universidade de aveiro
deti departamento de eletrónica, telecomunicações e informática

Software Engineering

Automate Deployments

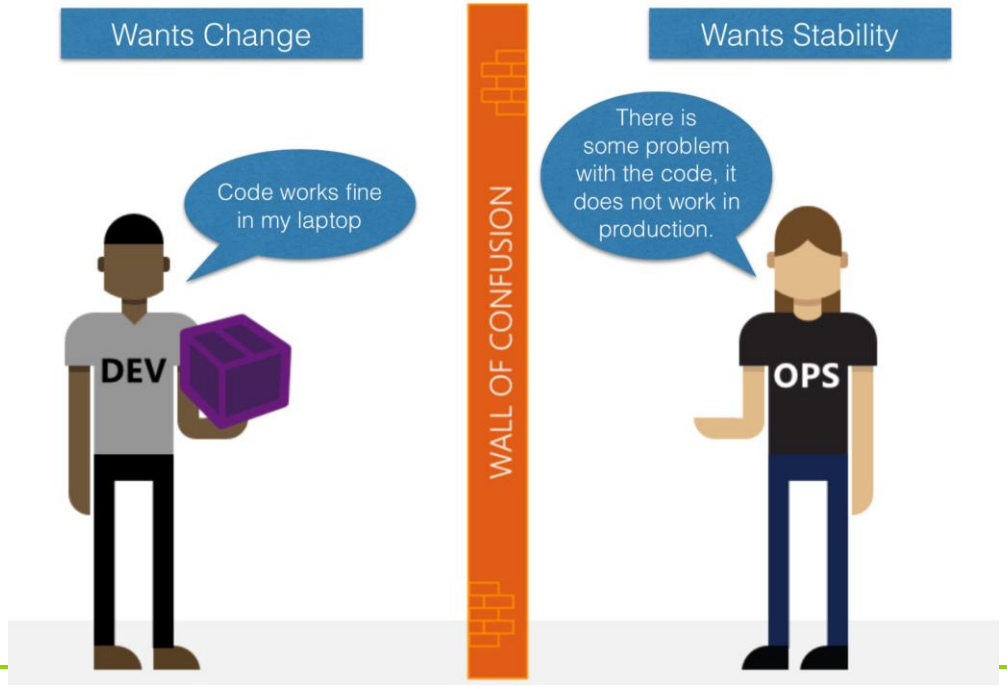
2025/2026

Outline

- Continuous Integration and Continuous Deployment
  - Definitions
- Transitions
  - From manual deployments to automatic deployment
  - The Mindshift
  - Anti-patterns and patterns
- CI-CD in Practice “with Github Actions”
  - Environment & Fundamentals
  - Workflow Triggers
  - Workflow Runners
  - Event Filters
  - Context
  - Functions

What are the limitations of Agile?

- The Development part is continuous
- The Operation is NOT continuous



https://blogs.msdn.microsoft.com/uk_faculty_connection/2016/06/23/devops-the-wall-of-confusion-understanding-the-basics-of-devops/

Dev vs Ops

- The **development** team kicks things off
  - by “throwing” a software release “over the wall” to Operations.

- **Operations** picks up the release artifacts and begins preparing for their deployment.
  - They also hand edit configuration files to reflect the production environment
    - which is **significantly different than the Development environments.**

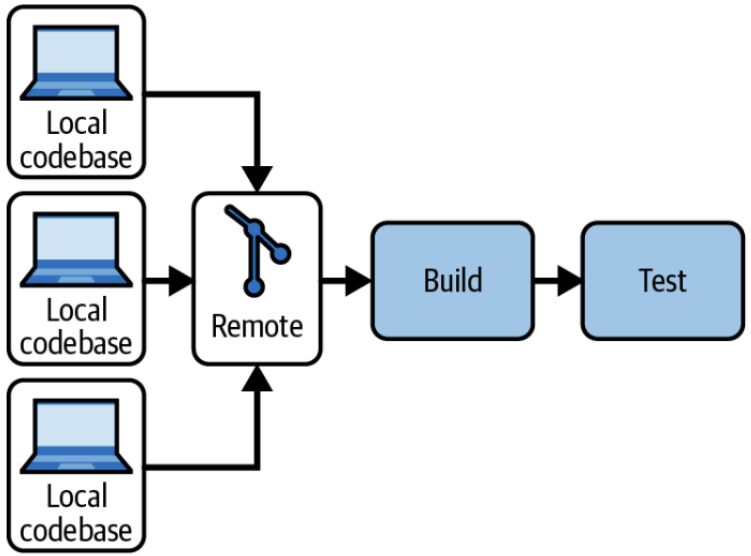
What is a pipeline?

- Definition: A pipeline in CI/CD is a structured sequence of automated steps that transform source code into a deployable software product.
- Ensures consistent, repeatable, and reliable software delivery by automating integration, testing, and deployment tasks.
- Key characteristics:
  - Sequential and/or parallel stages (build, test, deploy)
  - Automated execution triggered by events (e.g., code commit, merge)
  - Feedback loops to developers through logs, reports, or alerts
  - Environment consistency (development → staging → production)
- In other words:
  - A route from development to production
  - Triggered by different events (push, merge, etc.)
  - Different branches may take different paths

Continuous Integration & Deliver/Deployment

Continuous Integration (CI)

- It is the practice of integrating **developer changes** as **frequently** as possible in the **same version-controlled branch**.
- **CI is about ensuring that the amount of code to be merged and verified is reduced to a minimum.**



CI: Foundational Practice

- Ensures frequent integration of changes into the main branch to minimize divergence.
- Centralized, automated pipelines build, test, and validate each code increment.
- Produces artifacts that serve as authoritative, deployable units of software.
- Establishes a rapid feedback cycle, shifting “completion” to integrated and verified code.
- Reduces integration conflicts and facilitates early detection of defects.

