



Application Deployment

Software Engineering II
WS 2019/20

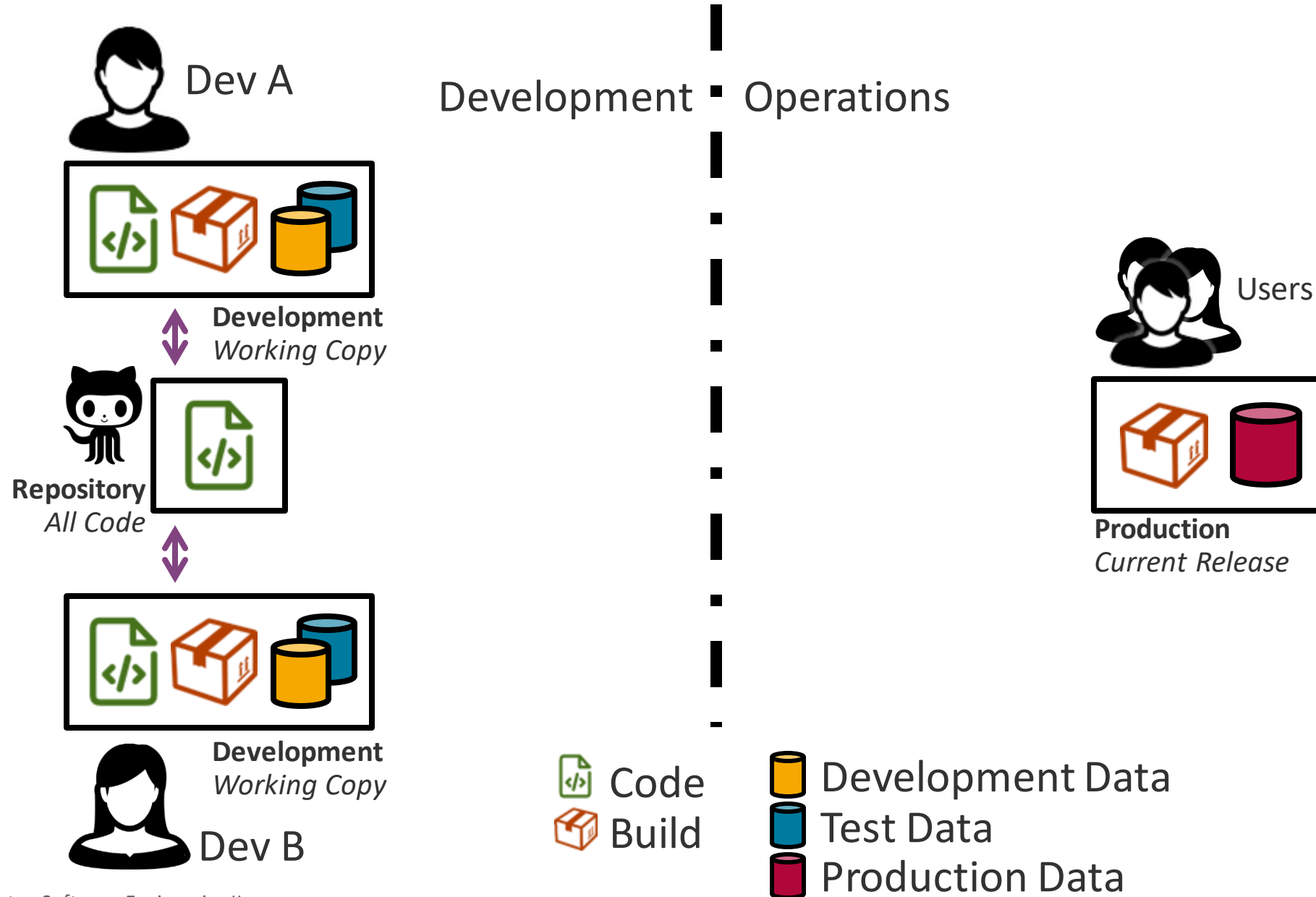
Enterprise Platform and Integration Concepts

Agenda



1. **DevOps**
2. Application Hosting Options
3. Automating Environment Setup
4. Deployment Scripting
5. Application Monitoring
6. Continuous Deployment and Scrum

Development vs. Operations



Development & Operations

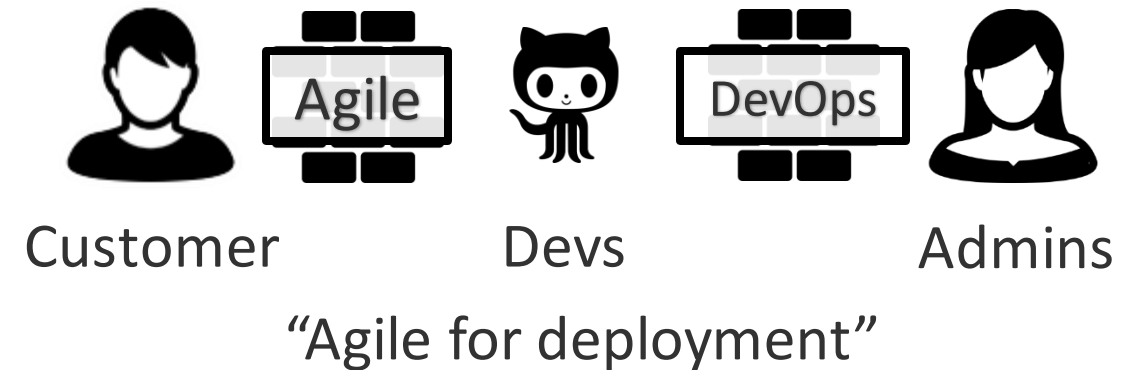


Problems

- Software needs to be operated, run in production, and maintained
 - Developers vs. Admins
- **Short** development and deployment cycles
- Maintain **quality standards**

