Release Engineering

Advanced Software Engineering FS18

Gerald Schermann, schermann@ifi.uzh.ch
Release engineering is a software engineering discipline concerned with the development, implementation, and improvement of processes to deploy high-quality software reliably and predictably.

— Andrej Dyck, Towards Definitions for Release Engineering and DevOps, 2015
“DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality.”

— DevOps - A Software Architect’s Perspective, 2015
DevOps - Benefits

Technical benefits:
> Continuous software delivery
> Less complex problems to fix
> Faster resolution of problems

Business benefits:
> Faster delivery of features (time to market)
> More stable operating environments
> More time available to add value (rather than fix/maintain)
DevOps - Why

Common conflicts in software development or enterprise IT:

**Adopt latest technology vs. long term planning**
Increasing rate of technology change, the gap between leading edge and every-one-else is growing

**Moving quickly vs. stability and security**
Facebook’s “move fast and break things”
DevOps - Implications

Ensure quality of deployed change
by test automation, tests in production on a limited set of users (→ CD practices)

Ensure high velocity, thus highly automated,
repeatable delivery mechanisms
avoid downtimes, keep system available

Two important time periods for measuring velocity
- developer commits code: change enters deployment pipeline
- code is pushed to production: change is released to users
DevOps - Practices

Treat operations as *first-class citizens* from the point of view of requirements
i.e., ensure that they get the monitoring data and logs they need

Make developers more responsible for relevant incident handling
i.e., *developer on call* policies, allows identifying and fixing issues faster

Use continuous delivery/deployment

Develop infrastructure code, such as deployment scripts, with the same set of practices as application code
i.e., Infrastructure-as-Code, allows version-control and mitigates misconfiguration
“DevOps is a culture, movement or practice that emphasizes the collaboration and communication of both software developers and other information-technology (IT) professionals while automating the process of software delivery and infrastructure changes.”

— Mike Loukides, 2012
Development
(SOFTWARE ENGINEERING)

QA
(QUALITY ASSURANCE)

DevOps

Operations
DevOps

DevOps is not purely technical, includes several aspects summarized under **CALMS**:

> Culture
> Automation
> Lean
> Measurement
> Sharing
DevOps - CALMS

Culture

- Teams over individuals
- Cross-functional teams instead of “silos”
- Embrace change & experimentation

WORKED FINE IN DEV
OPS PROBLEM NOW
## DevOps - CALMS

| Culture     | Teams over individuals  
|             | Cross-functional teams instead of “silos”  
|             | Embrace change & experimentation  |
| Automation  | Continuous Delivery / Deployment  
|             | Infrastructure as Code  |
| Lean        | Be minimalistic (meeting numbers and times, team sizes, …)  
|             | Focus on producing value for the end user  |
| Measurement | Collect data on everything  
|             | Ensure to provide visibility into all systems and events (e.g., dashboards)  |
| Sharing     | Collaboration & communication instead of “throwing things over the fence”  
|             | Not just reporting facts, regular exchange of ideas  |
Agile Development
Continuous Integration
Continuous Delivery
Continuous Deployment
DevOps

Continuous Integration is 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.

— Martin Fowler
CI - Why

In many software projects, for long periods of time the application is **not in a working state**. Usually no one is interested in running the whole application until it is finished.

**Drawbacks:**
- Integration is a long and unpredictable process
  - e.g., may take weeks or even months
- Testing happens late in the process, thus bugs being often detected late
- Delayed and infrequent releases
- Poor project visibility, not clear what is actually working
- High maintenance costs
Continuous Integration

CI requires that every time somebody commits any change, the entire application is built and a comprehensive set of automated tests is run against it. If the build or test process fails, the dev team fixes the problem immediately.

CI is a paradigm shift:
> Without CI, software is broken until somebody “proves” it works
> With CI, software is proven to work with every new change. The moment it breaks, you fix it.