Deploying Every Few Months or Years

- Manual, error-prone processes
- Slow integration of changes
- Builds created by hand
- Configs & dependencies outside version control
- Poorly documented, sequential deployment steps
- Manual testing for every release
- Release cycles: months or years

Figure 1-1. The typical path to production before the early 2000s

Deploying Every Few Days

- Automation at the core: build, test, deploy handled by tools
- Rationale: Human intervention often represents a bottleneck, whereas automation enables greater speed and reduces errors.
- Paradigm Shift: The focus has evolved from simply writing code to the broader goal of delivering software.

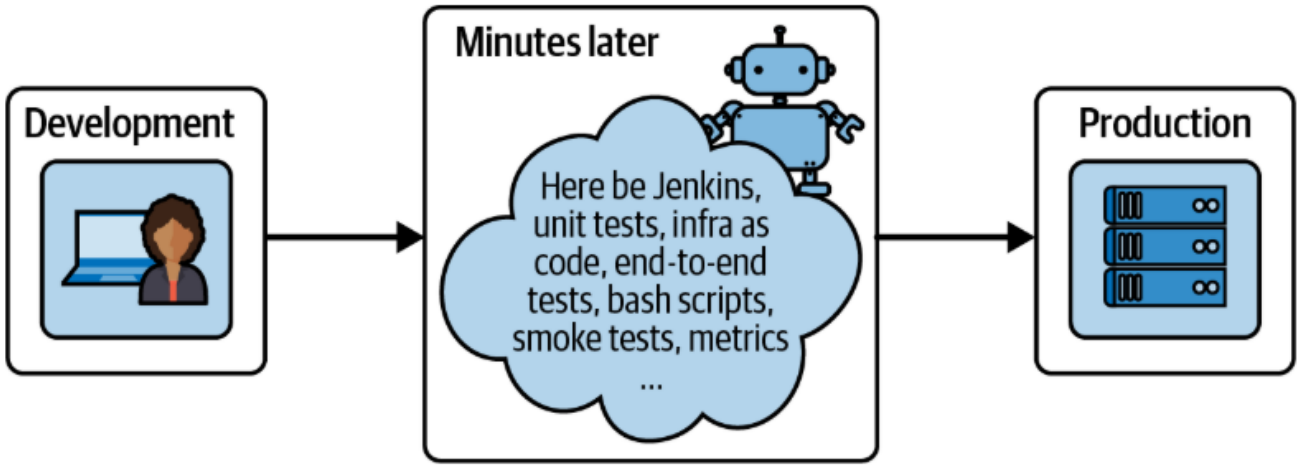


Figure 1-2. The typical path to production today

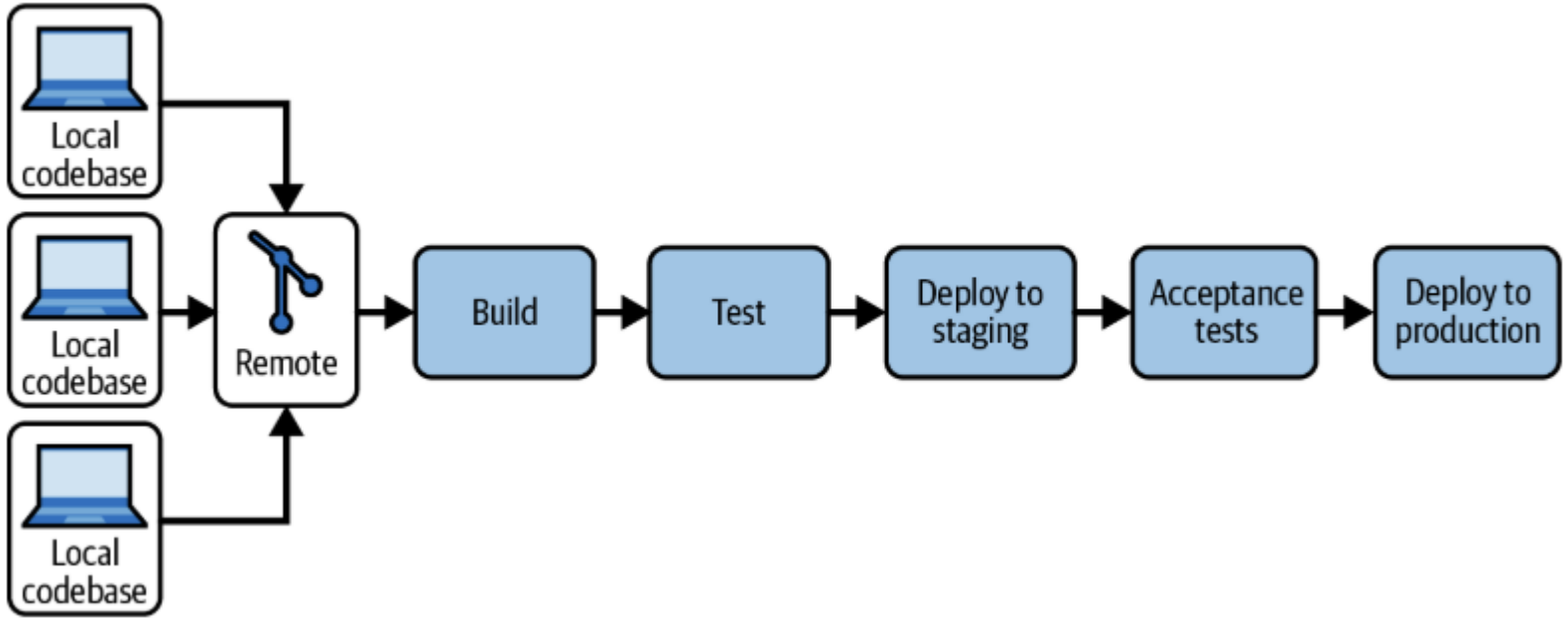
Continuous Deployment: Scope and Premise