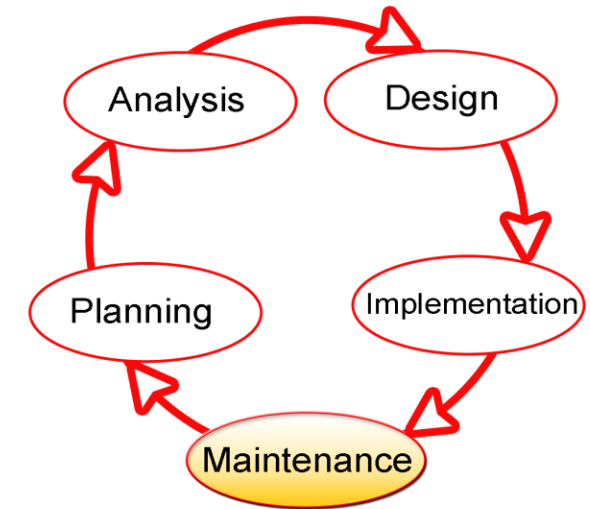
DevOps

- **Formalized** process for deployment
- Focus on communication, **collaboration**, and integration between Dev and Ops



Definition

- Fairly **recent** trend
- "[...] no uniform definition for [...] DevOps. [...] people use their own definitions" [Dyck, 2015]
- "There is no consensus of what concepts DevOps covers, nor how DevOps is defined" [Erich, 2017]
- Best practices to **shorten the application development life cycle**