Teams with CI are able to
> deliver software faster
> with fewer bugs and
> bugs are caught earlier in the process when they are cheaper to fix
> providing cost and time savings
CI - Prerequisites

> Version control

> Automated build -
able to build the application from the command line

> Automated test suite
unit tests, integration tests, …

> Agreement and discipline across the teams, frequently check in small changes, fix failed builds before committing further changes
CI - Overview

Source: https://insights.sei.cmu.edu/devops/2015/01/continuous-integration-in-devops-1.html
CI - Tools

Jenkins

GitLab

circleci

Bamboo

CODESHIP

Travis CI
Jenkins

Started by user admin
Building in workspace /var/lib/jenkins/jobs/morea-fe16-group7/workspace

Building on remote machine: admin@remote

Konsolenausgabe

Started by user admin
Building in workspace /var/lib/jenkins/jobs/morea-fe16-group7/workspace

git rev-parse --is-inside-work-tree # timeout=10
Fetched changes from 1 remote Git repository

  * git config remote.origin.url git@github.com:morea-fe16-group7.git # timeout=10

Using git@github.com:morea-fe16-group7.git for remote Git repository

Using git@github.com:morea-fe16-group7.git for remote Git repository

Fetched upstream changes from git@github.com:morea-fe16-group7.git

Fetched upstream changes from git@github.com:morea-fe16-group7-server.git

Build triggered by ref:refs/heads/trunk

Build triggered by ref:refs/remotes/origin/

Build triggered by ref:refs/heads/develop

Log not saved.

BUILD SUCCESSFUL

Total time: 7.116 sec
“Continuous Delivery is a software development discipline where you build software in such a way that the software can be released to production at any time.”

— Martin Fowler
Continuous Delivery

> Based on CI
> CI focuses on development teams, the output of CI is the input to manual testing process and to the rest of the release process
> Much of time is “wasted” on this way through testing and operations (e.g., testers wait for “good” builds of the software, operations teams wait for documentation or fixes)
> Software sometimes undeployable because it takes too long to get it into production-like environments and because of a slow feedback loop between dev, QA, and operations

CD prevents this by:
  close, collaborative working relationship between the stakeholders involved in delivery  
    ==> DevOps culture
  extensive automation of all parts of the delivery process  
    ==> CD pipeline
Continuous Delivery Pipeline

- Commit Stage: Compile, Unit tests, Code analysis, Assemble binaries
- Acceptance Stage: Configure, Deploy binaries, Smoke tests, Acceptance tests
- UAT Stage: Configure, Deploy binaries, Smoke tests, Exploratory tests
- Capacity Stage: Configure, Deploy binaries, Smoke tests, Capacity tests
- Production: Configure, Deploy binaries, Smoke tests

Adapted from “Continuous Delivery” © Dave Farley and Jez Humble 2010
Continuous Delivery Pipeline

CI Phase:
> build application only once throughout release process
> store resulting binary in repository
> reuse binary for all following environments (internal test stages as well as production)
> platform/environment specific configurations for binaries are version controlled
> binary repos including config files allow rolling back to specific previous versions in case of issues without the need of recompilation
Continuous Delivery Pipeline

Post CI Phases:
> consist of multiple environments/stages for testing purposes

> include manual and automated testing

> some stages might run in parallel (e.g., user acceptance testing (UAT) on UI and performance tests on backend systems)

> stages are either automatically executed when previous stage is finished, or after manual approval (e.g., push of a button)
Deployment Phase:
> new version is available to be released to customers

> various strategies and practices exist for rolling out:
  - canary release
  - A/B testing
  - dark launches
  - gradual rollouts
  - blue/green deployments
Why CD?

“I think the faster you move, the more tolerant you have to be about small things going wrong. But, the slower you move the more tolerant you have to be with large change sets that can be unpredictable.”