- From “code committed” to “code in production” as a unified process
- Culmination of decades of automation in software delivery
- Builds upon XP, DevOps, Continuous Integration, and Continuous Delivery
- Emphasis on repeatability, reliability, and speed
- Positions production as the primary locus of value realization

Continuous Deployment (CD)

- Definition: the practice of **structuring your software pipeline** so that it is completely **free of manual intervention**. With this methodology, **every** code **commit** that passes its quality gates is **automatically deployed** to production.
- Continuous delivery in such a way that deployments happen without human intervention in at least some preproduction environments for every build. However, the same automation does not carry changes all the way to production. To cover that last step, a human still has to press a button in the pipeline tool to deploy the release candidate.
- The **presence** of that **manual "button"** is the **main differentiator** between **continuous delivery** and **continuous deployment.**

Continuous Integration & Continuous Deliver and/or Deployment (CI-CD)



CD: Maintaining Deployability

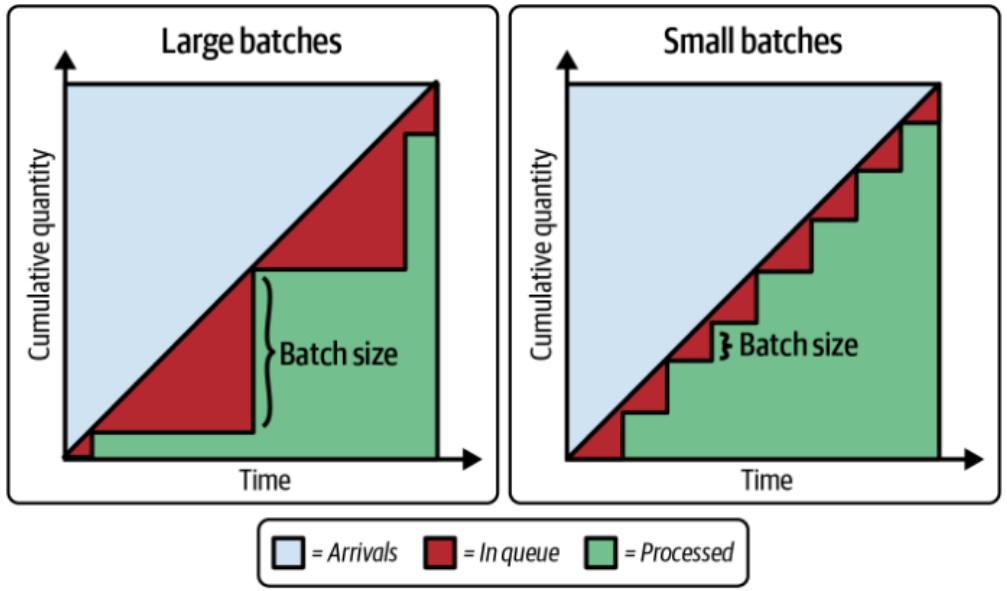
- Extends CI pipelines to support automated deployments across environments.
- Treats each commit as a potential release candidate, ready for production.
- Automated promotion to staging or UAT environments validates higher-order behaviors.
- Increases deployment frequency, thereby lowering per-release risk exposure.
- Retains a manual approval step as the final control before production deployment.

Implications and Organizational Outcomes

- Facilitates granular, low-risk deployments with immediate production feedback.
- Enables rapid detection, diagnosis, and recovery from faults in live environments.
- Accelerates experimentation and supports timely, data-driven product decisions.
- Improves key delivery metrics: lead time, throughput, defect rate, and recovery speed.
- Embeds a culture of accountability that prioritizes operational resilience and agility.

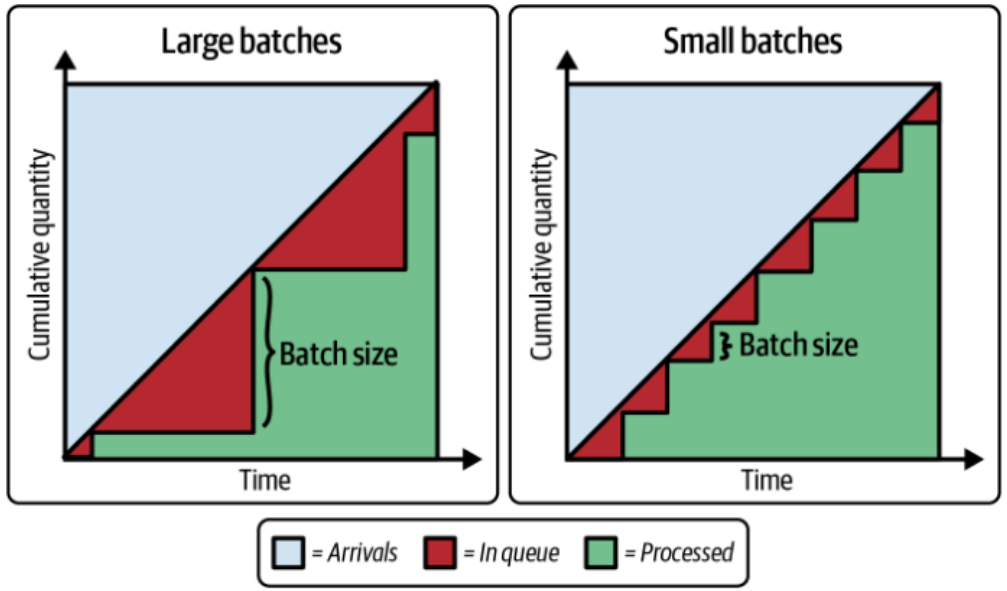
Benefits

- **Operational Efficiency and Waste Reduction**: Continuous deployment streamlines the software value stream by minimizing rework, batch sizes, and queueing, thereby improving end-to-end flow in alignment with Lean one-piece flow principles.