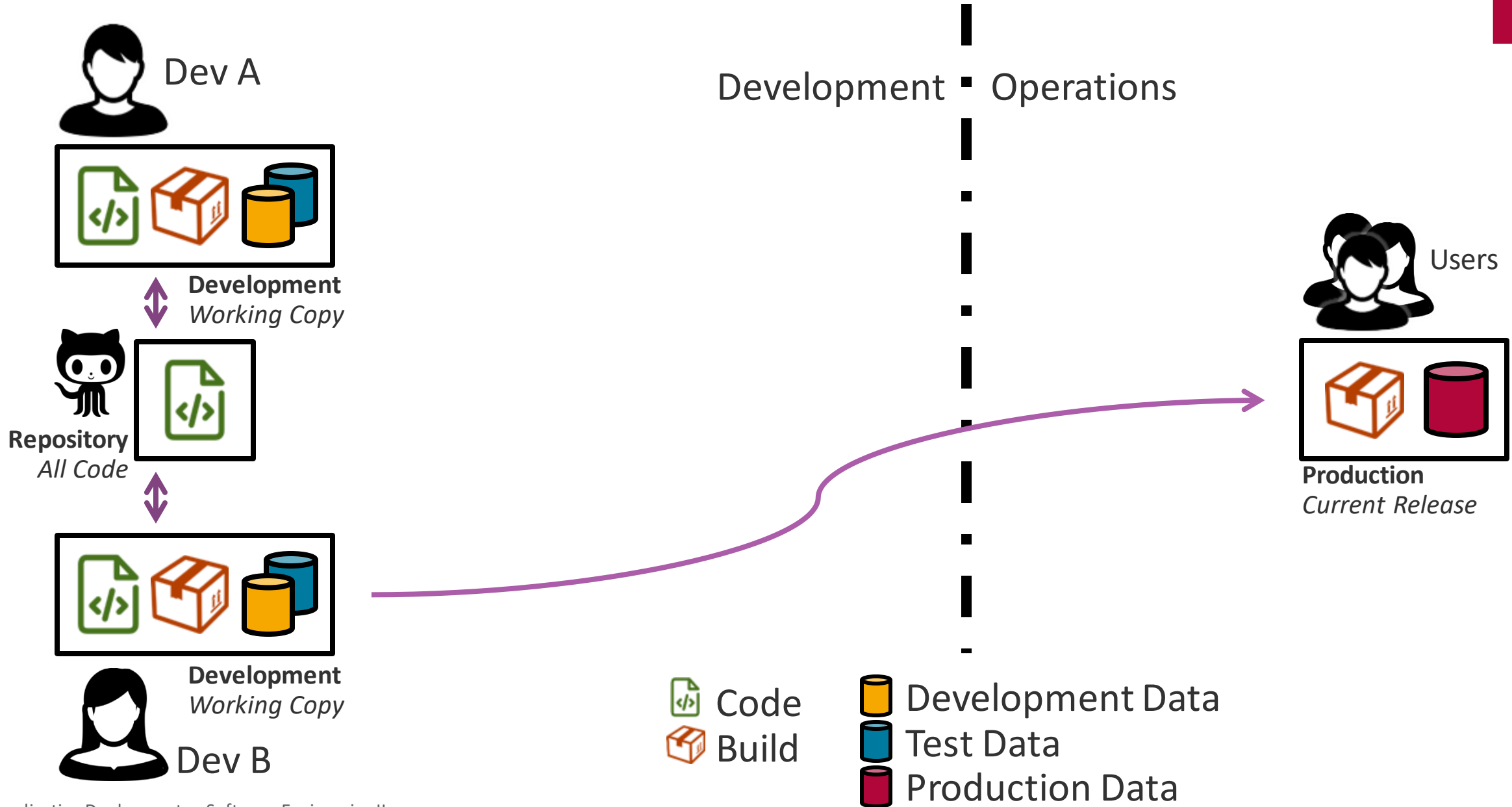


[Dyck, 2015] Dyck, Andrej; Penners, Ralf; Lichter, Horst (19 May 2015). "Towards Definitions for Release Engineering and DevOps". Proceedings of the 2015 IEEE/ACM 3rd International Workshop on Release Engineering. IEEE.

[Erich, 2017] Erich, F.M.A.; Amrit, C.; Daneva, M. (June 2017). "A Qualitative Study of DevOps Usage in Practice". Journal of Software: Evolution and Process. 29 (6).

Image: Dzonatas (CC BY-SA 3.0), <https://commons.wikimedia.org/wiki/File:SDLC-Maintenance-Highlighted.png>

Not DevOps

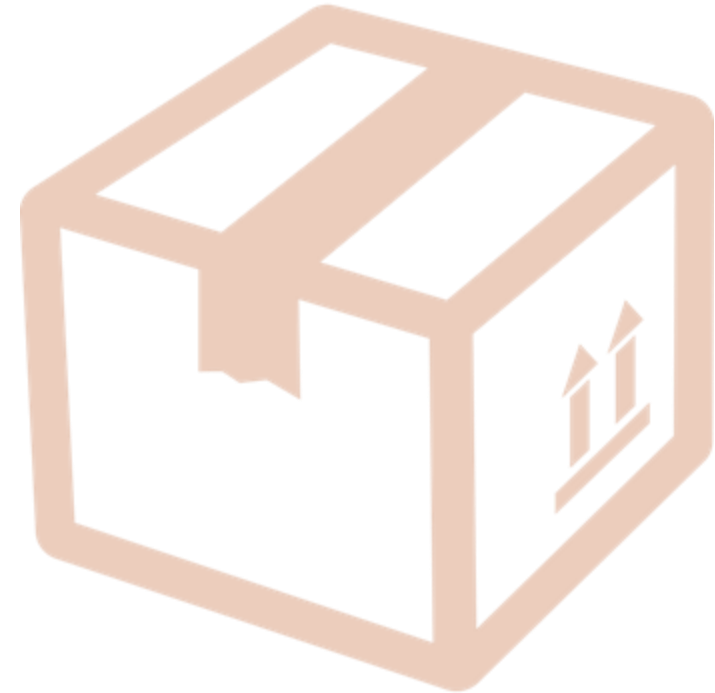


Terminology



Release

- **Planned state** of the application
- Set of requirements
- Examples
 - Next big version with new shiny features
 - Urgent hotfix
 - Anything in-between



Version

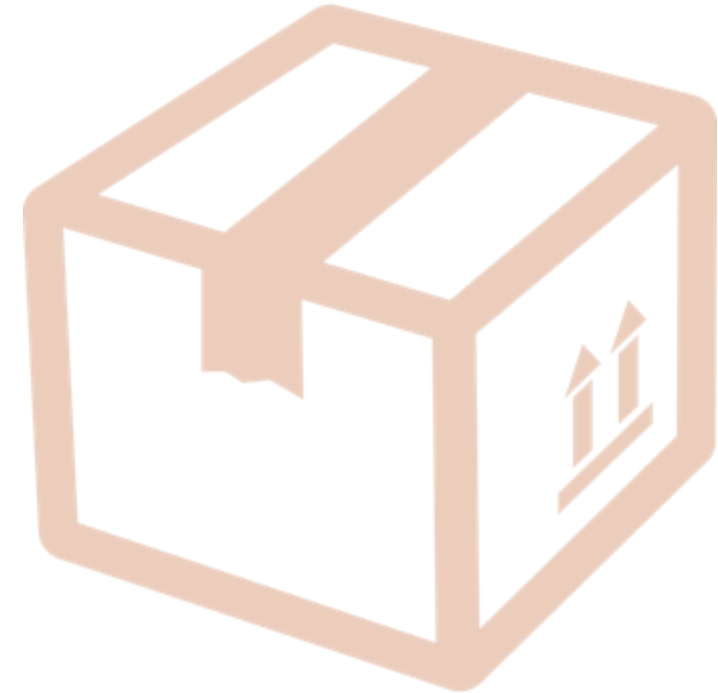
- Could be anything
- A release has a **version number**

Terminology



Build

- Attempt to implement a release
 - **Snapshot** of application
- Often the output of the build tool
 - Not: the build script/tool/process
- Version number is
“<Release Number>.<Build Number>”



Terminology



Environment

- A system on which the application can be deployed and used

To promote

- To deploy a build on the **next** environment

To release

- To promote a build to **production**
- Thereby finishing the release



Overview of Environments



Development

managed by developers

Development

- Where the developers work
- One per developer (if possible)

