DEGGENDORF INSTITUTE OF TECHNOLOGY

Software Engineering

Prof. Dr. Christoph Schober

Department of Applied Computer Science,
Deggendorf Institute of Technology

christoph.schober@th-deg.de

CI / CD

Examples

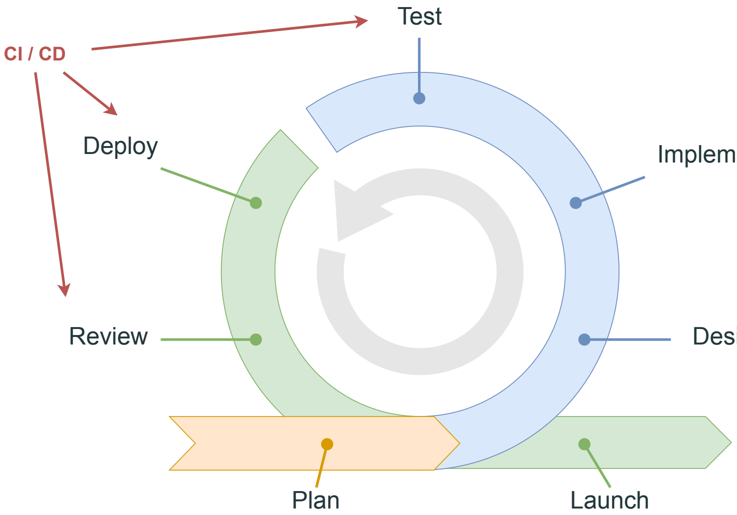
Find code examples for this lecture under:

`https://mygit.th-deg.de/aw-public/ci-cd-examples/`

CI / CD

1. Motivation

Motivation



Motivation

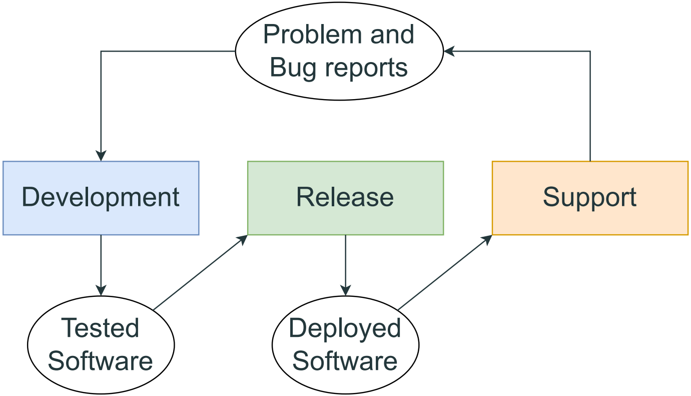
Traditionally, separate teams were responsible for:

- **Development**: Team that programs the software and passes a *final* version to the release team.
- **Release**: Team that builds the release version, performs quality assurance[1], and prepares release documentation before shipping to the customers.
- **Support**: Team that provides (1st-, 2nd-, 3rd-level) customer support.

→ Silos between development and operations.

[1] By extensive manual and automated testing

Motivation


Problem and Bug reports
Development
Release
Support
Tested Software
Deployed Software

Motivation

With the rise of agile methods, the traditional model for releasing a product to customers could not cope any longer.

→ The traditional release process introduced a bottleneck.

CI / CD

2. DevOps

DevOps

The only sustainable competitive advantage is an organization’s ability to learn and adapt faster that the competition

(Mark Schwartz, 2017)

DevOps

DevOps

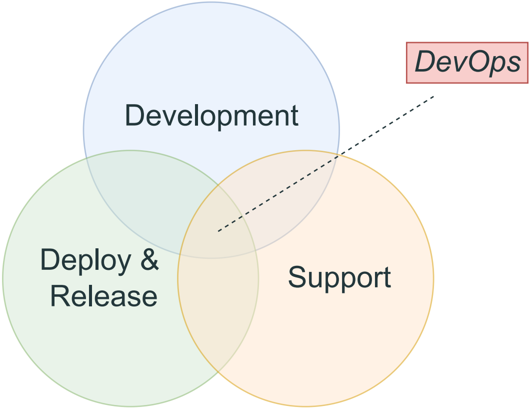
Approach that emphasizes collaboration to integrate development, deployment, and support, with a single team responsible for all these activities.

