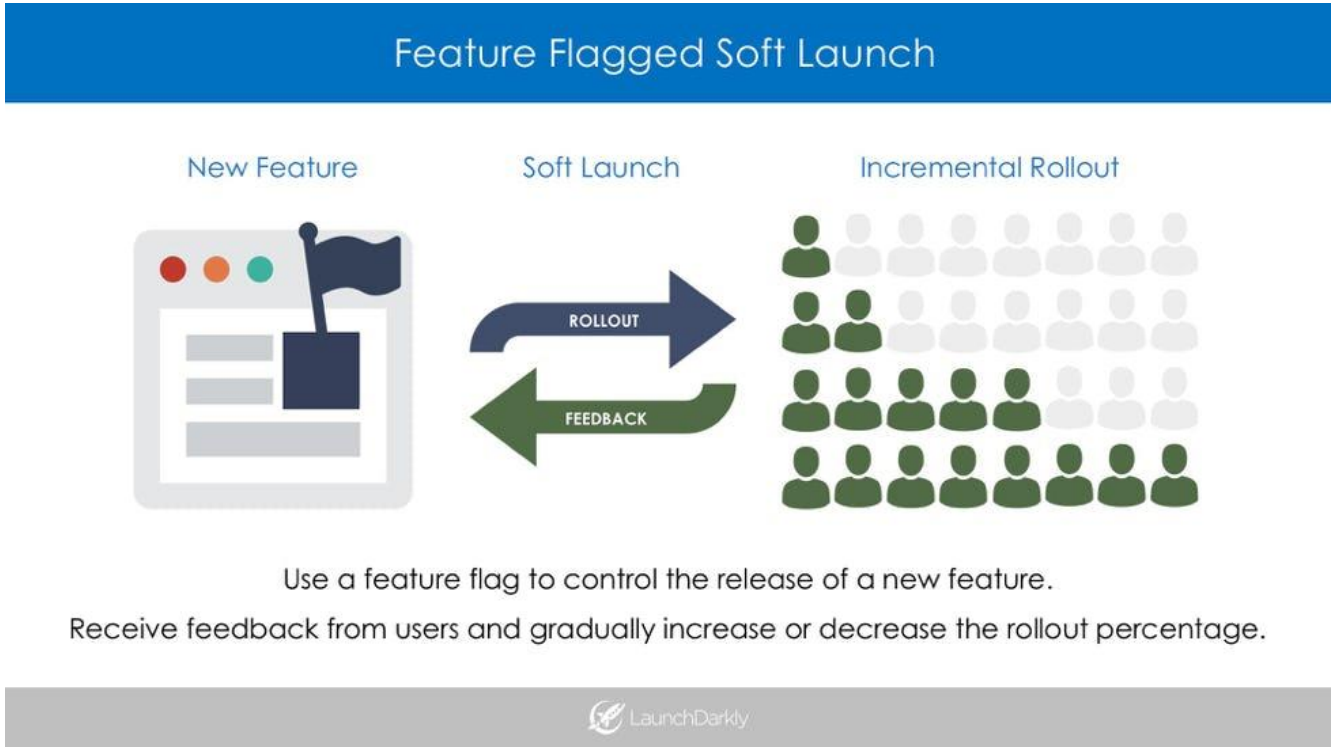
Do merge on trunk as fast as possible

Practices: Feature Flags

A check on your code that makes a feature available or not

Allow to include partial developments without manual intervention (cherry-picking)

Can be a good way to make beta/experimental feature available to a subset of your users



Practices: Shift-Right with Modern Deployments

Traditional approach is to do a “Big Bang” deployment, every user is automatically affected by new version

Before reaching production several environments are used to test your application: quality assurance environment, user acceptance test environment, load/capacity test environment, staging, …