Integration

- Runs all tests
- A try-out version

Quality Assurance

- Professional manual testing

Operations

managed by admins

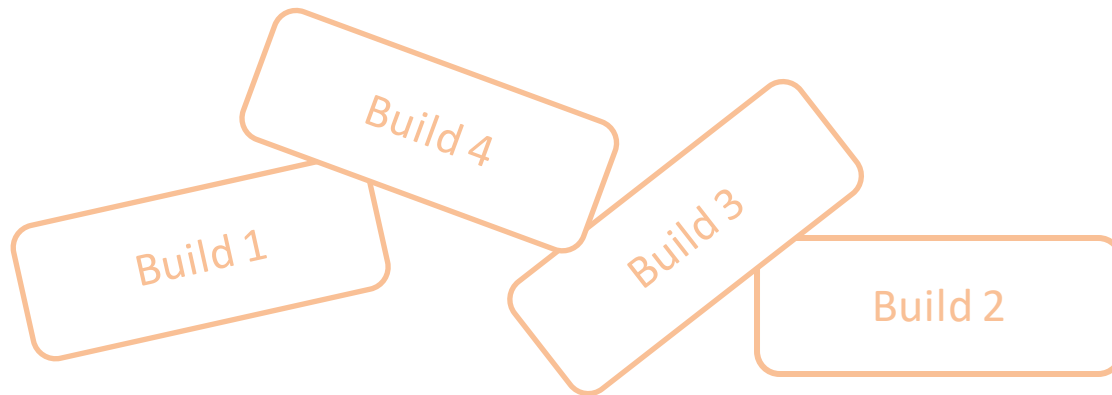
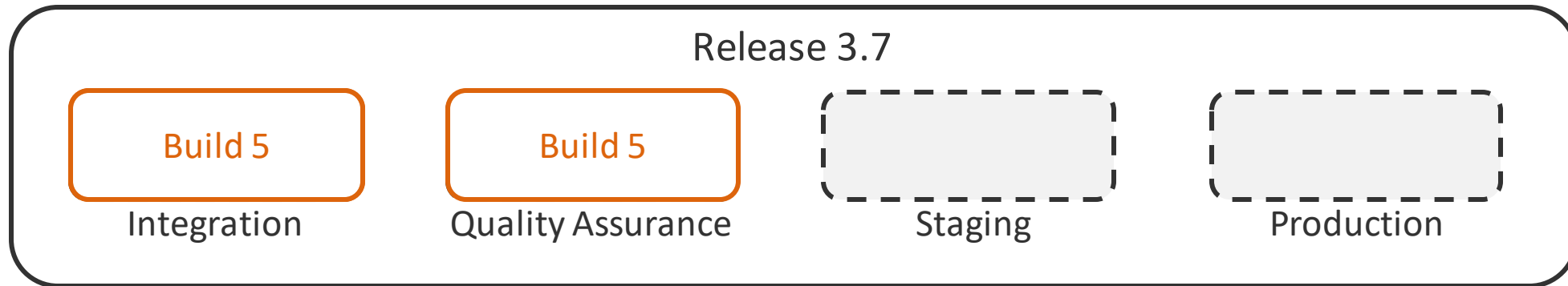
Staging

- Clone of production system
- Final rehearsal

Production

- The live system
- Failures are expensive here

Example



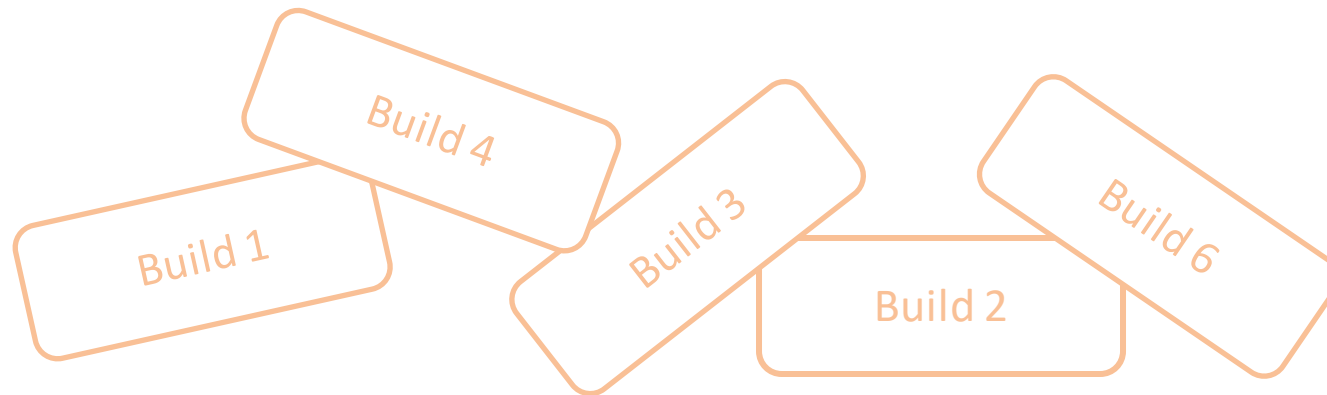
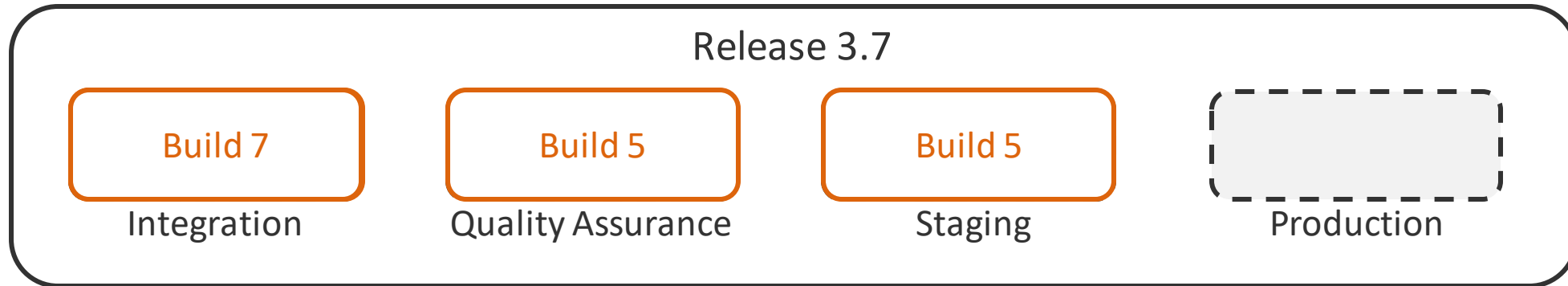
Example