Working in a DevOps team:

- Everyone is responsible for everything.
- Everything that can be *automated* should be automated.
- Credo: *Measure first*, change later.

DevOps

DevOps seeks to break the silos:



DevOps

Passionate

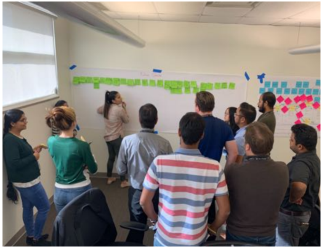
Bring it

Selfless

Share it

Accountable

Own it



DevOps

Common tasks that are **automated** in DevOps teams:

- Code compilation and building.
- Software testing, including unit-, integration, and e2e testing.
- Security testing and compliance checks.
- Software packaging.
- Deployment to testing, staging, and production environments.
- Configuration management, including infrastructure provisioning.
- Continuous monitoring and logging of application and infrastructure.

Key Performance Indicatior (KPI)

Measurable value used to assess effectiveness and success of a particular process, project, or product (in context of software engineering).

- KPIs can measure various aspects of software engineering, such as productivity, quality, efficiency, and customer satisfaction.
- By tracking KPIs, software engineering teams can identify areas where they are performing well and areas where they need to improve.
- Examples: code written per day, defect density, project completion time.

DevOps

Common KPIs **measured** by DevOps teams:

- *Deployment Frequency*: How often can a team deploy code to production?
- *Cycle Times*: How fast can a team deliver new features and fixes?
- *Mean Time to Detect (MTTD)*: How effective is the teams monitoring?
- *Mean Time to Recover (MTTR)*: How effective is the incident response process?
- *Change Failure Rate*: How stable and reliable is the teams code?
- *Deployment Success Rate*: How effective is the teams quality assurance?
- *Availability*: How reliable is the applications' infrastructure?

DevOps

Example

In 2020, American Airlines turned to the DevOps approach and published their results[2].

[2] https://github.com/devopsenterprise/2020-Las-Vegas-Virtual/raw/main/Ross%20Clanton%26MayaLeibman-%20DOES%202020%20-%20Final.pdf

DevOps

They measured and monitored the following KPIs:

- Development cycle time
- Deployment frequency
- Deployment cycle time
- Number of incidents
- Mean time to recover

DevOps

Experiences & Results:

- It took the company 3 years to embrace DevOps.
- Rapid design sessions helped teams arrive quickly at a minimum viable product, which led to a 145% increase in boarding pass scans to start check-in sessions and a 57% increase of the prepaid bag functionality.
- Deployment of 2,100 kiosks in 230 airports in six weeks.

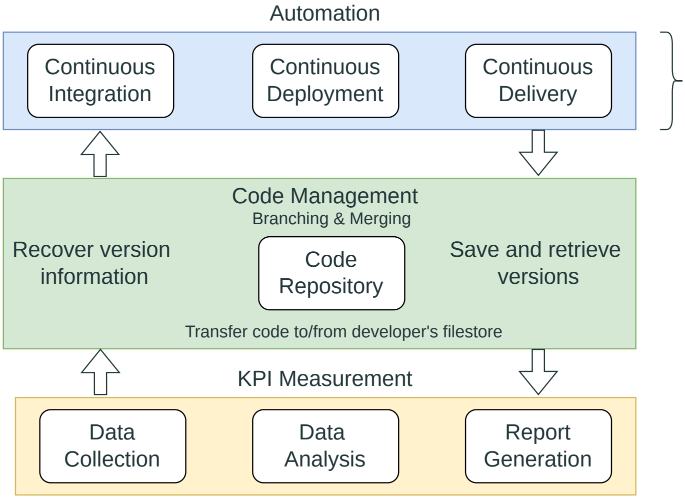
DevOps

General benefits of DevOps enabled teams:

- **Faster deployment** as reduced inter-team communication delays allow the software to be deployed to production more quickly.
- **Reduced risk** as the increment of functionality in each release is small, so there is less chance of feature interactions that software failures.
- **Faster repair** as DevOps teams work together to get the software up and running again as soon as possible.
- **More productive teams** as DevOps teams are measurably happier and more productive than teams involved in the separate activities.

→ Ok fine, but how about methods?

DevOps



Focus of this lecture

CI / CD

3. Definition

Definition

Continuous Integration

Continuous integration (CI) is the practice of automating the integration of code changes from multiple contributors into a single software product.

