Einstieg

- Softwaredebakel
- Kundenorientierung
- Anforderungen an moderne Software
- Teamarbeit
- Ab 1961: Margaret Hamilton, Apollo Guidance Computer
- 2001: Manifesto for Agile Software Development
- Software Engineering / Software Architecture
- Learning
- Managing Complexity
- Production Is Not Our Problem
- Space X Starship

Lernen

- Iteratives und inkrementelles Arbeiten
- Iterationen
- Embrace Change
- Feedback
- Empirisches und experimentelles Arbeiten

Kommunikation

- Domain Driven Design
- UML Klassendiagramm
- UML Klassendiagramm
- PlantUML
- C4 Model
- Architectural Decision Records

Komplexität

- Modularity & Separation of Concerns
- Testing
- Cohesion & Coupling
- Abstraction

Architekturen

- Schichtenarchitektur
- Ports and Adapters
- Traditional Monolithic Design
- Schichtenarchitektur im Client Server Modell
- Microservices
- Monolith First
- Event Driven Architecture
- Reactive Systems

- Fallstudie

Cloud Computing

- Abstractions
- XaaS
- What is a Cloud Native application?

Quellen

Einstieg

Softwaredebakel

Softwaredebakel

VBS

- Schweizer Armee ohne krisensichere Logistik bis 2035
- Armee-Debakel: 300-Millionen-Projekt seit Monaten suspendiert

Kantonsverwaltung

- Wegen fehlerhafter Software braucht es mehr Haftplätze

Polizei

- Berner Polizisten beklagen sich über die neue IT

Crowdstrike

- Der Tag, an dem die IT weltweit verrückt spielte – ein Überblick

Kundenorientierung

Kundenorientierung

Software soll den Kunden Mehrwert bringen

- Software soll stabil laufen
- Neue Features sollten schnell umgesetzt und nutzbar sein
- Softwaresysteme werden immer komplexer

Figure 1: w:1000px

Anforderungen an moderne Software

- Hohe Verfügbarkeit
- Skalierbarkeit
- Im Katastrophenfall sollen die Systeme schnell wiederhergestellt werden können
- Soll funktionieren, auch wenn Teile des Systems Offline sind (Resilienz)
- Kostengünstig
- Einfach
- Updates müssen einfach eingespielt werden können

Teamarbeit

Teamarbeit

Mehrere Personen arbeiten am selben Softwareprojekt

- Versionsverwaltung wird verwendet (Git, SVN)
- Konflikte entstehen und sind aufwendig

Ab 1961: Margaret Hamilton, Apollo Guidance Computer

Ab 1961: Margaret Hamilton, Apollo Guidance Computer

2001: Manifesto for Agile Software Development

2001: Manifesto for Agile Software Development

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

https://agilemanifesto.org/

Software Engineering / Software Architecture

Software Engineering / Software Architecture

> *Software engineering is the application of an empirical, scientific approach to finding efficient, economic solutions to practical problems in software*

(Farley, 2022, S.4)

> *The goal of software architecture is to minimize the human resources required to build and maintain the required system*

(Martin, 2018)

Übergang zwischen Software Entwicklung und Software Architektur ist fliessend

Learning

Learning

- Iteratives und inkrementelles Arbeiten
- Feedback
- Empirisches und experimentelles Arbeiten

(vgl. Farley, 2022, S.4)

Managing Complexity

Managing Complexity

- Modularity & Separation of Concerns
- Cohesion & Coupling
- Abstraction

(vgl. Farley, 2022, S.5)

Production Is Not Our Problem

Production Is Not Our Problem

- Softwareentwicklung ist meistens Kreativarbeit
- Die Herausforderung der “Produktion” existiert kaum

