Cloud Computing

Cloud-Native applications

Slide set 6

Henry-Norbert Cocos

cocos@fb2.fra-uas.de

Computer Science

Department of Computer Science and Engineering

Frankfurt University of Applied Sciences

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Agenda

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Traditional Applications

- **Monolithic**
  - Traditional (monolithic) architectures do not scale up to the demands of the customers.
  - These architectures are heavyweight and the deployment in the CSP is very complicated.
- **Three-Tier architectures**
  - Three-Tier architectures are an outdated architecture for web applications.
  - The deployment depends on specific web server software.
  - The frameworks are very heavyweight and not interoperable.
- **Service-oriented architectures**
  - SOA was a first direction into reusable services for business.
  - Every business application is orchestrated with individual atomic services.
  - However the ESB is a bottleneck and limits the whole architecture.

Question

What do we need for the development of cloud-ready applications?

12 factor app

Source: `https://12factor.net/`

A methodology for developing web apps, or software-as-a-service offerings.
Demands:

- Declarative formats for setup automation, to minimize time and cost. ⇒ **Infrastructure as Code!**
- Clean contract with the underlying operating system, offering maximum portability. ⇒ **Container Runtime!**
- Deployment on modern cloud platforms, without the need for servers and systems administration. ⇒ **PaaS/CaaS/FaaS!**
- Minimize divergence between development and production. ⇒ **Staging!**
- Scale up without significant changes to tooling, architecture, or development practices. ⇒ **DevOps!**

Applicability

The twelve-factor methodology can be applied to apps written in any programming language, and which use any combination of backing services (database, queue, memory cache, etc).

Question

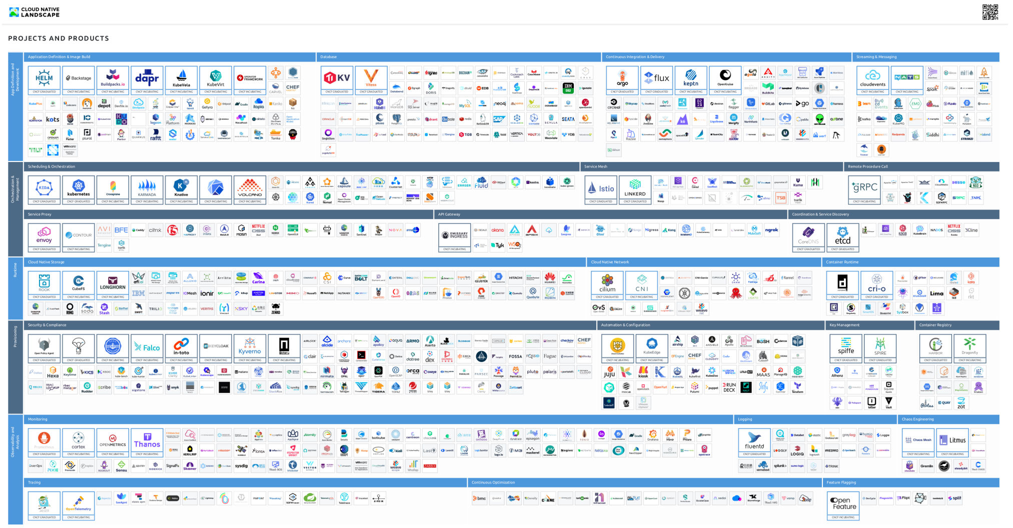
How can these demands be fulfilled in practice?

12 factor app

Source: https://12factor.net/

1. **Codebase**
   - One codebase tracked in revision control, many deploys
2. **Dependencies**
   - Explicitly declare and isolate dependencies
3. **Config**
   - Store config in the environment
4. **Backing services**
   - Treat backing services as attached resources
5. **Build, release, run**
   - Strictly separate build and run stages
6. **Processes**
   - Execute the app as one or more stateless processes
7. **Port binding**
   - Export services via port binding
8. **Concurrency**
   - Scale out via the process model
9. **Disposability**
   - Maximize robustness with fast startup and graceful shutdown
10. **Dev/prod parity**
    - Keep development, staging, and production as similar as possible
11. **Logs**
    - **Treat logs as event streams**
12. **Admin processes**
    - Run admin/management tasks as one-off processes

Cloud-Native Computing Foundation – Landscape



Cloud-Native definition

Definition Source: Cloud-Native Computing, Kratzke

"A cloud-native application is a distributed, observable, elastic service-of-services system optimized for horizontal scalability that isolates its state in (a minimum of) stateful components. The application and each self-contained delivery unit of this application are designed according to cloud-focused design patterns and operated on these elastic self-service platforms."

Question

How can we implement Cloud-Native applications and what technologies and workflows do we need?