— Study participant
CD Benefits

> instead of releasing 3-4 times a year, releasing on a weekly or daily basis allows companies **staying ahead of competition**

> more frequent releases means obtaining user feedback more quickly, thus **building the right product**

> setting up a CD pipeline and automation leads to **increased productivity and efficiency**

> releases become routine, thus **more reliable** instead of having stressful releases 3-4 times a year

> highly automated quality checks **preserve human-caused errors** and increase product quality

> having reliable, frequent releases incorporating customer feedback leads to **customer satisfaction**

There is no “holy grail” for CD, usually this is a slow process and involves multiple challenges companies are faced with:

Organizational
  > domain constraints and traditional processes (e.g., manual approvals)
  > top-management must be convinced (added value of CD), rather high costs for automation
  > suppliers that rely on the company’s software

Technical
  > legacy applications
  > application architecture
  > ensure product quality, automation (test and deployment)
  > dealing with higher risks

Social
  > resistance to change
  > requires strong collaboration between teams, break down barriers and promote collaborative culture
  > changing responsibilities (e.g., dev on call, code ownership)
Detour Sign
Traditional 3-tier Web Application Architecture - Example
Traditional 3-tier Web Application Architecture - Example

Downsides:
> Scaling
> Every change requires redeployment of whole application
(Micro-)services
Architecture

- Browse Products UI
- User Management UI
- Checkout UI
- Order Service
- Recommendation Service
- User Management Service
- Product Info Service
(Micro-)services Architecture

Advantages:

> scale on service level
> deploy new versions of single services
> run multiple versions at the same time (→ CD rollout practices)
> services communicate using standardized communication protocols (e.g., HTTP), thus single services can be based on the technology which fits best

In DevOps, teams are often responsible for single services (both developing and operating them)
(Micro-)services Architecture

Downsides:

- higher communication costs
- hard to start from scratch with microservices, usually functionality is broken down once it gets too complex
Continuous deployment is the “next step” of continuous delivery: every change that passes the automated tests is deployed to production **automatically**.

Continuous delivery doesn't mean every change is deployed to production. It means every change is proven to be **deployable at any time**. However, the decision to release a new version is made by a **human**.
Continuous Experimentation
Continuous Experimentation

Releasing on a weekly or even daily basis implies a higher **risk that something might go wrong**. However, the change is “smaller”, thus problems can be identified and fixed faster.

Continuous experimentation and rollout practices can be seen as additional **risk mitigation strategies**, e.g., test a new feature on a small user group first, then decide about further rollouts.
Continuous Experimentation Practices

- Eat your own dog food
- Canary releases
- Gradual rollouts
- A/B testing
- Dark launches
- Blue/green deployments
Eat your own dog food

“Eating your own dog food, also called dogfooding, is a slang term used to reference a scenario in which a company uses its own product to test and promote the product.”
— wikipedia

Facebook employees use the newest version internally, if everything is as expected they roll it out to further users.
Canary Releases

**Idea:** release a new version to a subset of users first, while all other users still see the old, stable version

In case of issues with new version, only a small amount of users is affected, thus the impact of the issue is kept small

Canary is compared to the existing version in terms of a set of criteria such as stability, performance, or correctness

User selection based on:
- (geographic) location
- role (admin, early, stable, etc.)
- random basis (e.g., 1% of traffic)
Gradual Rollouts

**Idea:** increase the number of users assigned to the newest version in a stepwise manner until it completely replaces the former version shows whether the new version can cope with increasing load and scales correctly

often used in combination with canary releases
Canary / Gradual Example

How they started in Spain and Ireland:

How they rolled it out globally (after months):
Dark Launches

**Idea**: mitigate performance and reliability issues of new features when facing production-like load levels by deploying those features on production without being visible for users.

Through monitoring, dev teams are able to identify and fix remaining bugs and scalability concerns before enabling the feature for users.

Example: Facebook Chat
https://www.facebook.com/notes/facebook-engineering/facebook-chat/14218138919/
A/B Testing

Idea: compare two versions of software with each other, often only differentiated in one tested aspect, to determine the effect of a certain change.

form of statistical hypothesis testing, thus requires big enough sample size to have “statistical power”

mainly used on UI for testing various layouts or design aspects

collect and use metrics 

decide about winner using collected metrics and statistics
A/B Testing

50% of users

50% of users
Blue/Green Deployments

**Idea:** have two identical production environments, one hosting (green) the current, stable version, the other (blue) represents the final stage of testing for the new version.