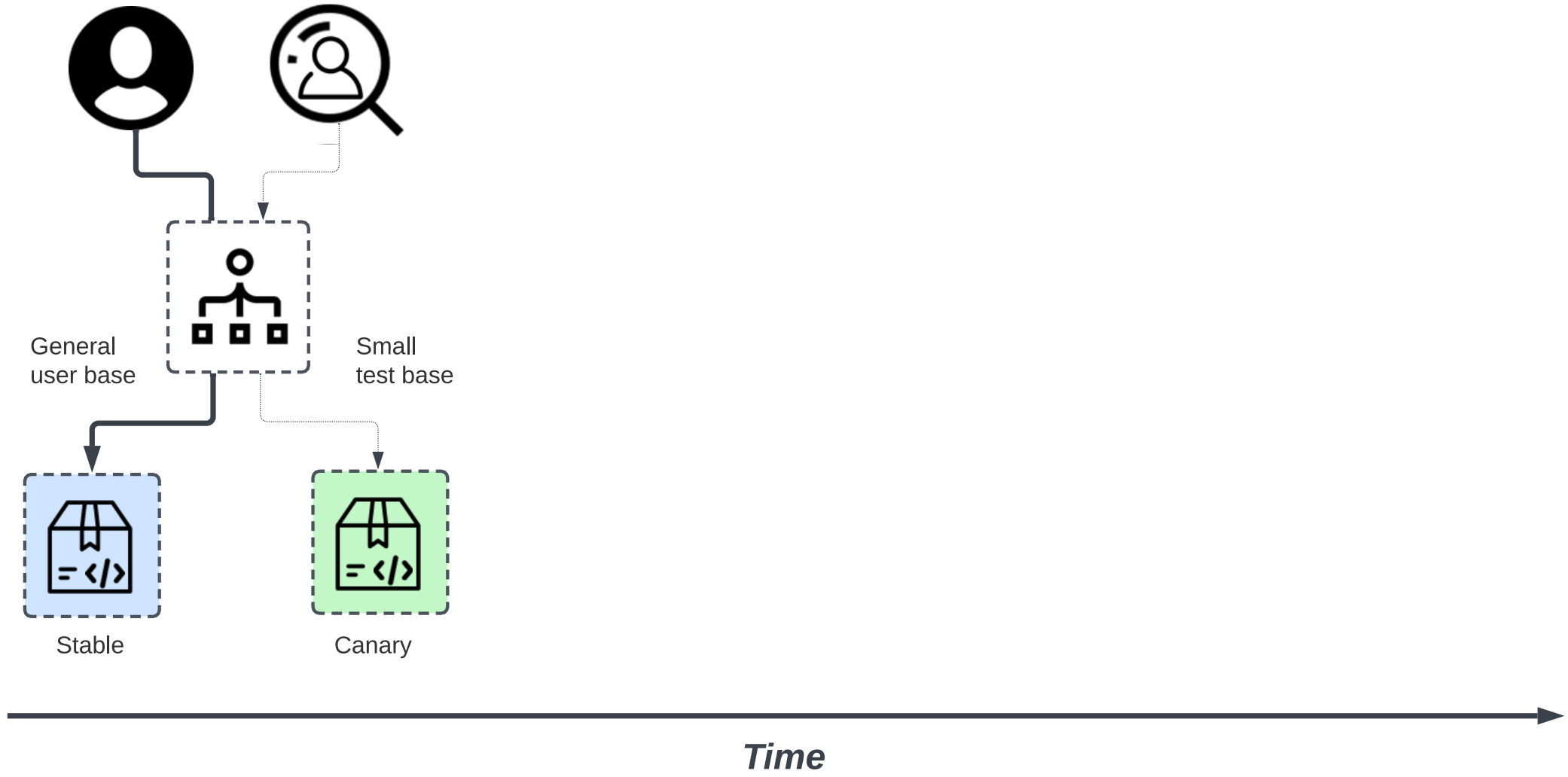
Modern approaches have the goal to reach production as soon as possible

Shift-right on testing doing it on production environment (only environment the same as production ☺)

Different techniques: Canary or Ring Deployments, Blue-green or red-black or A/B deployments

Infra as Code (detailed on next session) takes a crucial role on these techniques

Practices: Canary Deployments



Practices: Canary Deployments

Pros of Canary Deployment

Components can be **updated independently** in real-time with the safety of monitoring for performance consistency in live production before a full-scale rollout.