Space X Starship

Space X Starship

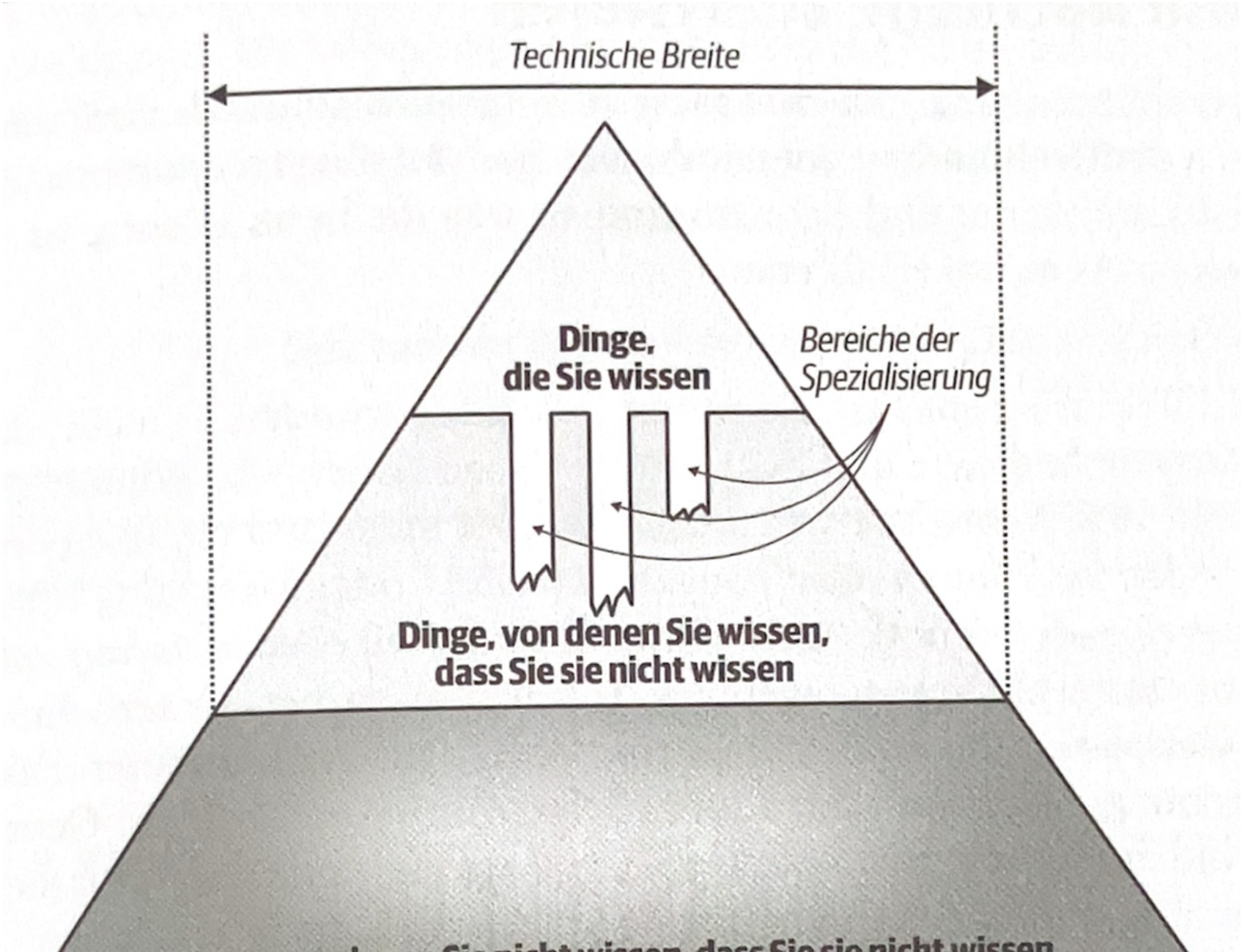
How Not to Land an Orbital Rocket Booser, 2017 WOW! Watch SpaceX Catch A Starship Booster In Air, 2024

Finanzierung: **ca 3 Mrd. Dollar**

Apollo-Programm: 1958 bis 1969, inflationsbereinigt: **163 Mrd. Dollar** (ohne Mercury und Gemini)

Lernen

Lernen


Technische Breite
Dinge,
die Sie wissen
Bereiche der
Spezialisierung
Dinge, von denen Sie wissen,
dass Sie sie nicht wissen