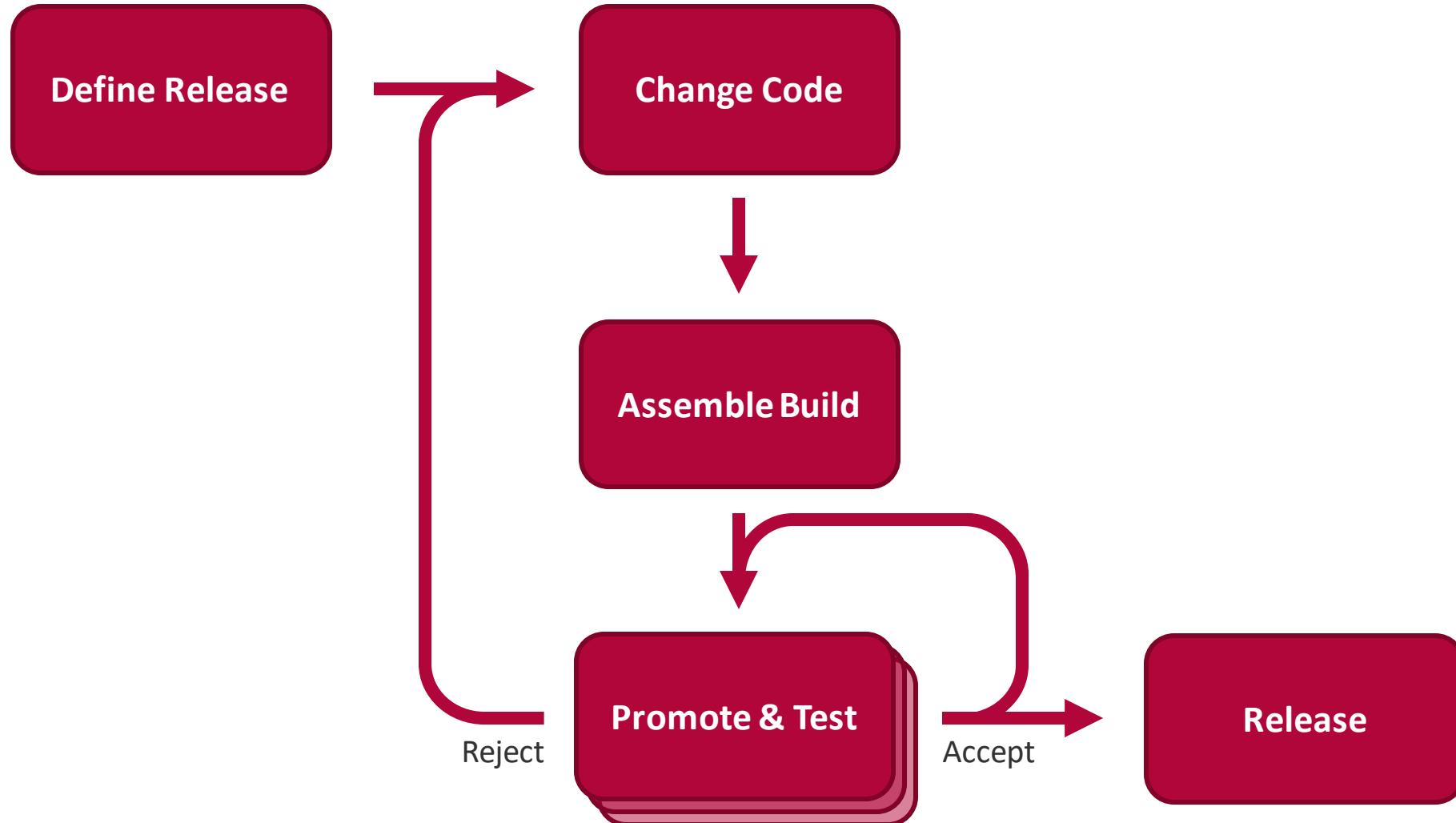
Build 8



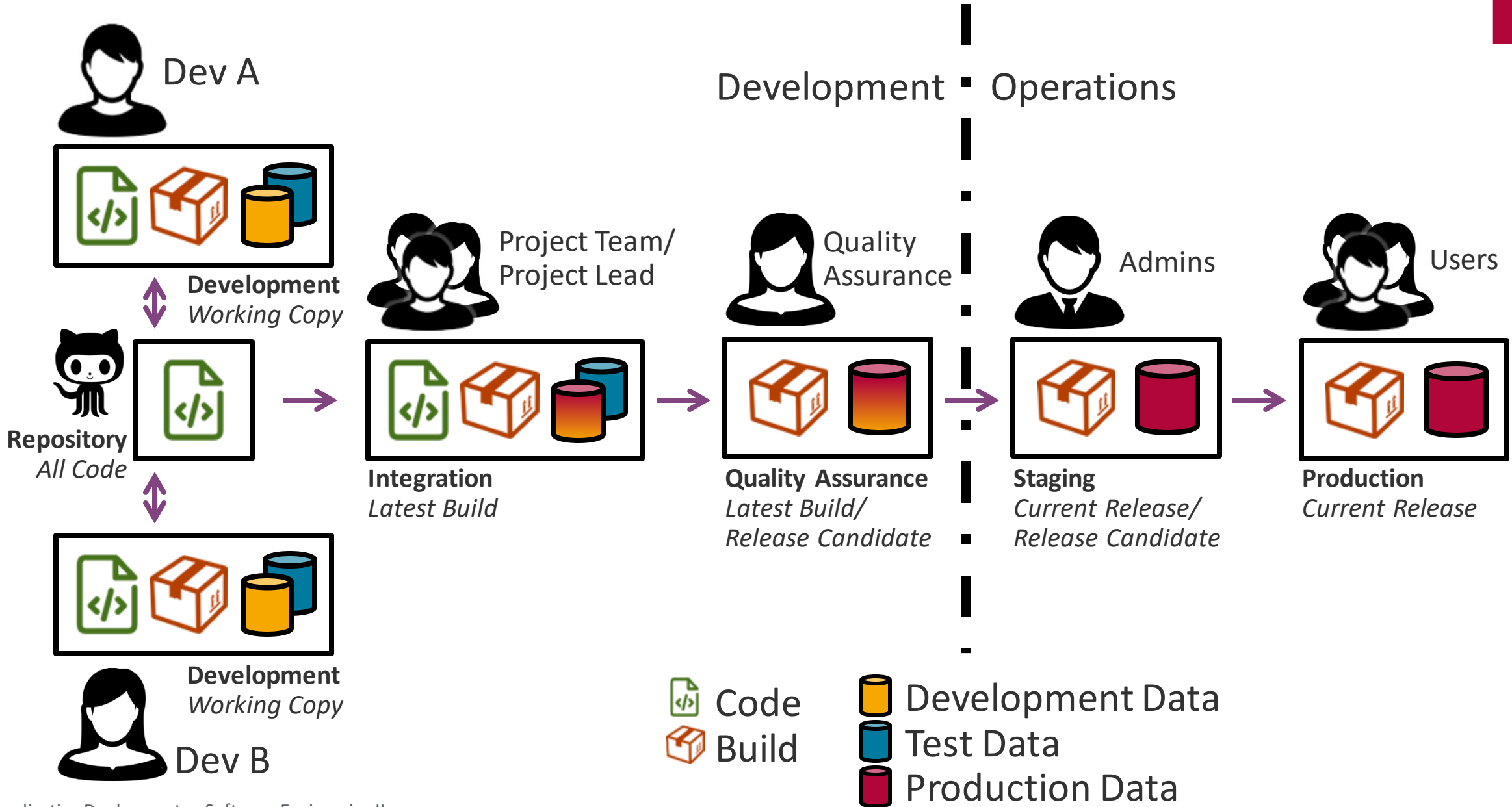
Developers
changing Code



Workflow



DevOps



Implications



Builds are immutable

- If changed, previous testing was pointless
 - Even the smallest change has to go through all environments

Many systems required

- Each environment has to be maintained
- Automation?

Deployment overhead

- Manual steps are potential for human failure
- Automation?

} Remainder of this lecture

Agenda