Cloud-Native implementation

Source: Cloud-Native Computing, Kratzke

1. **Containerization ⇒ Slide Set 2!**
   - Containers allow developers to package applications and their dependencies in isolated environments and run them on any cloud platform.
2. **Orchestration Slide Set 2!**
   - Orchestration tools such as Kubernetes make it possible to provision, scale, monitor and manage container clusters automatically.
3. **Microservices Slide Set 5!**
   - Cloud-native applications generally comprise smaller, independent services (microservice architecture) that communicate via resource APIs.
4. **Infrastructure as Code Slide Set 2!**
   - Infrastructure as code means that the entire infrastructure (e.g., servers, networks, load balancers) is defined in code.
5. **Automation This Slide Set!**
   - By automating development, testing, and deployment processes using deployment pipelines, teams can provide faster and more reliable software updates.

Components of Cloud-Native

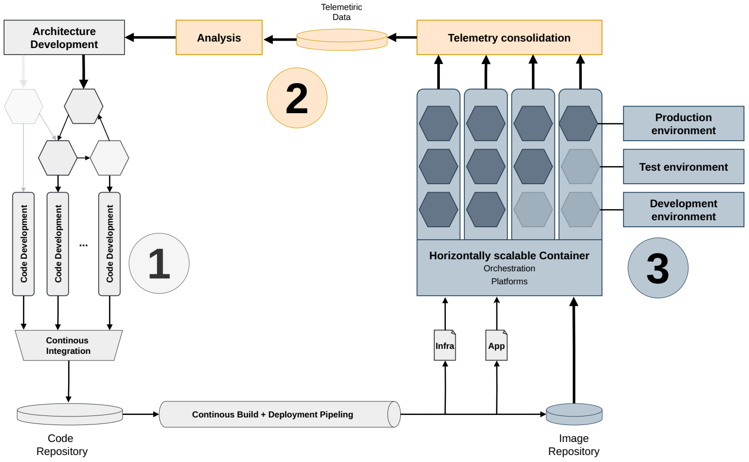
IDEAL Source: Cloud-Native Computing, Kratzke

*"[..] a cloud-native application should follow the **IDEAL** model, i.e., it should have an **I**solated state, be **D**istributed in nature, be **E**lastic via horizontal scaling, be operated via an **A**utomated Management System, and its components should be **L**oosely Coupled."*

Benefits Source: Cloud-Native Computing, Kratzke

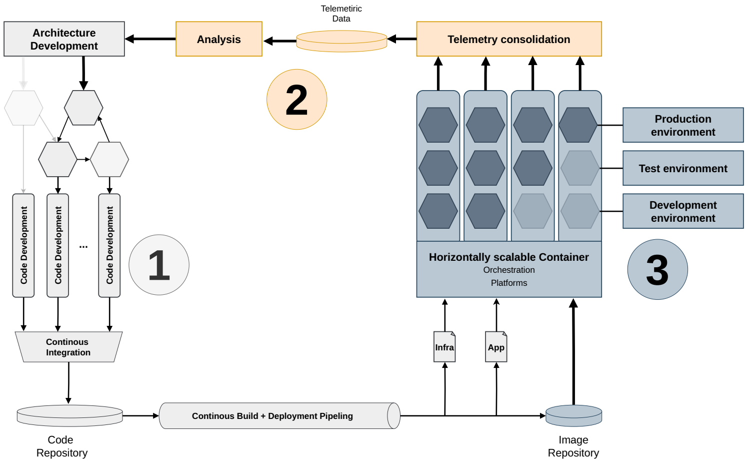
*"[...] common motivations for cloud-native application architectures must also be taken into account; for example, that software-based solutions can be brought into production much faster today **(development speed)**, that systems should be designed from the ground up to be fault-isolating, fault-tolerant and self-healing **(security)**, that more and more horizontal **(instead of vertical)** application scaling must be made possible **(scaling)** and, finally, that a wide variety of **(mobile)** platforms and legacy systems must be supported **(client diversity)**."*

DevOps



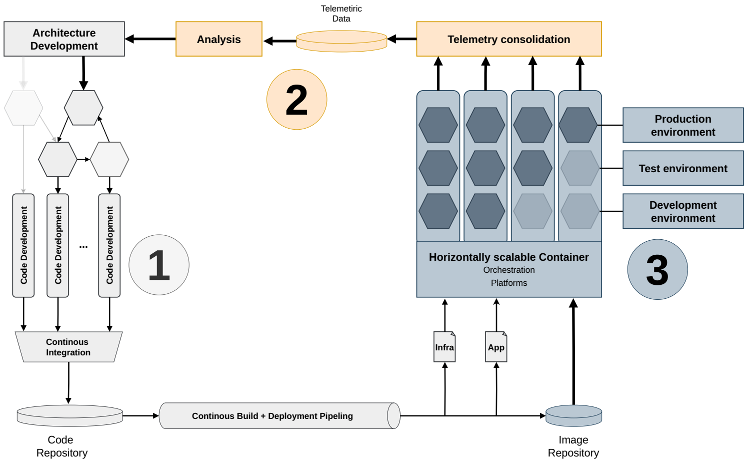
1. **Principles of flow**
   - Make work visible
   - Limit work in progress
   - Minimize bottlenecks
2. **Principles of feedback**
   - Recognize problems early
   - Solve problems immediately
   - Taking professional responsibility for problems
3. **Automated execution environment**
   - Container plattform
   - Implementation of Services (Dev)
   - Operation of Services (Ops)