Iteratives und inkrementelles Arbeiten

Iteratives und inkrementelles Arbeiten

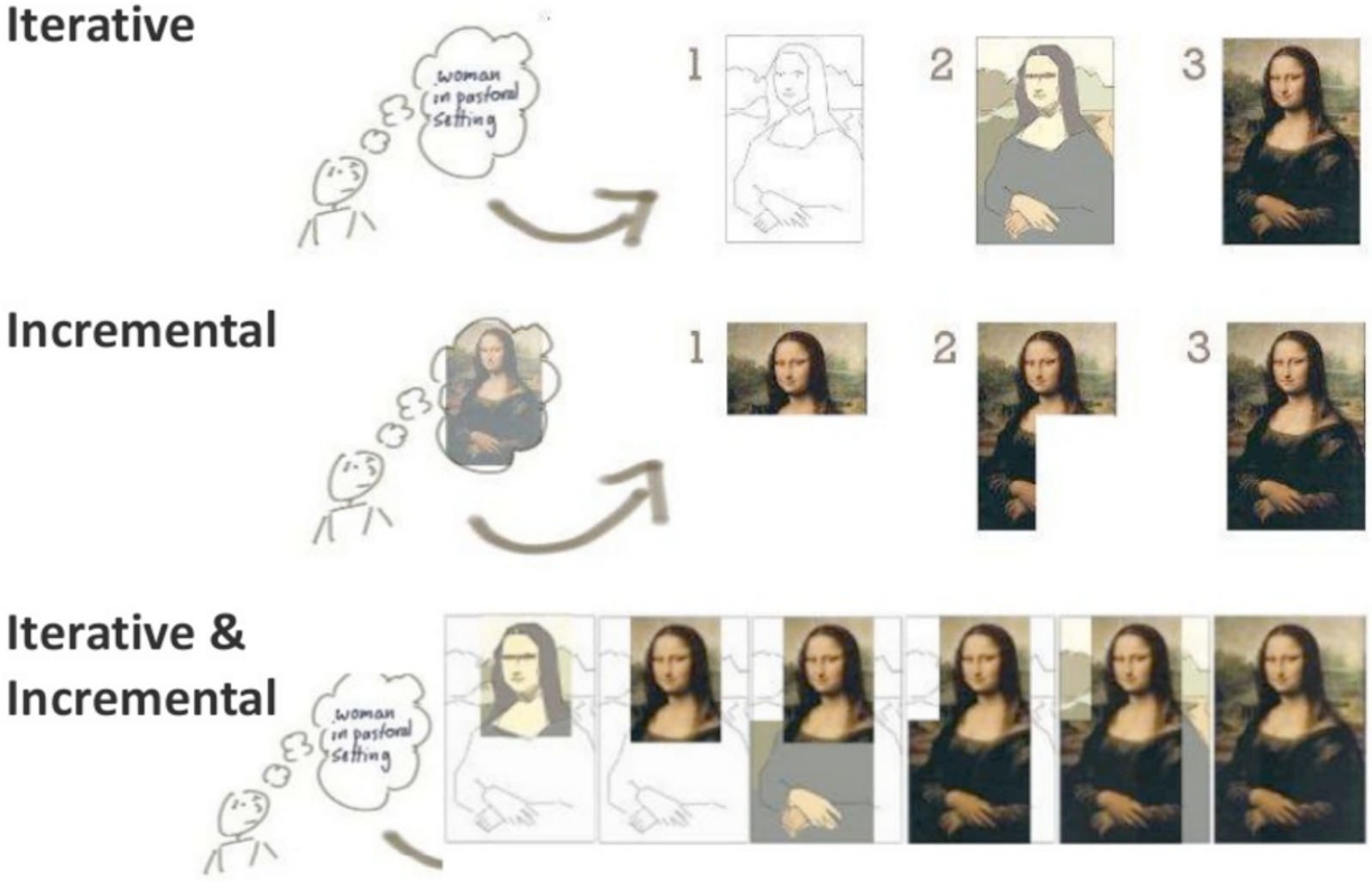


Figure 3: w:800px Mona Lisa

Iterationen

Iterationen

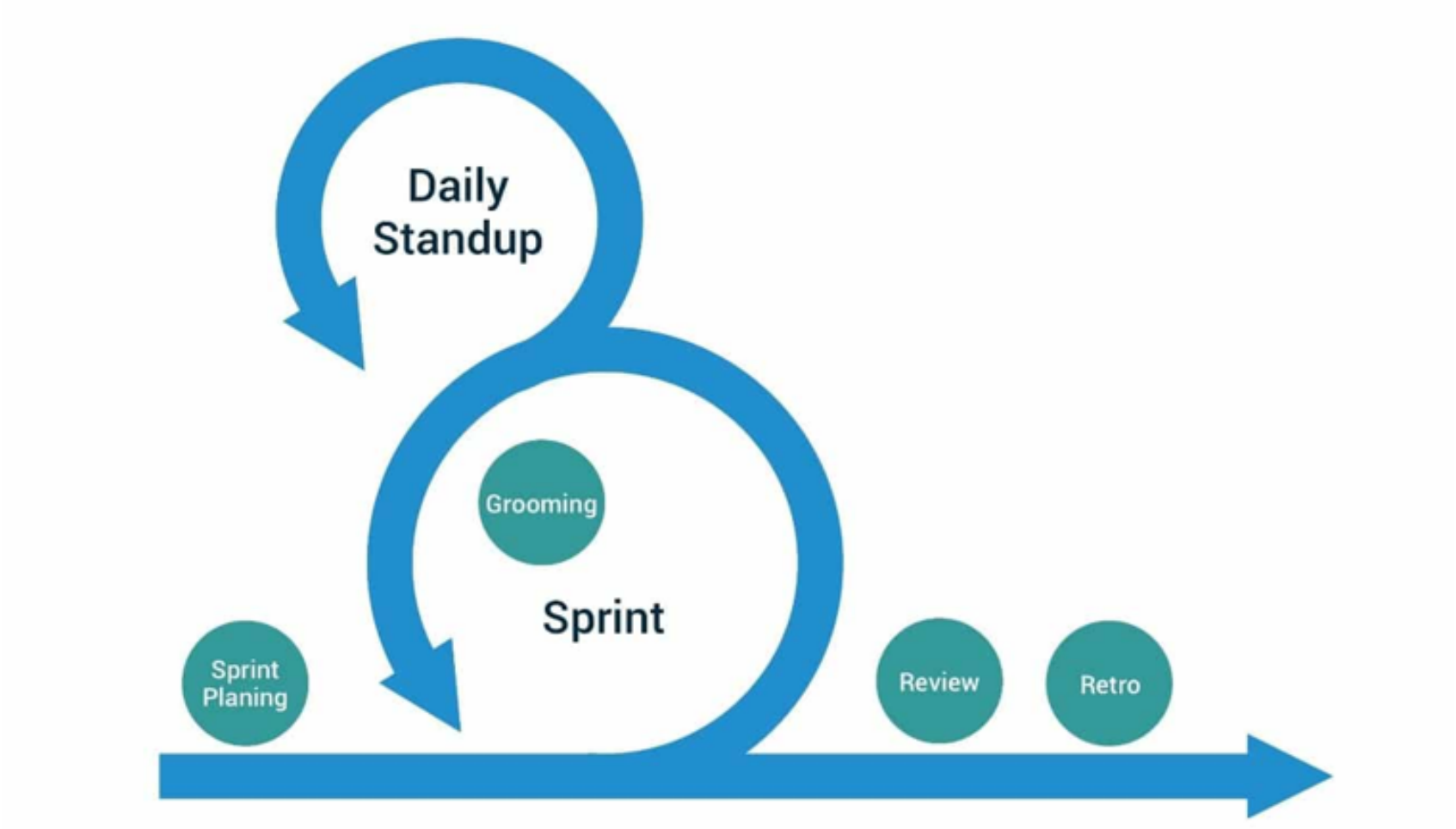


Figure 4: Iterations