1. DevOps
- 2. Application Hosting Options**
3. Automating Environment Setup
4. Deployment Scripting
5. Application Monitoring
6. Continuous Deployment and Scrum

Application Hosting Options

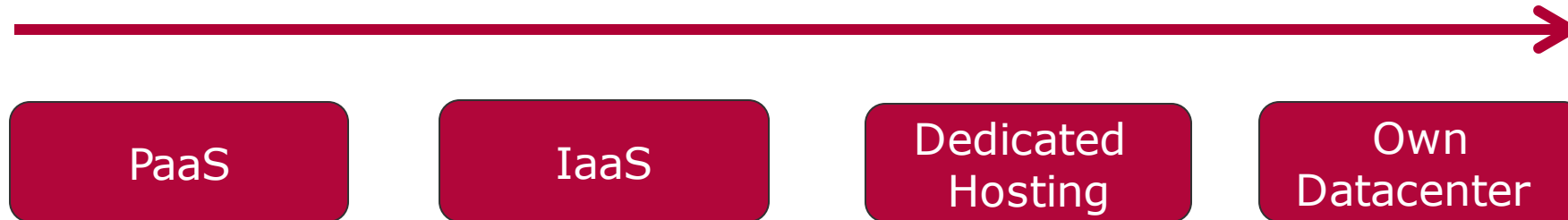


Choice of hosting options is driven by a variety of parameters

- Initial setup effort, cost, and required expertise
- Operational costs and effort
- Targeted service level agreements (SLAs)
- Legal considerations (data privacy, liability, etc.)