Least risk for platforms or services handling hundreds of thousands of users each day (gaming, banking, e-commerce, social media, and so on) where a faulty upgrade might result in significant financial losses.

"Real-world" testing on dynamic client traffic is the best possible test and only feasible on live production servers.

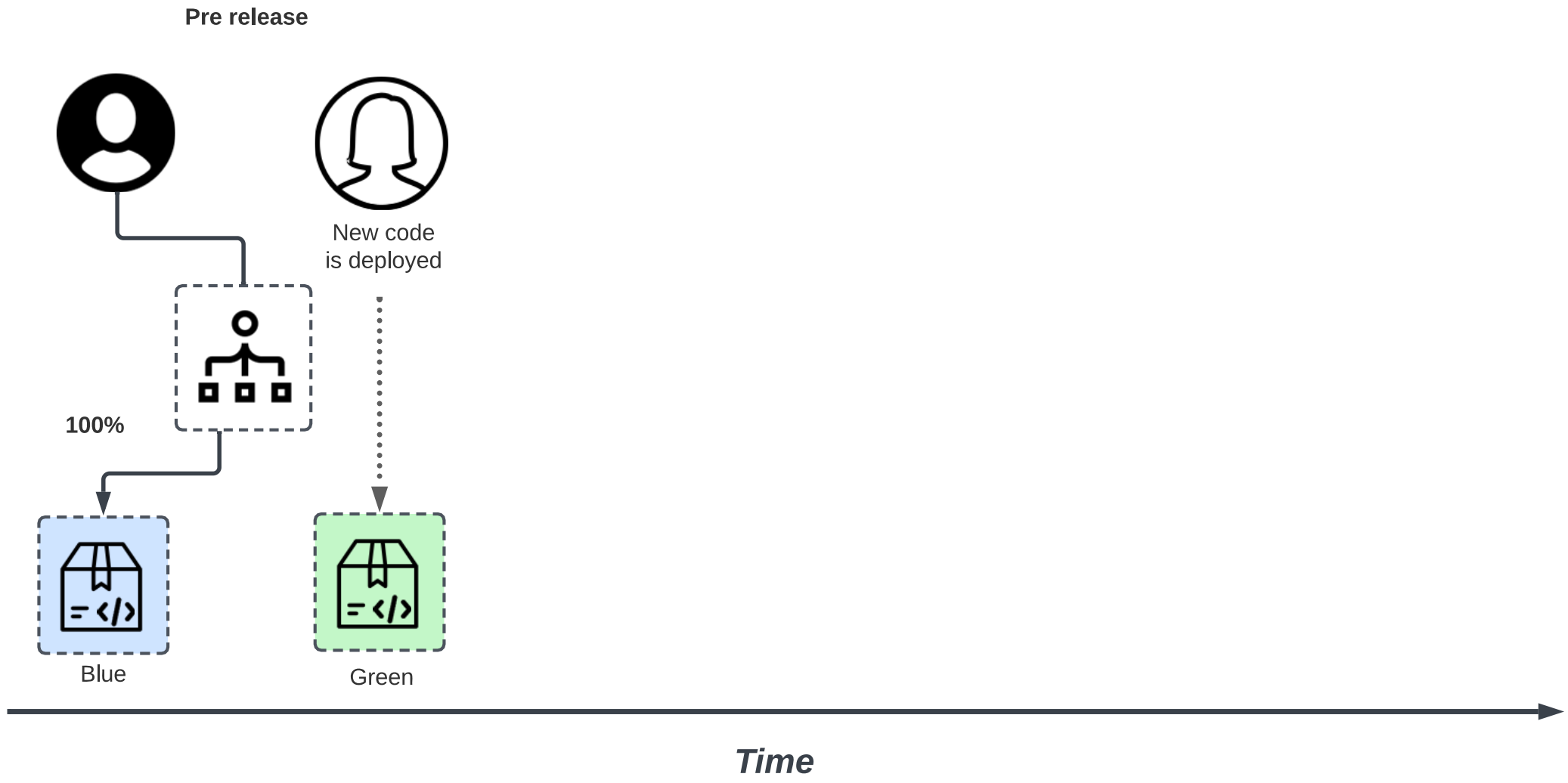
Cons of Canary Deployment

Traditional infrastructure is static and incurs the **expense of maintaining two production environments**.