Embrace Change

Embrace Change

Extreme Programming

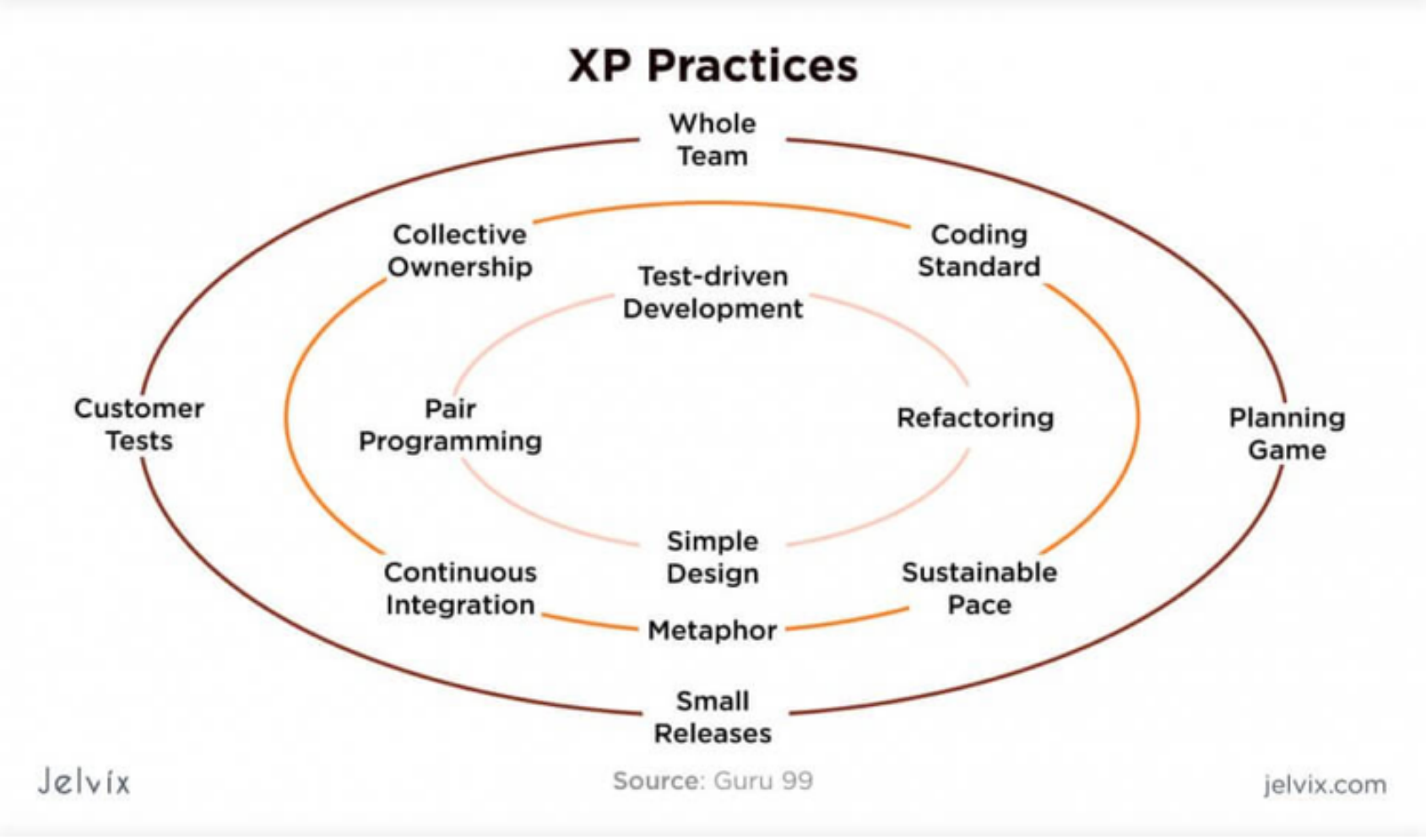


Figure 6: Extreme Programming

Feedback

CI/CD

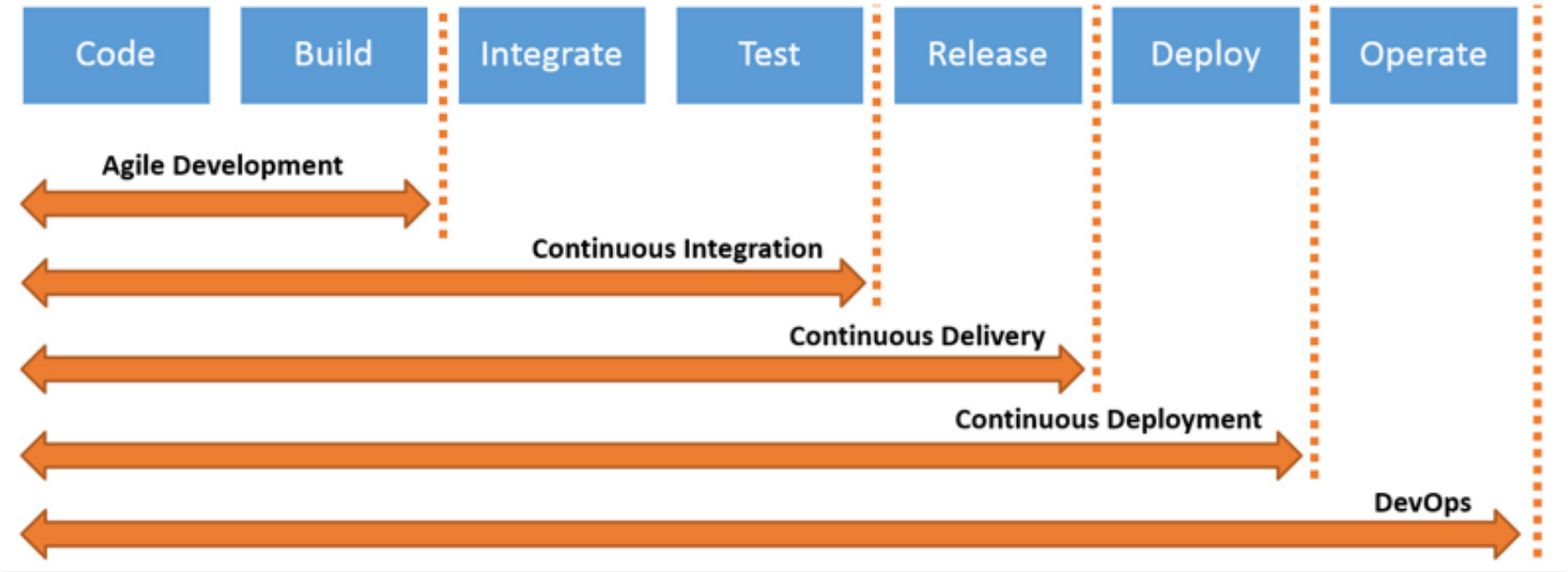


Figure 7: img.png

Continuous Integration

- **Kein Branching**, alle Änderungen werden von allen Teammitgliedern **mehrmals täglich** in den Master Branch eingecheckt.
- Dieser Branch ist **jederzeit lauffähig**
- Dadurch werden die **Releases vereinfacht**
- Eine sehr hohe, **automatische Testabdeckung** ist zwingend