Low Effort
Little Control

High Effort
High Control



Platform as a Service (Paas)



Providers deliver OS, execution environment, database, web server, monitoring, etc.

Advantages

- Minimal effort and knowledge required for setup
- Only platform development knowledge (e.g. Python, Ruby) needed, no need for hardware / OS maintenance
- Possibility to scale up quickly and easily

Disadvantages

- Usually fixed environment with little variation points
- Provider SLA targets might differ from yours, e.g. downtime, response times
- Limited technical support

Examples: Heroku, Azure Compute, Google App Engine

Infrastructure as a Service (IaaS)



Providers deliver virtual private servers (VPS) with requested configuration
Setup of execution environment, database servers, etc. is up to customers

Advantages

- Flexibility regarding execution environment
- Avoid management of underlying hardware
- Dynamic on-demand scaling of resources

Disadvantages

- Server administration know-how and efforts required
- It's still a VM: Potential performance drops, Disk I/O, etc.

Examples: Amazon EC2, Google Compute Engine, Rackspace Cloud, DigitalOcean

Dedicated Hosting



Providers allocate *dedicated* hardware, classical approach

Advantages

- Complete control over server, down to bare metal, full power always available
- No virtualization-related performance issues
- More control over network configuration
- Dedicated SLAs

Disadvantages (compared to IaaS)

- No easy scaling of resources
- Administration efforts for servers, e.g. monitor disk failures

Examples: Hetzner, OVH, Rackspace, Host Europe

Own datacenter



You host your own servers

Advantages

- Complete control over data, security, operations, network etc.
- Custom designed servers possible
- Add cabinets in available space with low cost

Disadvantages

- Huge upfront costs, e.g. space, cooling, fiber, hardware
- Expanding the space of the datacenter is expensive
- Provide around the clock support, monitoring, personnel, etc.
- Not feasible for small companies

Examples: Google, Facebook

Agenda



1. DevOps
2. Application Hosting Options
3. Automating Environment Setup
 - **Virtualization**
 - Provisioning
4. Deployment Scripting
5. Application Monitoring
6. Continuous Deployment and Scrum

Setting up an Environment



Main challenges in preparing infrastructure:

- Minimize the effort required to repeatedly setup identical execution environments
- Without relying on “administration gurus”

Solutions:

- *DevOps*, i.e. a strong collaboration between the development and the operations team
- A strong bias towards automation

Where to Start With "Deploying"?



- Hosted solutions aren't always feasible for initial experiments
- Maintaining local installs of server stacks in different versions can get cumbersome
- Development vs. production environment differences result in *"it works on my machine"* problems
- Don't want to force all developers to use same development environment (e.g. choice of OS)

Possible solution: VirtualBox + Vagrant (<https://www.vagrantup.com/>)

- "Deploy" to a virtual machine on your local OS for development

Agenda



1. DevOps
2. Application Hosting Options
3. Automating Environment Setup
 - Virtualization
 - **Provisioning**
4. Deployment Scripting
5. Application Monitoring
6. Continuous Deployment and Scrum

Next Step: Automate VM Config



Virtualization software provides a VM.

Provisioning tools configure it, e.g. install required software.

Why not provision manually?

- Error prone, repetitive tasks
- Documentation has to be kept up-to-date
- Explicit knowledge transfer required if Admin changes

One provisioning tool example: Chef (<http://chef.io>, <https://github.com/chef/chef>)

- Formalize software install and configuration state into *recipes*
- Recipes are shared (<https://supermarket.chef.io/cookbooks>)
- Ensure software is installed based on dependencies
- Ensure that files, packages, and services are in the prescribed state

Common alternative: Puppet (<https://puppetlabs.com/>)

Provisioning Summary



Create your VM, e.g. describe it with Vagrant.

Using provisioning tools, you can:

- Define the required packages for all required servers
- Install and configure necessary services
- Create the directory structure for your application
- Create custom configuration files (e.g., database.yml)

Not touched here but also possible:

- Use templates to create different files based on variables
- Environments (staging vs. production)
- Central management of configuration files that are automatically transferred to clients

Agenda



1. DevOps
2. Application Hosting Options
3. Automating Environment Setup
- 4. Deployment Scripting**
5. Application Monitoring
6. Continuous Deployment and Scrum

How to deploy?



Necessary steps after the server is configured:

- Checkout code changes
- Update your dependencies (i.e. gems)
- Run database migrations, restart application servers
- Optional: Restart index servers, setup new Cron jobs, etc.