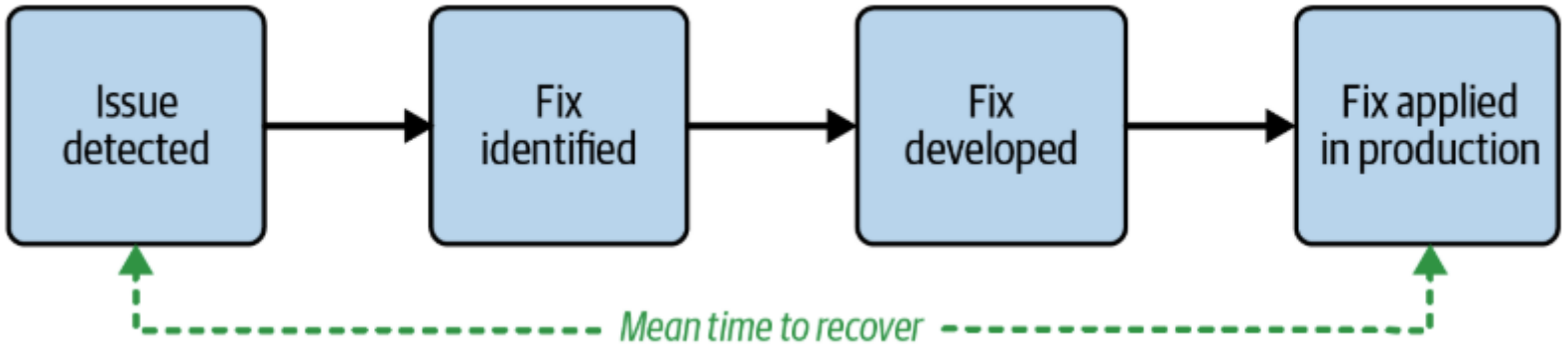
Benefits

- **Accelerated Delivery Cadence:** Lead time from code commit to production is substantially reduced, and deployment frequency increases, enabling timely delivery of increments and faster responsiveness to stakeholder needs.



Benefits

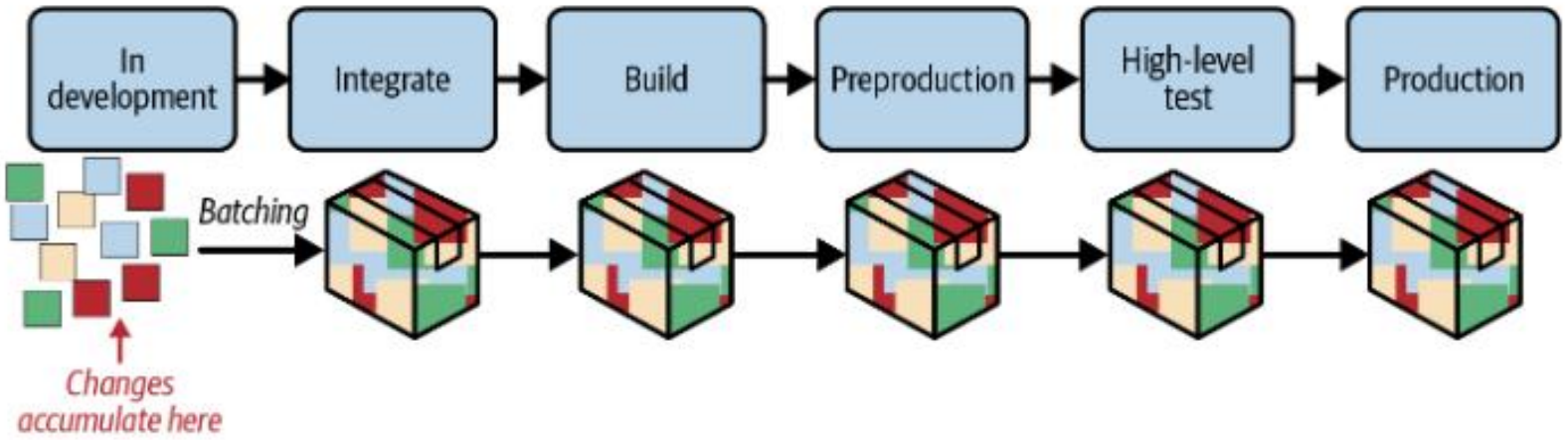
- **Enhanced System Stability**: Deploying smaller, isolated changes lowers deployment risk, reduces the change-failure rate, and facilitates rapid remediation, thereby improving mean time to recovery (MTTR).