Continuous Deployment

- Ziel: **Releases werden vereinfacht**
- **Time to market ist kürzer**, neue Features sind sofort verfügbar
- Durch automatisierte Deployments ist der Aufwand initial höher, anschliessend jedoch sehr klein
- **Higher quality**, **Better products**
- Kaum mehr Release-Stress, **Happier teams**

https://www.continuousdelivery.com/

Deployment Pipelines

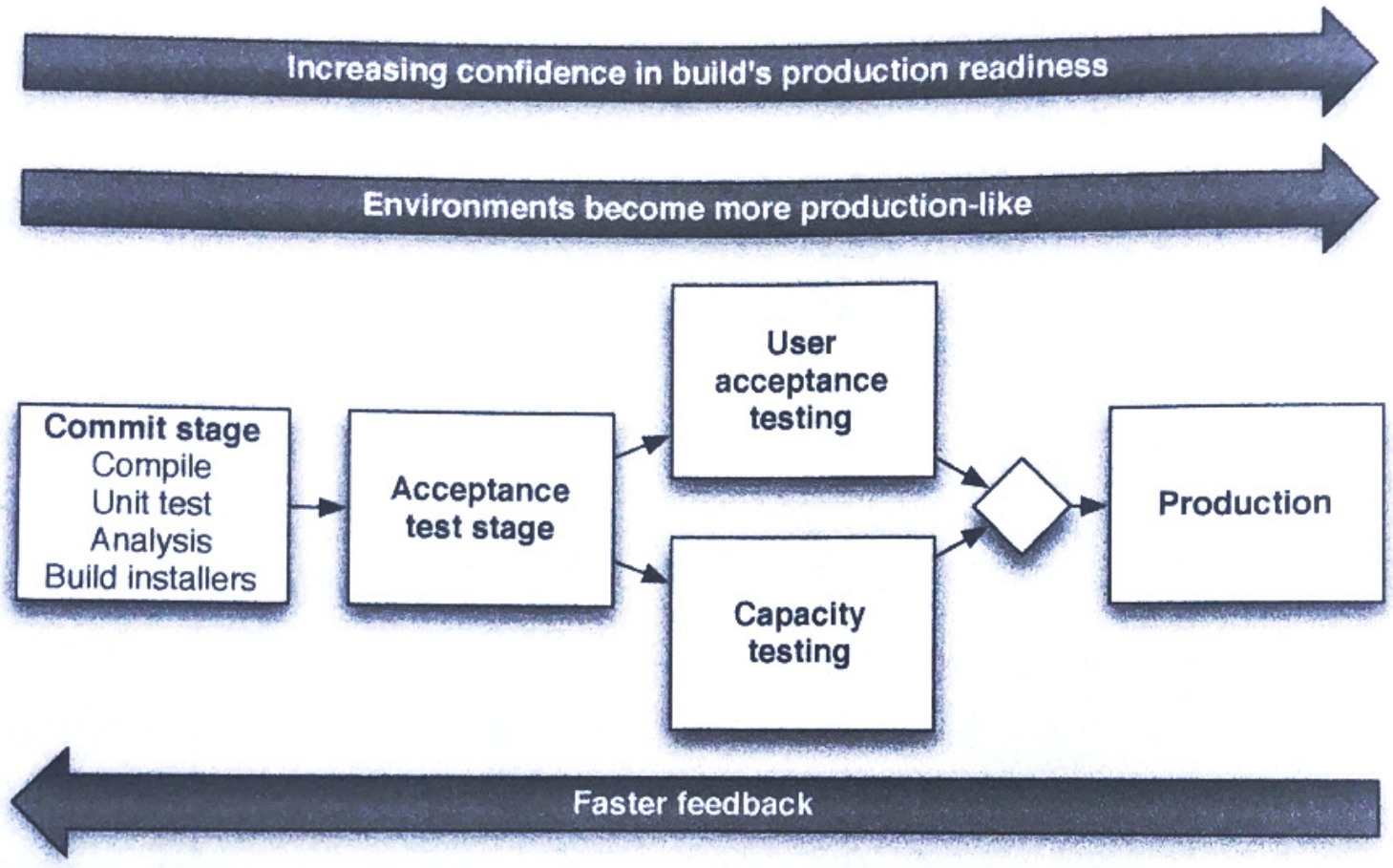
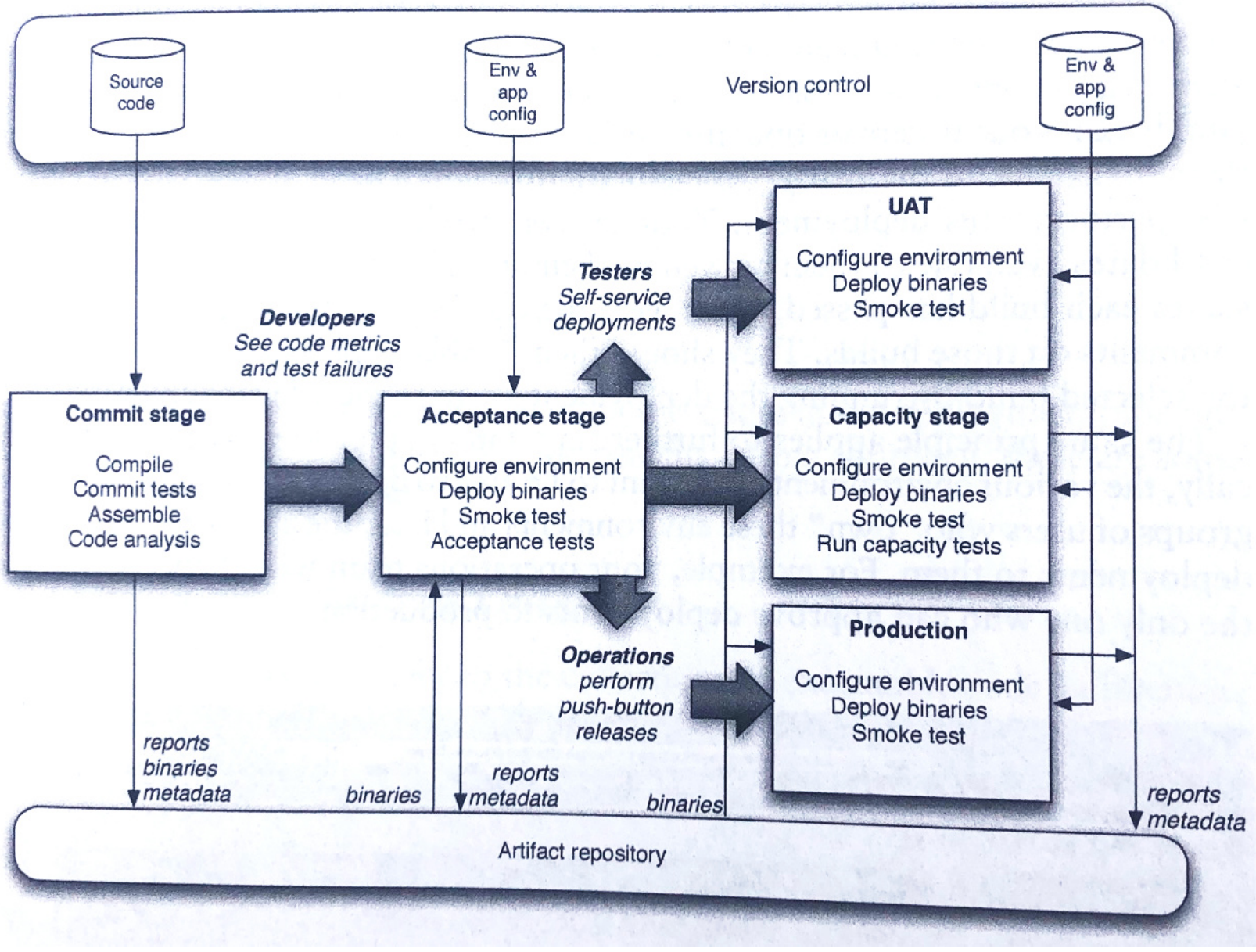