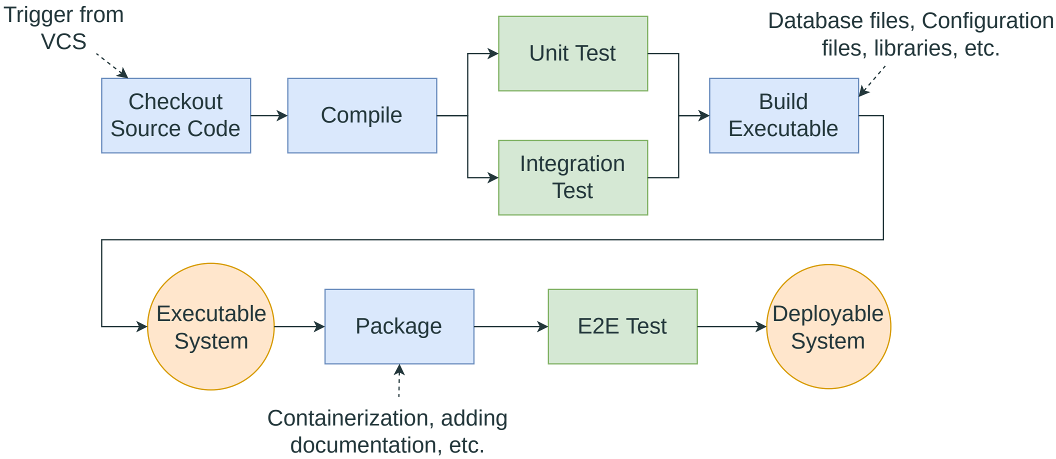
CI in context of the SDLC:

- CI starts when changes are pushed to the VCS[3].
- CI ends when a deployable system is made available.
- CI automates all the steps in between.

[3] Version Control System, e.g. git

Definition

Typical activities of a CI pipeline:



Definition

Continuous Delivery

Continuous Delivery is a step beyond *Continuous Integration.* Not only is the application built and tested each time a code change is pushed to the codebase, the application is also automatically deployed to a productive environment.

→ With continuous delivery, the deployments are triggered manually.