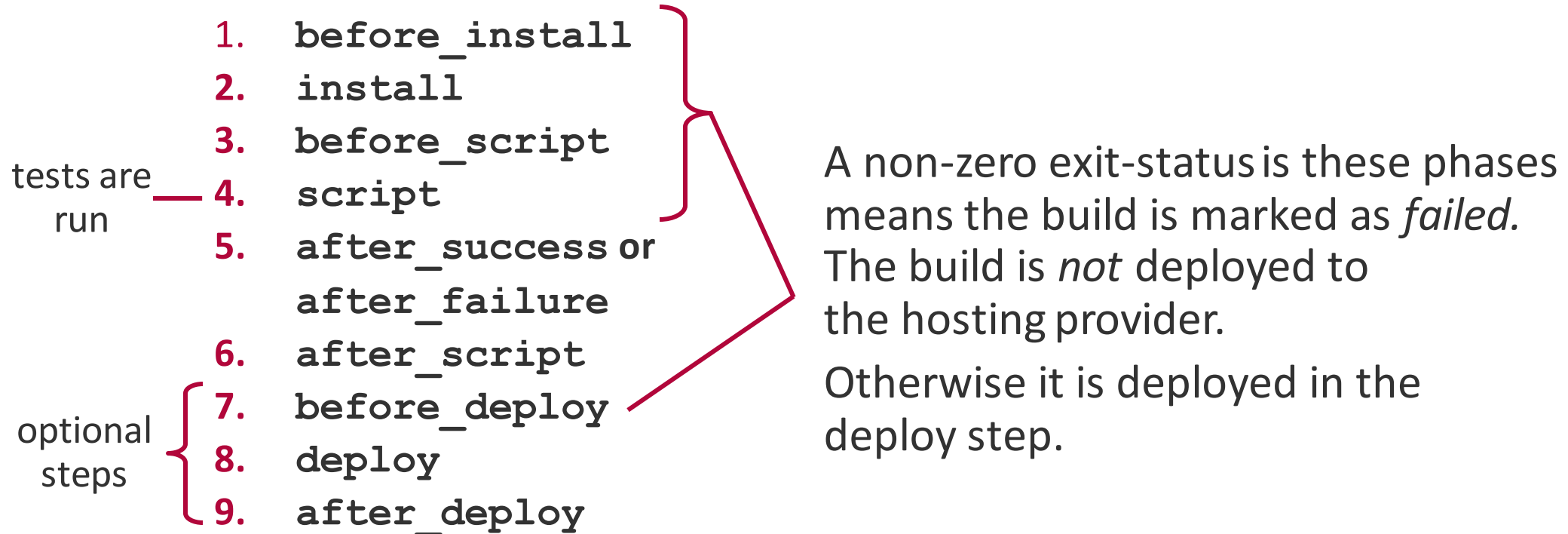
Remember: Automation!

- Easiest: **CI solutions** (e.g. Travis CI) support deploying to hosting providers
 - Deploy after all the tests pass
- Alternative: Dedicated tools (e.g. Capistrano in Ruby, <https://github.com/capistrano/capistrano>)
 - Prepare the server for deployment (possibly using provisioning tools)
 - Deploy the application as updates are made

Deployment with CI



Travis CI Continuous Integration and Deployment Workflow:



- A custom `after_success` step can be used to deploy to own servers (<http://docs.travis-ci.com/user/deployment/custom/>)

Agenda



1. DevOps
2. Application Hosting Options
3. Automating Environment Setup
4. Deployment Scripting
- 5. Application Monitoring**
6. Continuous Deployment and Scrum

Monitoring Servers & Applications



Keep an eye on server health and applications:

- Monitor in production, this is where errors are costly
- Issue alerts when components fail or exceed predefined thresholds
- Examples:
 - Uptime Robot—HTTP GET / ping every 5 mins (<https://uptimerobot.com/>)
 - Nagios—Monitor infrastructure, down to switches and services (<http://nagios.org>)

Monitor application errors and performance bottlenecks:

- Monitor errors that happen at runtime, discovered by users
- Notifications on application errors, slow downs
- Examples:
 - Errbit—Collect and organize errors (<https://github.com/errbit/errbit>)
 - New Relic—Performance monitoring, response times, SQL (<http://newrelic.com/>)

Agenda



1. DevOps
2. Application Hosting Options
3. Automating Environment Setup
4. Deployment Scripting
5. Application Monitoring
6. **Continuous Deployment and Scrum**

Deploying 50 times a day? Continuous Delivery



Advantages:

- Users get a sense of “something happening” frequently, shorter feedback loop
- Business value of features immediately present
- Deploy scripts used often, less likely to contain errors
- Reduced amount of code changes per release → faster fixes, less downtime

Prerequisites/Disadvantages:

- Only feasible with extensive set of *good* tests
- Tests / deployment need to run fast (*Continuous Integration*)
- Additional training for developers (*DevOps*) required
- May not be feasible for applications that require planning or long-term support (e.g. operating systems)



Discussion:

Operating systems feature both CD (rolling releases) and classical approaches (LTS releases)

Continuous Deployment vs. Scrum



How do 50 deployments a day fit into Scrum's notion of Sprints?

Some ideas (let's discuss):

- Intermediate Reviews for individual stories by the PO
 - At sprint review, each finished story is already running in production
 - Review meetings become shorter, more of a high level overview
- Get faster feedback from stakeholders for next Scrum meeting
- Deploying to staging or testing systems becomes part of the definition of done
- Acceptance of features not only based on PO approval but stakeholder approval?
 - A/B testing?
- "Working software is the primary measure of progress" —*Agile Manifesto*
 - Is software that is not deployed *working*? (*DevOps*)

Summary



1. DevOps
2. Application Hosting Options
3. Automating Environment Setup
4. Deployment Scripting
5. Application Monitoring
6. Continuous Deployment and Scrum

Conclusion: **Automate everything!**