Real-time analytics and insights are required for rollback decisions.

Database management might be complex since the two environments are supposed to operate simultaneously (even if they aren't receiving any traffic).

Practices: Blue/Green Deployments



Practices: Blue/Green Deployments

Pros of Blue/Green Deployment

Low risk of exposing a defective feature to public traffic because we testing the new codebase in the production-like staging environment.

Rollback is easy because we have a complete set of Production environments.

Deployment is fast and straightforward compared to other deployment strategies..

Cons of Blue/Green Deployment

High effort and cost of managing/maintaining two Production environments.

Long-lived transactions in the Blue environment make switching over to Green difficult because you have to wait to close in-flight transactions gracefully.

Database management might be complex since the two environments are supposed to operate simultaneously (even if they aren't receiving any traffic).

Hard to test experimental features to a small audience of live users.

Pipeline As Code

Pipeline As Code

Practice of defining deployment pipelines through source code

Pipeline as code is part of a larger “as code” movement that includes infrastructure as code, docs as code, etc.

Teams can configure builds, tests, and deployment in code that is trackable and stored in a centralized source repository.

Use a declarative language, like YAML (more open), or vendor-specific language (such as Jenkins and Groovy)

Tries to remove the challenges that UI interfaces bring like limited auditing, hard to versioning, difficult to rollback and prone to break on code changes

Pipeline As Code: Benefits

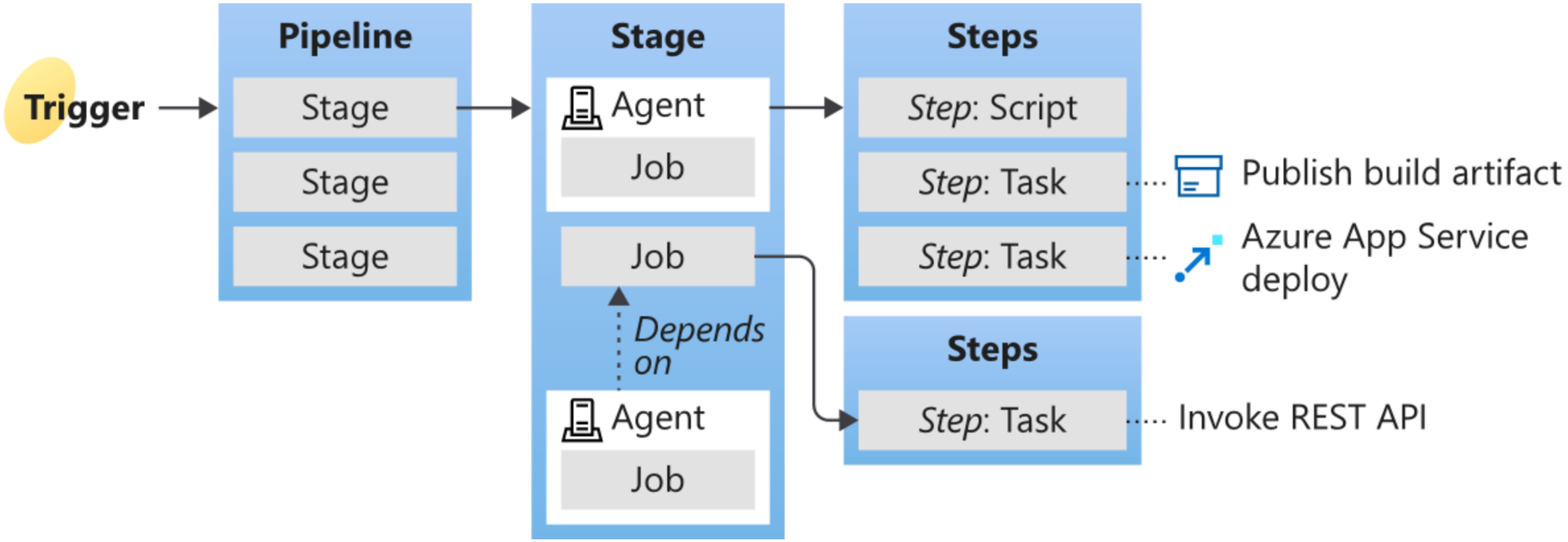
CI pipelines and application code are stored in the same source repository. All the information teams need is located in the same place.