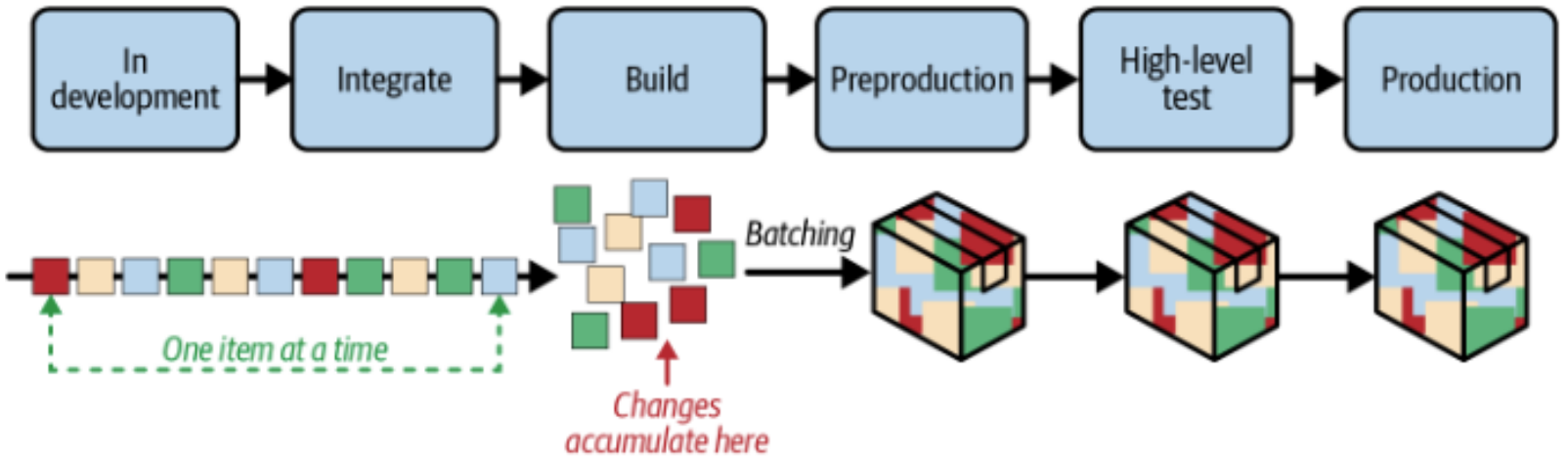
Benefits

- **Shift-Left Quality Assurance:** Quality controls testing, security checks, and other cross-functional verifications: are executed earlier and automatically within the pipeline, promoting disciplined engineering practices and consistent release readiness.
- **Strengthened Engineering Ownership:** End-to-end responsibility for changes in production reduces handovers and loss of context, fostering accountability and sustained attention to operational excellence.
- **Alignment with DORA Performance Indicators:** The practice positively influences all four key metrics, deployment frequency, lead time for changes, MTTR, and change-failure rate, supporting superior software delivery performance.