DevOps – 1.) Principles of flow (1/3)



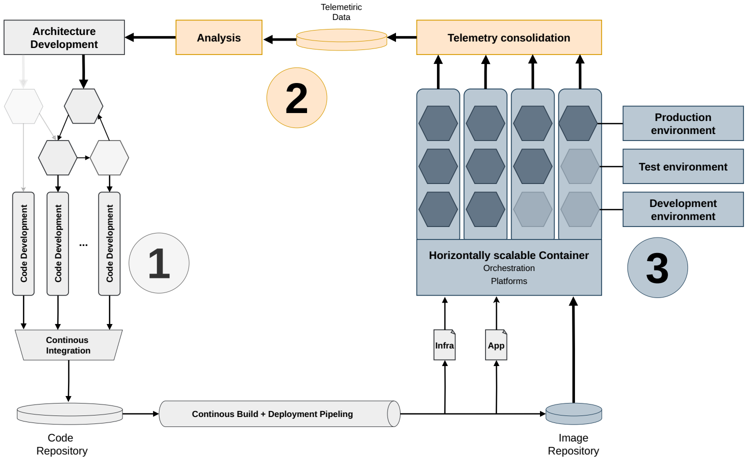
- **Make work visible**
  - Visual work boards (e.g. Kanban boards) are often used for this purpose. Modern version management systems such as GitLab can create such overviews automatically.

DevOps – 1.) Principles of flow (2/3)



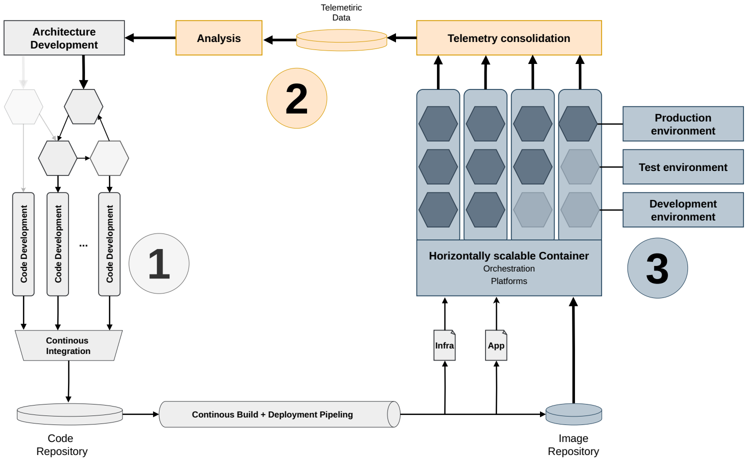
- **Limit work in progress**
  - The number of features to be implemented that developers should have *"on their desks"* simultaneously should be limited. Otherwise, the risk of jumping back and forth between too many tasks is too significant. Above all, you should avoid handing over tasks to development unplanned (*"Can you please quickly ..."*). Otherwise, you risk overloading developers who can only save themselves through *"tactical fouls"*.

DevOps – 1.) Principles of flow (3/3)



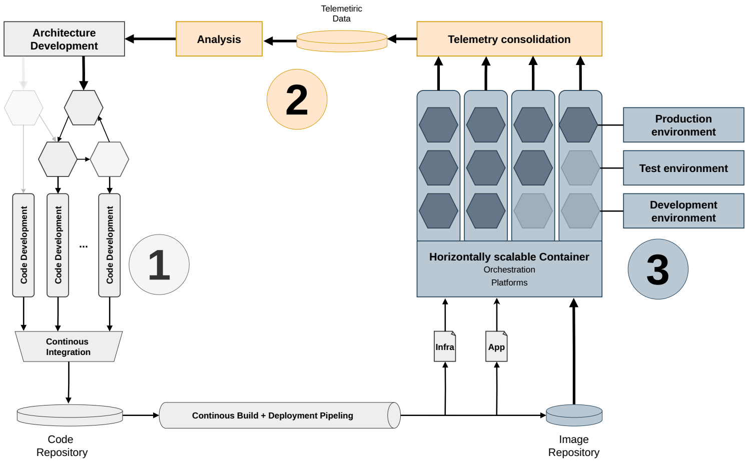
- **Minimize bottlenecks**
  - Manual activities often cause bottlenecks in software development. In particular, the degree of automation of activities such as the creation of test and production environments, the testing of changes, or the execution of code deployments should continuously increase in a DevOps culture to minimize bottlenecks.