Release: simply switch the router so that all incoming traffic goes to the blue (or green) environment and use the green (or blue) environment for testing the next version.

Advantage: in case of problems directly after the release, a quick rollback (switch) to the previous version is possible.

Obstacle: DBs are part of both environments, thus a switch requires DB migration/synchronization.

Source: [http://martinfowler.com/bliki/BlueGreenDeployment.html](http://martinfowler.com/bliki/BlueGreenDeployment.html)
Facebook Deployment Pipeline

Source: Development and Deployment at Facebook, IEEE Internet Computing, 2013
Facebook Deployment Pipeline

Source: Development and Deployment at Facebook, IEEE Internet Computing, 2013
Facebook Deployment Pipeline

Source: Development and Deployment at Facebook, IEEE Internet Computing, 2013
Facebook Deployment Pipeline

Source: Development and Deployment at Facebook, IEEE Internet Computing, 2013
Continuous Experimentation - Example
Continuous Experimentation - Example

Gradual Rollout

95% -> Existing System

5% -> Recommendation Feature

1 day

Response Time < 50 ms
CPU Utilization < 80 %
Continuous Experimentation - Example

Gradual Rollout

90% Existing System

10% Recommendation Feature

Response Time < 50 ms
CPU Utilization < 80%
Continuous Experimentation - Example

Gradual Rollout

80% → Existing System

20% → Recommendation Feature

Response Time < 50 ms
CPU Utilization < 80 %
Continuous Experimentation - Example

A/B Test
50%
30 days
50%

Existing System

Recommendation Feature

# Sold Items

59
Continuous Experimentation - Example
Continuous Experimentation - Example

- operate multiple versions at the same time
- metrics to monitor, duration of experiment, user groups
Data-driven decision making requires …

Monitoring
Monitoring

Monitoring is the process of observing and recording system state changes (e.g., logs) and data flows (e.g., communication between components).

Monitoring is important to

> verify that everything runs as expected (identify failures, performance problems)

> characterize workload (capacity planning, billing purposes)

> support rollout decisions (e.g., increase traffic assigned to canary based on measured user reactions) => data-driven decision making

> detecting intruders
Monitoring

Performance degradation observed by comparing current performance to historical data or by complaints from clients/users

Typical performance measures include
> latency
> throughput
> utilization

measured on application and infrastructure level (i.e., technical metrics)
Business Metrics

A set of an organization’s particular set of metrics to determine the effectiveness of their offerings and their support services.

Examples:

- Photo gallery website: photo upload rate, photo processing times, …
- Online shop: number of sold products within 24 hours, number of added user reviews, levels of user activity, …
- Video platform: average of daily views, number of clicked ads, …
Dashboards - Example VM Utilization
Dashboards - Example APM
Dashboards - Example Amazon EC2
Dashboards - Business Metrics

Source: https://blog.rjmetrics.com/2013/04/25/business-intelligence-dashboards-redefined/
Continuous Experimentation - Implementation Techniques

Feature Toggles
- dynamically enable and disable code segments (i.e., multiple “versions” in the code)
- basically as simple as an if-statement
- useful and cheap, might introduce complexity over time

Dynamic Traffic Routing
- use a network component (e.g., proxy, router) to determine what service (and thus which version) should receive and process a request
- multiple versions (thus instances) are running in parallel
Feature Toggles

```javascript
function doSomething(){
    if( featureIsEnabled("use-new-algorithm") ){
        ...
    }
    else{
        ...
    }
}
```

Usage scenarios:
> prevent incomplete features from being executed in production, i.e., enabled only on internal testing environments
> control rollout experimentation, e.g., which users see the newest version/feature (A/B testing, canary releases, gradual rollouts)
Feature Toggles

Feature toggles have to be treated with care as controlling and maintaining them could become problematic, especially the more feature toggles you have.