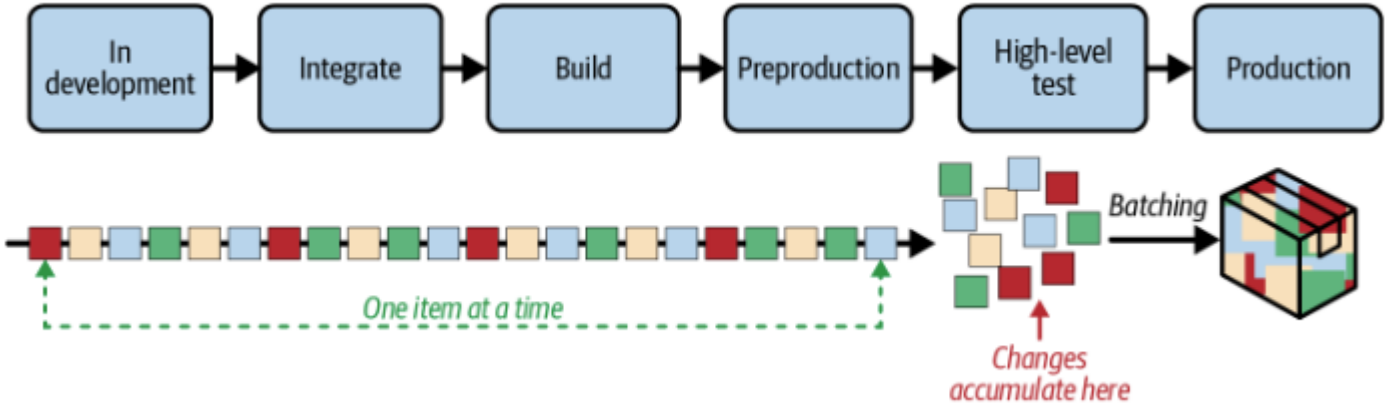
Lowering transaction cost in software



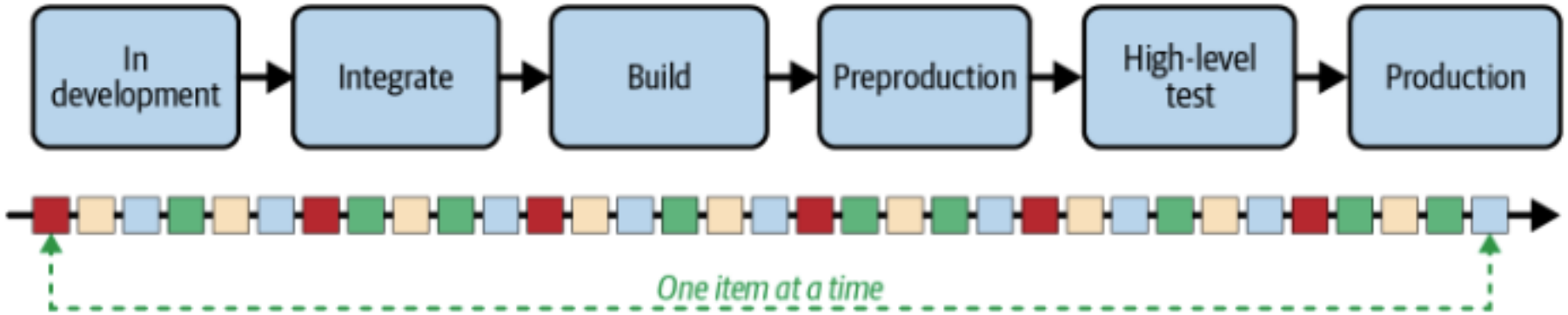
Lowering transaction cost in software



Lowering transaction cost in software

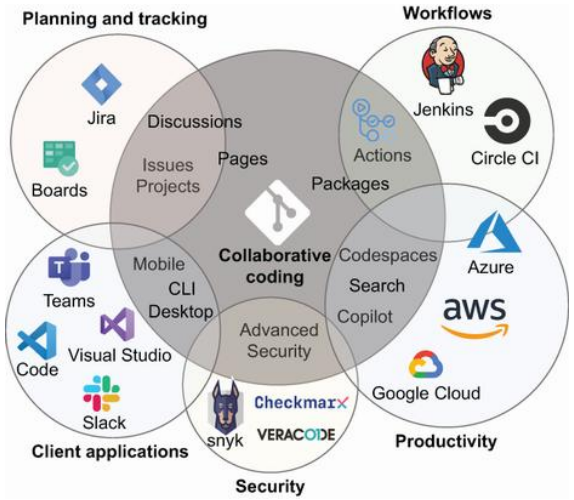


Lowering transaction cost in software



CI & CD with Github Actions

Github Actions: Fundamentals



- Benefits:
  - Automated deployments
  - Version control for deployment scripts
  - Collaboration and access control
  - Continuous integration and delivery
- GitHub Actions provides a unified automation framework inside the development lifecycle
- Enables both traditional CI/CD pipelines and ancillary automation (project management, quality assurance, collaboration)Illustrates modern trends in developer-centric automation ecosystems

Github Actions: Workflows

- Defined in YAML files under *.github/workflows/*
  - Human-readable data serialization; strict superset of JSON
  - Newlines and indentation carry meaning; no braces
  - Files typically .yml or .yaml, UTF-8 in GitHub
  - Comments start with #; inline comments allowed
  - Ideal for version control: line-oriented diffs
  - Core scalar types include integers, floats, strings, booleans, nulls, and timestamps.
  - Maps (dictionaries) and sequences (lists) are the principal collection types.
- Triggered by specific events (e.g., push, pull request, label added)
- Composed of jobs, which consist of sequential steps
- Jobs run on runners (GitHub-hosted or self-hosted; Linux, macOS, Windows)
- Jobs run in parallel by default; dependencies can be defined with needs

Github Actions: Workflow triggers

- Workflows can be triggered by a variety of repository events
- Provides fine-grained control over workflow execution based on project activities
- Well-chosen triggers reduce the need for conditional logic inside workflows and ensure that automation runs only when it is meaningful to do so.

Github Actions: Webhook Triggers

- Webhook triggers respond directly to repository activity.
  - Examples include:
    - Push events: triggered when code is pushed to a branch.
    - Pull request events: triggered when a pull request is opened, updated, or merged.
    - Issue events: triggered when an issue is opened, labeled, or closed.
- Webhook triggers can be filtered by branch names or file paths, ensuring that workflows run only when significant changes occur.
- This fine-grained control conserves computational resources and avoids redundant runs.

Github Actions: Scheduled Triggers

- Scheduled triggers are expressed using cron syntax, which consists of five fields: minute, hour, day of month, month, and day of week. They support ranges, steps, and lists, enabling complex scheduling. For example, workflows can be set to run every 15 minutes, only on Fridays at midnight, or quarterly on the first day of a month. Multiple schedules may coexist in the same workflow, allowing flexible timing.

```
on:
  schedule:
    # Runs at every 15th minute
    - cron:  '*/15 * * * *'
    # Runs every hour from 9am to 5pm
    - cron:  '0 9-17 * * *'
    # Runs every Friday at midnight
    - cron:  '0 0 * * FRI'
    # Runs every quarter (00:00 on day 1 every 3rd month)
    - cron:  '0 0 1 */3 *'
```

Github Actions: Manual Triggers

- Manual triggers give users or systems explicit control over workflow execution.
  - **workflow_dispatch** enables workflows to be started from the GitHub user interface or CLI, and it can accept typed input parameters.
  - **repository_dispatch** allows external systems to trigger workflows via the GitHub API, optionally sending custom payloads.
- These triggers are useful for controlled releases, on-demand tests, and integrations with external platforms.

Github Actions – Workflow / Hands-on (1/2)

```yaml
name: My First Workflow

on:
  push:
    branches:
      - main

  repository_dispatch:
    types: [event1, event2]

  workflow_dispatch:
    inputs:
      homedrive:
        description: 'The home drive on the machine'
        required: true
      logLevel:
        description: 'Log level'
        default: 'warning'
        type: choice
        options:
        - info
        - warning
        - debug
      tag:
        description: 'Apply tag after successfull test run'
        required: true
        type: boolean
      environment:
        description: 'Environment to run tests against'
        type: environment
        required: true
```

Activate on push to main branches

Triggered by API

```
$ curl \
  -X POST \
  -H "Accept: application/vnd.github.v3+json" \
  https://api.github.com/repos/{owner}/{repo}/dispatches \
  -d '{"event_type":"event1"}'
```

Manual inputs for workflow / action

Github Actions – Workflow / Hands-on (2/2)

```
jobs:
  MyFirstJob:
    runs-on: ubuntu-latest
```

will execute the job on the latest Ubuntu machine hosted by GitHub.

```
    steps:
      - run: |
          echo "Homedrive: ${{ inputs.homedrive }}"
          echo "Log level: ${{ inputs.logLevel }}"
          echo "Tag source: ${{ inputs.tag }}"
          echo "Environment ${{ inputs.environment }}"
        name: Workflow Inputs
        if: ${{ github.event_name == 'workflow_dispatch' }}
```

Steps will execute different type of actions. This will use different *runs*, and allows to use input parameters, using *mustache-style*.

```
      - run: |
          echo "Payload: ${{ toJSON(github.event.client_payload) }}"
        name: Payload
        if: ${{ github.event_name == 'repository_dispatch' }}


      - run: echo "👋 Hello World!"
```

Condition step, e.g. for *repository_dispatch* only

```
      - name: Checkout
        uses: actions/checkout@v4

      - name: List files in repository
        run: |
          echo "The repository ${{ github.repository }} contains th efollowing files:"
          tree
```