DevOps – 2.) Principles of feedback (1/3)



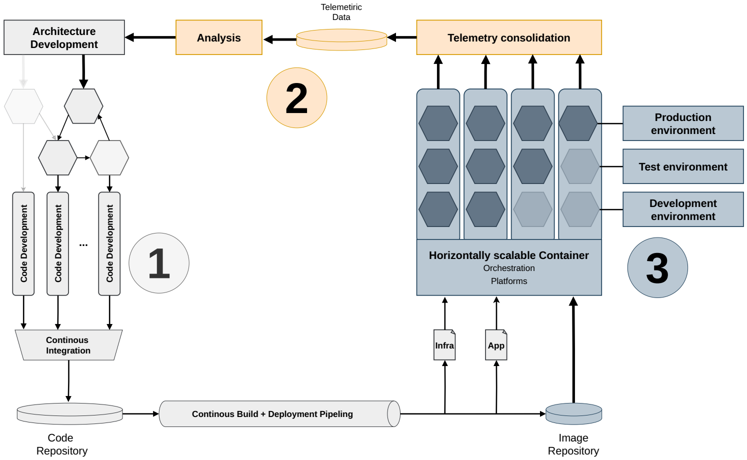
- **Recognize problems early**
  - Complex systems cannot be fully grasped and understood by a single person. Therefore, assumptions made during design should be continuously tested. This is done with chaos engineering, for example, to build confidence in the system's ability to withstand error and failure situations. This requires, among other things, feedback loops in the production system that continuously and automatically collect telemetry data.

DevOps – 2.) Principles of feedback (2/3)



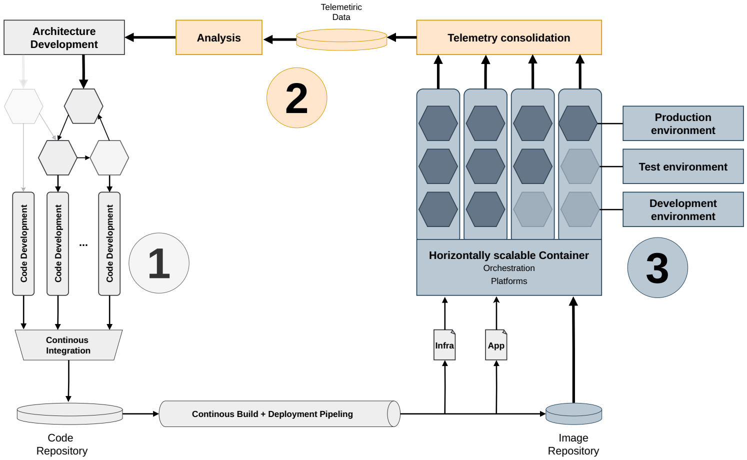
- **Solve problems immediately**
  - This means that problem-solving in the production system should always be given higher priority than further feature development. If problems arise in the production system, the development pipeline should be stopped, and all available development staff should focus on solving the problem in the production system.

DevOps – 2.) Principles of feedback (3/3)



- **Taking professional responsibility for problems**
  - The DevOps philosophy, states that developers also have responsibility for operations (*"you build it, you run it"*). This means that responsibility for quality, reliability, and decision-making authority lies with those who do the work and have the most technical expertise. DevOps teams, therefore, usually have greater autonomy but also greater responsibility than pure development teams, which can throw responsibility *"over the wall"* to operations.

DevOps – 3.) Automated execution environment



3. **Automated execution environment**
   - Container plattform
   - Implementation of Services (Dev)
   - Operation of Services (Ops)

Outlook

~~1st part~~:	~~Introduction~~
~~2nd part~~:	~~Technological foundations~~
~~3rd part~~:	~~Service models, deployment models~~
~~4th part~~:	~~Adoption and strategy~~
~~5th part~~:	~~Architectures and applications~~
6th part:	Cloud-Native applications $\Longleftarrow$ *This slide set*
7th part:	Current and future trends

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

7th part: Current and future trends

Topics:

- **Introduction**

Thank You For Your Attention!

Henry-Norbert Cocos, M.Sc
Frankfurt University of Applied Sciences
Room 1-230
☎ +49 69 1533-2699
✉ `cocos@fb2.fra-uas.de`
`www.henrycocos.de`

FRANKFURT UNIVERSITY OF APPLIED SCIENCES