Developers can make changes without additional permissions and can work in the tools they're already using.

Teams can collaborate more efficiently. Keeping information accessible means teams can collaborate and then act on their decisions.

Pipeline changes go through a code review process, avoiding any break in the pipeline integration.

Pipeline As Code: Basic (common) concepts



Pipeline As Code: Basic (common) concepts

A **trigger** tells a Pipeline to run.

A **pipeline** is made up of **one or more stages**

A **stage** is a **way of organizing jobs** in a pipeline and each stage can have one or more jobs.

Each **job runs on one agent**. Each agent runs a job that contains one or more steps.

A **step** can be a **task or script** and is the smallest building block of a pipeline.

An **artifact** is a collection of files or packages **published by a run**.

GitHub Actions

GitHub Actions

Automate from code to cloud

Flexible automation triggered by events

Powerful CI/CD

Any OS, any language and any cloud

Secure and automated workflows at scale

Faster innovation

GitHub Actions: Complete Flow

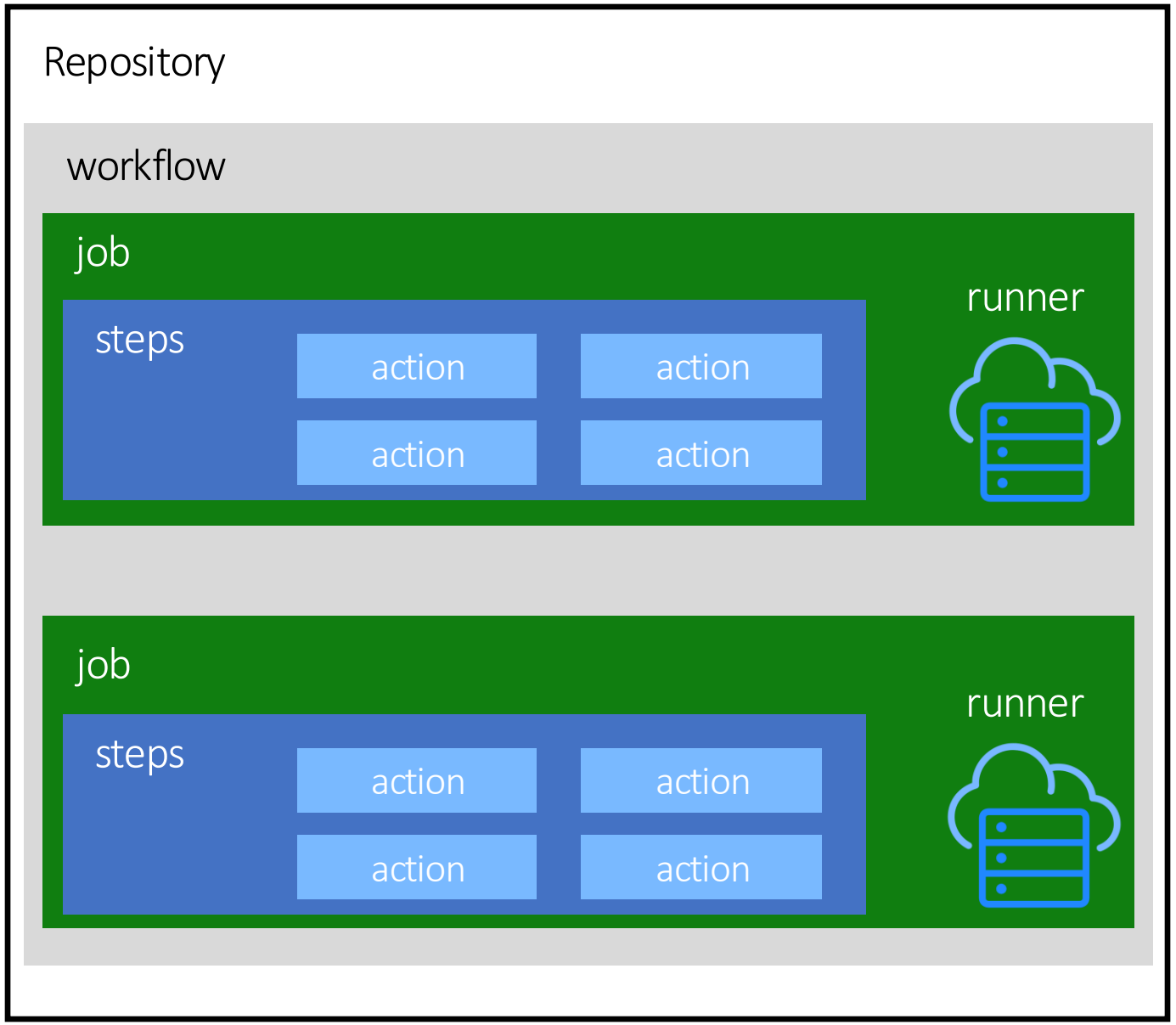
Some event happens on GitHub

Triggers a workflow

Use actions to perform tasks

Using a runner to execute

GitHub Actions: Composition



GitHub Actions: Triggers

Events that occur in your workflow's repository

Events to explicitly trigger an workflow from other workflow

Scheduled times

Manual

GitHub Actions: Workflow

Automated process available on your repository

Defined in yaml on .github/workflows folder (filename don't have a rule)

Versioned as code

Triggered by any event

- Push, Pull request, Label on PR, …
- New issue, new comment, …

Different triggers makes GitHub Actions to be much more than a CI automation tool

- Can create a "bot" behavior on your issues or PRs

Sample Workflow: Create Issue on Schedule

16 lines (16 sloc) | 396 Bytes

Raw Blame History

```yaml
on:
  schedule:
  - cron: 01 00 * * 1
name: Top 5
jobs:
  createAnIssue:
    name: Create an issue