Actions from marketplace.
In this case, to checkout code from repo

Github Actions: Workflow more in deep

- Jobs and Runners: Jobs are the core building blocks of a workflow.
  - Each job runs on a runner, which may be GitHub-hosted or self-hosted.
  - Jobs execute in parallel by default, but dependencies can be expressed using the needs keyword, allowing complex execution graphs (e.g., build → test → deploy).
  - Runner images such as ubuntu-latest or windows-latest provide preconfigured environments, while self-hosted runners support custom setups.
- Steps and Shells
  - Steps are executed sequentially within a job.
  - Each step can either run a shell command or invoke a reusable action.
  - By default, Linux and macOS jobs use Bash, while Windows jobs use PowerShell.
  - Other shells (cmd, python, perl) can also be specified. Multi-line scripts are typically written as literal YAML blocks for clarity. Clear naming of steps improves the readability of workflow logs, making debugging easier.
- Reusable Actions
  - Reusable actions allow encapsulation of common functionality. They are referenced with the uses keyword and identified by a repository and version reference. Best practice is to pin actions to specific versions or commit hashes for stability. Local actions can be included via relative paths, and container-based actions can be pulled from registries. Inputs and environment variables allow actions to be customised, increasing flexibility and reusability.

Github Actions: Matrix Strategy

- The matrix strategy allows the same job to run across multiple parameter combinations (for example, operating systems, language versions, or toolchains).
- GitHub Actions generates one job per combination, providing broader test coverage.
- Options such as fail-fast and max-parallel control whether failures stop other jobs and how many jobs run simultaneously.
- This approach is essential for projects that support multiple environments.

```yaml
jobs:
  job_1:
    strategy:
      fail-fast: false
      max-parallel: 3
      matrix:
        os_version: [macos-latest, ubuntu-latest]
        node_version: [12, 14, 16]

    name: My first job
    runs-on: ${{ matrix.os_version }}
    steps:
      - uses: actions/setup-node@v3.6.0
        with:
          node-version: ${{ matrix.node_version }}
```

Github Actions: Expressions

- Expressions, written as ${{ … }}, allow conditional logic and access to runtime information.
- They can reference contexts such as:
  - github (event data and metadata),
  - env (environment variables),
  - matrix (parameters from a matrix build),
  - inputs (user-supplied values),
  - needs (outputs from dependent jobs).
- Expressions are used in conditions (if), dynamic naming, and parameter passing.
- Their use should remain clear and avoid unnecessary complexity.

Github Actions: Functions (1/2)

- GitHub provides built-in functions such as *contains, startsWith, endsWith, join, toJSON, and hashFiles*. These assist in working with strings, arrays, and structured data. Additionally, status functions such as *success*(), *failure*(), *cancelled*(), and always() allow workflows to branch based on job outcomes. These functions help implement robust workflows that adapt to real-world conditions.

Function	Description	Examples
`contains (search, item)`	Returns `true` if `search` contains `item`	`contains('Hello world', 'llo')` returns `true`. `contains(github.event.issue.labels.*.name, 'bug')` returns `true` if the issue related to the event has a label `bug`.
`startsWith (search, iten)`	Returns `true` when `search` starts with `item`	
`endsWith (search, item)`	Returns `true` when `search` ends with `item`	
`format (string, v0, v1, ...)`	Replaces values in the `string`	`format('Hello {0} {1} {2}', 'Mona', 'the', 'Octocat')` returns `'Hello Mona the Octocat'`.

Github Actions: Functions (2/2)

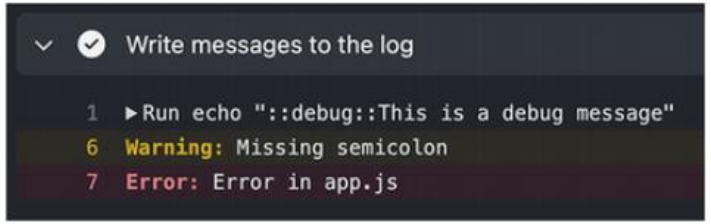
- GitHub provides built-in functions such as *contains, startsWith, endsWith, join, toJSON, and hashFiles*.
- These assist in working with strings, arrays, and structured data.
- Additionally, status functions such as *success*(), *failure*(), *cancelled*(), and always() allow workflows to branch based on job outcomes.
- These functions help implement robust workflows that adapt to real-world conditions.

`join (array, optS)`	All values in `array` are concatenated into a string. If you provide the optional separator `optS`, it is inserted between the concatenated values. Otherwise, the default separator `,` is used.
`toJSON(value)`	Returns a pretty-print JSON representation of `value`
`fromJSON(value)`	Returns a JSON object or JSON data type for `value`
`hashFiles(path)`	Returns a single hash for the set of files that matches the `path` pattern

Github Actions: Workflow commands

- Workflow commands enable steps to interact with the runner.
- Examples include logging messages, creating warnings or errors with file references, and passing outputs to later steps.
- Outputs are written to $GITHUB_OUTPUT and read by other steps.
- Environment variables can be persisted across steps by writing to $GITHUB_ENV.
- These mechanisms are fundamental for sharing data and maintaining traceability in workflow execution.