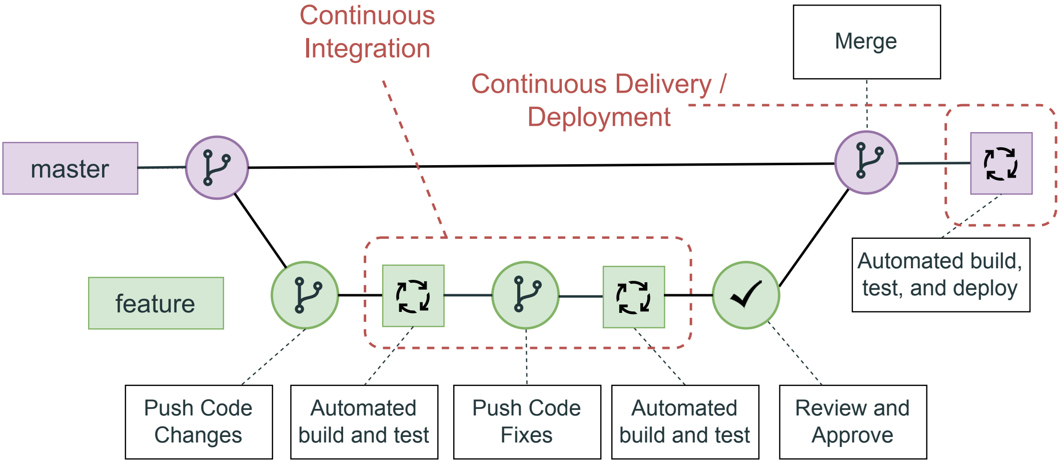
Definition

Continuous Deployment

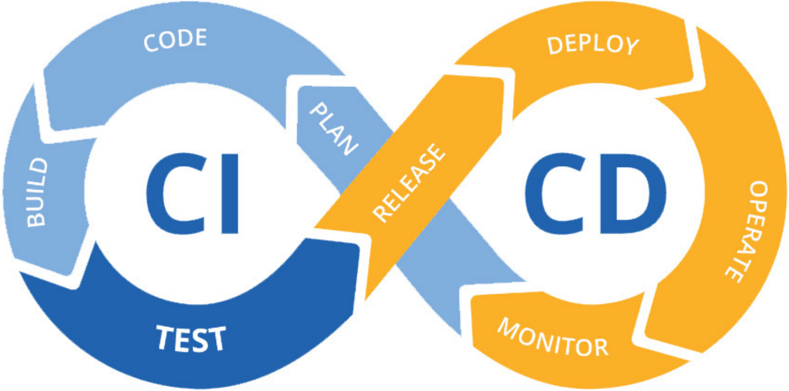
Continuous Deployment is another step beyond *Continuous Integration*, similar to *Continuous Delivery*. The difference is that instead of deploying the application manually, it is set to be deployed automatically.

→ With continuous deployment, human intervention is not required.

Definition


Continuous Integration
Merge
Continuous Delivery / Deployment
master
feature
Automated build, test, and deploy
Push Code Changes
Automated build and test
Push Code Fixes
Automated build and test
Review and Approve

Definition



CI / CD

4. Gitlab

Gitlab

Gitlab: Out-of-the-box DevOps tool supporting code repositories and, among others, the ”continuous methodologies”:

- Continuous integration.
- Continuous deployment.
- Continuous delivery.

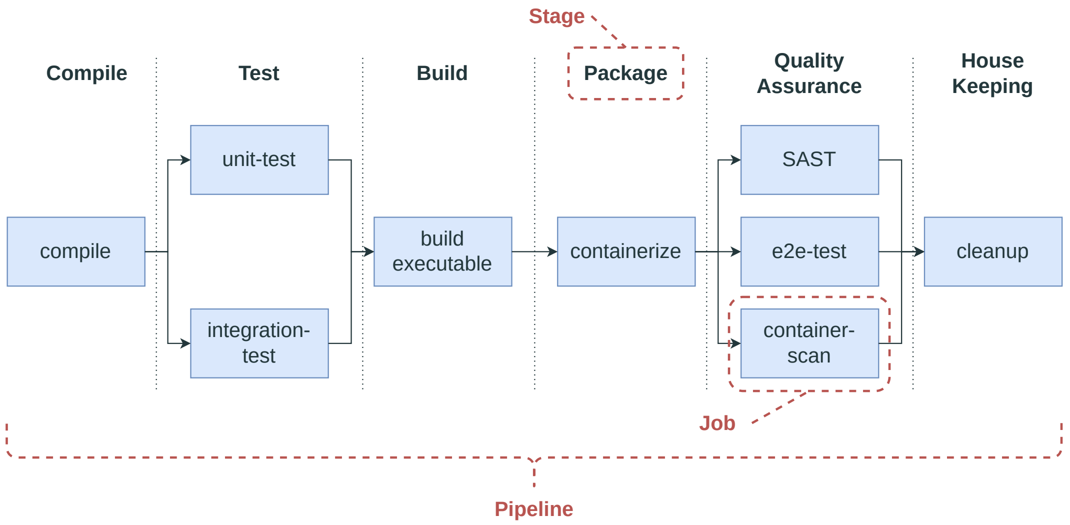
Note: There are plenty of tools that cover CI. However, Gitlab CI provides state-of-the-art features and is highly relevant in industry.

Gitlab

Basic structure of Gitlab CI/CD:

- The top level component of Gitlab CI/CD is a *pipeline*.
- A pipeline is composed of *stages*, which are executed in sequence and define when and in which order something needs to be performed.
- One stage is composed of *jobs*, which are executed in parallel and define what and how something needs to be performed.

Gitlab


Stage
Compile
Test
Build
Package
Quality Assurance
House Keeping
unit-test
SAST
compile
build executable
containerize
e2e-test
cleanup
integration-test
container-scan
Job
Pipeline

Gitlab

Stages are defined in `.gitlab-ci.yml` as YAML dictionary at root level named `stages` containing a YAML list of strings representing the respective stages.

Example:

```yaml
stages:
    - compile
    - test
    - build
    - package
    - e2e
```

Gitlab

Jobs are defined in `.gitlab-ci.yml` as YAML dictionary at root level with arbitrary name.