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@master
    - name: Create an issue
      uses: bdougie/create-an-issue@e43b083ea71e22e77a81ffb4a55dacb2addb71ed
      env:
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
      with:
        args: .github/ISSUE_TEMPLATE/TOP5.md
```

Sample Workflow: Greets user on his first comment

13 lines (11 sloc) | 354 Bytes

Raw Blame

```yaml
name: Greetings

on: [pull_request, issues]

jobs:
  greeting:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/first-interaction@v1
      with:
        repo-token: ${{ secrets.GITHUB_TOKEN }}
        issue-message: 'Message that will be displayed on users'' first issue'
        pr-message: 'Message that will be displayed on users'' first pr'
```

GitHub Actions: Jobs

Defines a list of steps (actions) to be performed in sequence

A job runs on a GitHub Runner

Defines a virtual environment inside the runner

Selects the runner using tags to matching

Sharing between jobs needs to be done explicitly

GitHub Actions: Actions

Unit of work that represents one task to be performed

Can be implemented in JavaScript, using a container (Linux) or a composite (template) action

Marketplace, community or own actions

- actions/checkout
- GitHub Marketplace
- Your action available on your repo

Any action available on a repo can be selected by its version

- Git branch
- Commit SHA
- Git tag (human-readable name for a commit)

GitHub Actions: Runners

GitHub Runners available as GitHub-Hosted and self-hosted

GitHub-Hosted Runners

- Available on Windows, Linux and MacOs
- For each run, a new runner is created with clean state
- New versions (at least very week)
- Simplicity and quickness

Self-Hosted Runners

- Can be installed on Windows, Linux MacOS
- Infra and all management is done by “you”
- Control and specific scenarios

GitHub Actions: Additional Features

Matrix builds

Live logs for real-time feedback

Built-in variables, secret store and workflow artifact store

GitHub Packages to share libraries, packages and containers