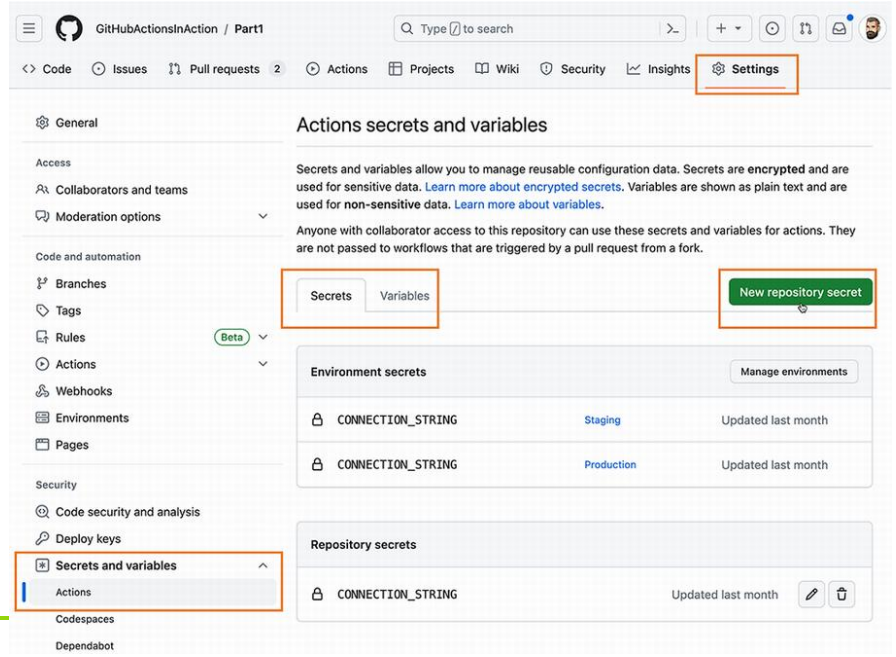
```
- run: echo "::error file=app.js,line=1::Missing semicolon"
```



Github Actions: Secrets and variables

- Secrets store sensitive data such as API keys and are encrypted at rest.
- They are masked in logs and can only be decrypted at runtime.
- Variables store non-sensitive configuration values.
- Both secrets and variables can be defined at the organization, repository, or environment level, with values at lower levels overriding higher ones.
- Following strict naming conventions and limiting scope is essential for security and maintainability.

```yaml
- name: Set secret and var as input
  uses: ActionsInAction/HelloWorld@v1
  with:
    MY_SECRET: ${{ secrets.secret-name }}
    MY_VAR: ${{ vars.variable-name }}
```



Github Actions: Debugging

- Workflow development requires careful iteration.
- In greenfield repositories, workflows can be authored directly in the main branch.
- In active projects, it is safer to work in branches or forks to avoid disrupting the team.
- Manual triggers allow iterative testing.
- Debugging can be enhanced with verbose logging by setting environment variables (ACTIONS_STEP_DEBUG, ACTIONS_RUNNER_DEBUG).
- Linting tools and editor extensions, such as actionlint or Visual Studio Code plugins, further improve workflow correctness.

What is an Action?

- An action is a self-contained, reusable step that can be invoked from a workflow. It encapsulates logic, dependencies, configuration, and an interface for inputs and outputs.
- Actions promote consistency and reduce duplication by allowing teams to standardise common tasks such as checking out source code, setting up toolchains, running linters, or performing deployments.
- Every action is described by a manifest file named action.yml (or action.yaml).
  - This file declares the action's name, description, author, branding, inputs, outputs, and the execution model under the runs section.
  - Because the filename is fixed, each action resides either at the repository root or in its own subfolder.
  - The manifest is the contract between the action and its consumers and should be treated as a stable interface.

Github Actions: Action

- There are three different types of actions:
  - *Docker container actions*
  - *JavaScript actions*
  - *Composite actions*
- Docker container actions only run on Linux, not on Windows or macOS.
- Docker container actions can retrieve an image from a Docker library, like Docker Hub, or build a Dockerfile.
- JavaScript actions run directly on the runner using NodeJS and are faster than container actions.
- Composite actions are a wrapper for other steps or actions.

Github Actions: Action

```
name: 'Your name here'
description: 'Provide a description here'
author: 'Your name or organization here'
inputs:
  input_one:
    description: 'Some info passed to the container
    required: false
  input_two:
    default: 'some default value'
    description: 'Some info passed to the container'
    required: false
runs:                                                    ❶
  using: 'docker'
  image: 'docker://ghcr.io/wulfland/container-demo:latest'
  args:                                                  ❷
    - ${{ inputs.input_one }}
    - ${{ inputs.input_two }}
  env:                                                   ❸
      VARIABLE1: ${{ inputs.input_one }}
      VARIABLE2: ${{ inputs.input_two }}
```

❶ runs: defines the type of action—here, this is the Docker image.

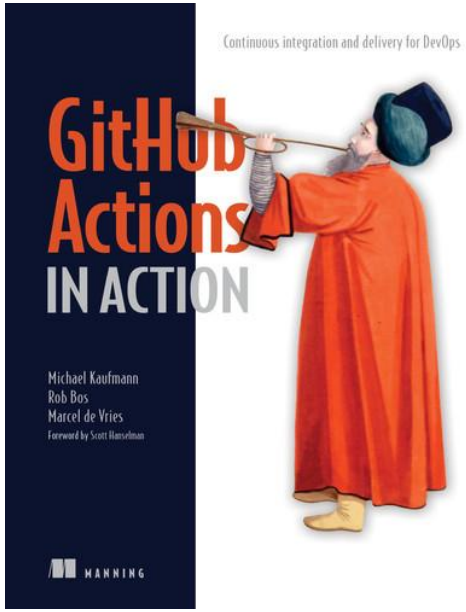
❷ args: you can pass in inputs as arguments to Docker when the container is created.

❸ env: you can pass in inputs as environment variables to be available at container run time.

Practical Guide

Practical Guide

- Resources to this lab:
  - https://github.com/GitHubActionsInAction/ActionInAction?tab=readme-ov-file
  - Hands On: https://github.com/GitHubActionsInAction



Practical Guide

- In your project:
  - Implement a pipeline to build your software components (e.g. backend, frontend,…)
    - Use Docker multi-stage, if applicable
    - Run tests (only boilerplate, is ok)
  - Implement a pipeline to deploy your software components
    - You can use local runners
      - https://docs.github.com/en/actions/concepts/runners/self-hosted-runners
    - It can be triggered on push to main, or dev, at least
    - It should build your projects, and update containers

Bibliography