- The key `stage` specifies the stage to which the job belongs.
- The key `script` contains the commands of the job.

```yaml
stages:
    - build

job1:
    stage: build
    script:
        - pip3 install build
        - python3 -m build
```

Gitlab

The key **`image`** specifies a Docker image that the job runs in.

```yaml
stages:
    - build

job1:
    stage: build
    image: python:3.11.2-alpine
    script:
        - pip3 install build
        - python3 -m build
```

→ This way, any dependency can be made available to CI scripts.

Excursus: DinD

Excursus: DinD

Docker in Docker (DinD)

Technique used to run a Docker container inside another Docker container, essentially creating a nested container environments.

DinD comes with performance and security considerations:

- Additional layer of virtualization → performance hit.
- The inner container may not have its own Docker daemon.
- *Note:* If an inner container acts as a client to the host’s Docker daemon, bind mounts are interpreted at host level (not at the outer-container level).

Gitlab

Example in Gitlab CI/CD:

```
stage:
    - package

containerize:
    stage: package
    image: docker:23.0.1-dind
    script:
        - docker build . -t mycontainer
        - docker push mycontainer
```

Gitlab

CI/CD variables are configurable values that are passed to jobs. They can be used at the global level, and also at the job level. Example:

```
stages:
    - build

variables: # scope is the pipeline
    foo: "bar"

job1:
    stage: build
    image: python:3.11.2-alpine
    variables: # scope is the job
        bar: "foo"
    script:
        - echo "Foo has a value of $foo"
```

Gitlab

Gitlab CI/CD initializes a set of **predefined variables**[4] to support pipelines.

Examples:

- `CI_JOB_ID`: The internal ID of the job, unique across all jobs.
- `CI_COMMIT_BRANCH`: The commit branch name.
- `CI_COMMIT_REF_SLUG`: The branch or tag in lowercase, shortened to 63 bytes, and with everything except `0-9` and `a-z`.

[4] `https://docs.gitlab.com/ee/ci/variables/predefined_variables.html`

Gitlab

Jobs can output an archive of files and directories.

```
job1:
    stage: build
    image: python:3.10.4-alpine
    script:
        - pip3 install build
        - python3 -m build
    artifacts:
        paths:
            - dist/
        expire_in: 1 week
```

→ Artifacts are a means to share files between stages.

Gitlab

Test reports are published using the job artifacts mechanism:

```yaml
stages:
    - test

job1:
    stage: test
    image: python:3.11.2-alpine
    script:
        - pip install pytest
        - pytest --junitxml=report.xml
    artifacts:
        reports:
            junit: report.xml
```

Gitlab

Coverage reports are published using the job artifacts mechanism:

```
job1:
    stage: test
    image: python:3.11.2-alpine
    script:
        - pip install pytest pytest-cov
        - coverage run -m pytest
        - coverage report
        - coverage xml
    coverage: 'some regex' # configs coverage extraction
    artifacts:
        reports:
            coverage_report:
                coverage_format: cobertura
                path: coverage.xml
```

Gitlab

The key `rules` is used to include or exclude jobs in pipelines:

- Use `if` to specify when to add a job to a pipeline.
- Use `when` to configure the conditions for when jobs run.

Example:

```yaml
deploy:
    stage: deploy
    image:
    script:
        ...
    rules:
        - if: $CI_COMMIT_BRANCH == "master"
          when: manual
```

Gitlab

The full keyword reference for the `.gitlab-ci.yml` is available here[5].

[5] `https://docs.gitlab.com/ee/ci/yaml/`

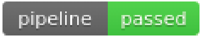
Gitlab

Badges

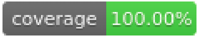
A graphical image that displays metadata about a project, such as build status or code coverage, and can be embedded on external websites or README files.

Examples:

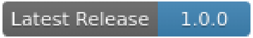
- Pipeline status



- Test coverage



- The Latest release



Details on *Badges* are available here[6]

[6] https://docs.gitlab.com/ee/user/project/badges.html

CI / CD

5. Summary

Summary

CI/CD best practices

- Use parallel jobs to shorten the execution time of the pipeline.
- It is okay to install packages in a job (if the execution times will allow it).
- Always clean up unused artifacts (incl. build artifacts, docker images, etc.)
- On very large and complex pipelines: Use includes.

Summary

Summary

You should have acquired the following competencies:

- Understand the DevOps mindset.
- Distinguish between *Continuous Integration*, *Continuous Delivery*, and *Continuous Deployment*
- Create pipelines for CI/CD using Gitlab.