- Toggles can interfere with each other (i.e., hard to test)
- Can result in technical debt
- Should be removed at some point from the code base, which might introduce new issues
Dynamic Traffic Routing

Removes feature toggle complexity from the code base, requires a properly managed router/proxy mechanism and multiple running instances.
Empirical Study
# State of Continuous Experimentation

<table>
<thead>
<tr>
<th></th>
<th>Regression-Driven Experiments</th>
<th>Business-Driven Experiments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main Goal</strong></td>
<td>Mitigation of technical problems</td>
<td>Evaluation from a business perspective (e.g., monetary incentives)</td>
</tr>
<tr>
<td><strong>Practices</strong></td>
<td>Canary Releases, Dark Launches, Gradual Rollouts, Blue/Green Deployments</td>
<td>A/B Testing</td>
</tr>
<tr>
<td><strong>Data Interpretation</strong></td>
<td>Often intuitive, less process driven</td>
<td>Hypothesis- and data-driven</td>
</tr>
<tr>
<td><strong>Duration</strong></td>
<td>Minutes to multiple days</td>
<td>Order of weeks</td>
</tr>
<tr>
<td><strong>User Selection</strong></td>
<td>Small scoped, sometimes gradually increased</td>
<td>Two or more groups, constant size</td>
</tr>
<tr>
<td><strong>Responsibility</strong></td>
<td>Siloization</td>
<td>Multiple teams and services</td>
</tr>
<tr>
<td><strong>Implementation</strong></td>
<td>Feature toggles, traffic routing, distribution of binaries</td>
<td></td>
</tr>
<tr>
<td><strong>Obstacles</strong></td>
<td>Architecture, limited number of users, missing business value or not worth investments, lack of expertise</td>
<td></td>
</tr>
</tbody>
</table>
To conclude...
• What is release engineering?

• What is DevOps and what benefits does it have compared to “silozed” development, QA, and operations teams?

• What does CALMS stand for?

• What is the fundamental idea of continuous integration (CI)?

• What are the benefits of building the application just once in the CI stage of a pipeline?

• What challenges do companies face on their way for establishing continuous delivery (CD)?

• What are the advantages of (micro-)service-based architectures compared to monolithic architectures?
• What is the difference between continuous delivery and continuous deployment?

• What is the primary idea of continuous experimentation?

• Which experimentation practices do exist?

• Why is monitoring important in the context of experimentations?

• How are business metrics different from technical metrics?

• What are the two common techniques for implementing continuous experiments?

• What are potential obstacles for conducting continuous experiments?
# Periodic Table of DevOps Tools (V2)

<table>
<thead>
<tr>
<th>Element</th>
<th>Symbol</th>
<th>Company</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Github</td>
<td>Gh</td>
<td>Open Source</td>
<td>SCM</td>
</tr>
<tr>
<td>GitLab</td>
<td>Gt</td>
<td>Open Source</td>
<td>CI</td>
</tr>
<tr>
<td>Bitbucket</td>
<td>Bb</td>
<td>Enterprise</td>
<td>Deployment</td>
</tr>
<tr>
<td>LLDB</td>
<td>Lb</td>
<td>Enterprise</td>
<td>Cloud / IaaS / PaaS</td>
</tr>
<tr>
<td>Jenkins</td>
<td>Jn</td>
<td>Open Source</td>
<td>Release Mgmt</td>
</tr>
<tr>
<td>Docker Hub</td>
<td>Dh</td>
<td>Open Source</td>
<td>Logging</td>
</tr>
<tr>
<td>Travis CI</td>
<td>Tr</td>
<td>Open Source</td>
<td>Build</td>
</tr>
<tr>
<td>CircleCI</td>
<td>Cc</td>
<td>Open Source</td>
<td>Testing</td>
</tr>
<tr>
<td>GitHub Pages</td>
<td>Gp</td>
<td>Open Source</td>
<td>Containerization</td>
</tr>
<tr>
<td>Bitbucket Pages</td>
<td>Bp</td>
<td>Enterprise</td>
<td>Collaboration</td>
</tr>
<tr>
<td>GitLab Pages</td>
<td>Gp</td>
<td>Open Source</td>
<td>Security</td>
</tr>
</tbody>
</table>