Figure 5.3 *Trade-offs in the deployment pipeline*

(Jez Humble, David Farley (2010): Continuous Delivery)



(Jez Humble, David Farley (2010): Continuous Delivery)

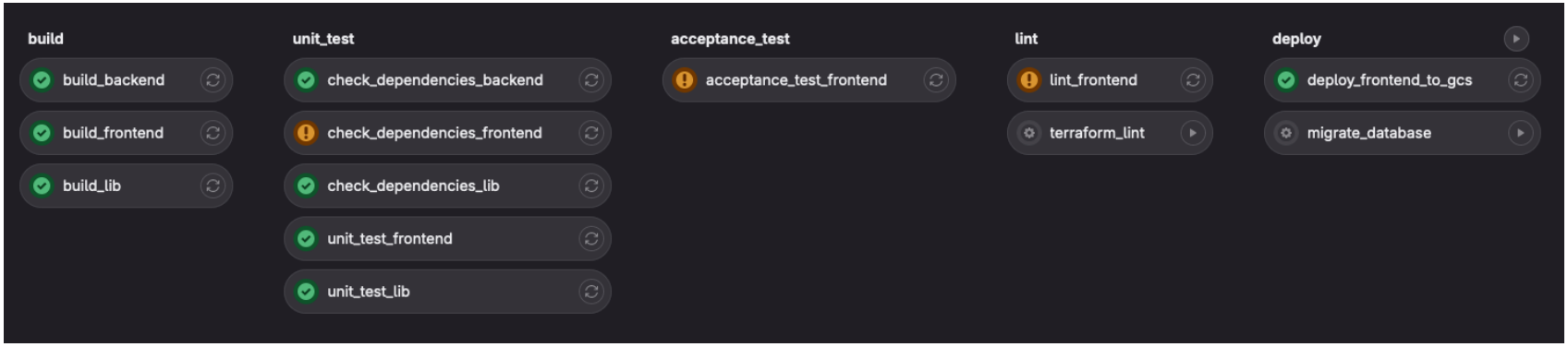


Figure 8: w:1000px

- Youtube: Continous Delivery - Deployment Pipelines
- Jez Humble, David Farley (2010): Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley Signature Series (Fowler)

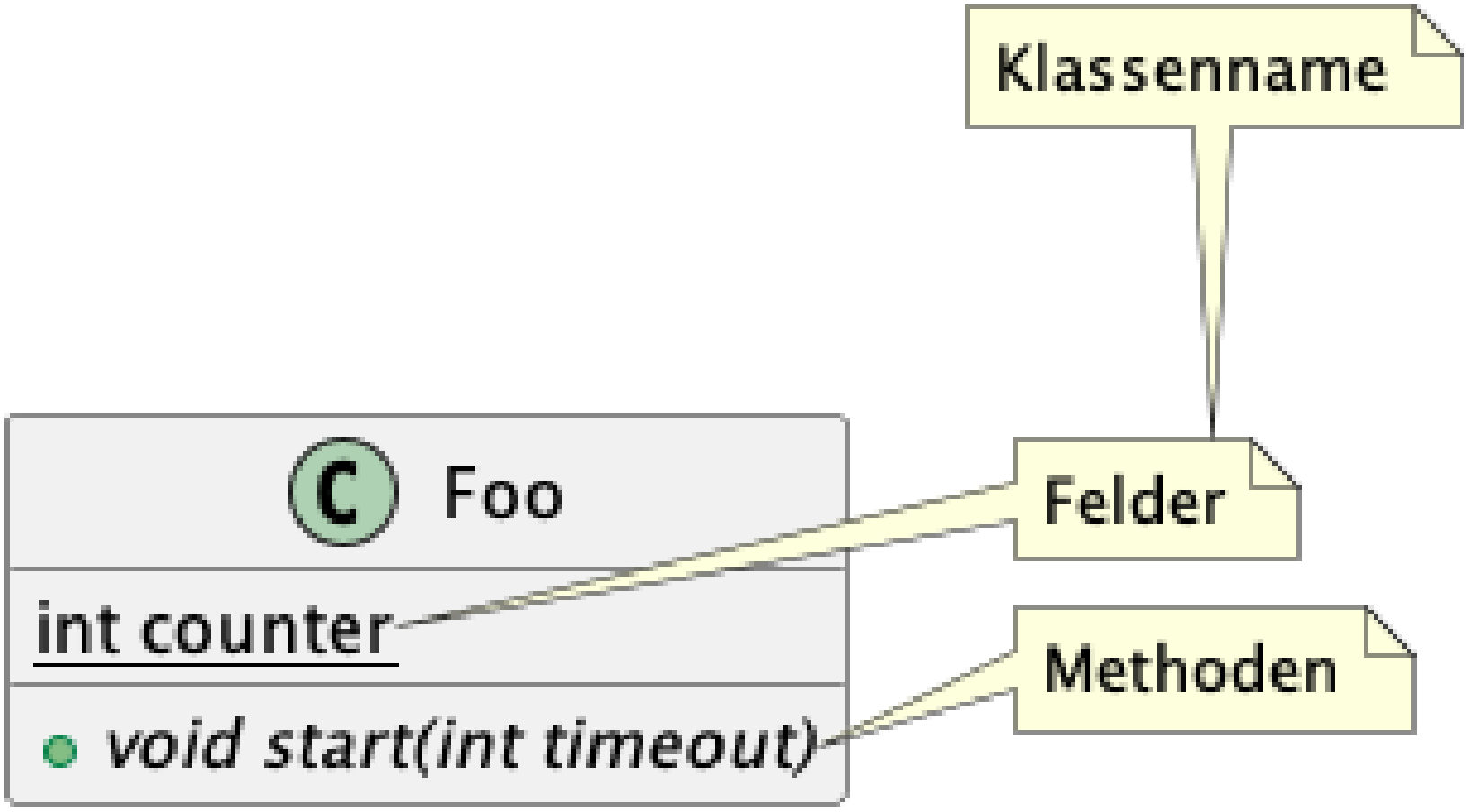
Empirisches und experimentelles Arbeiten

Kommunikation

Domain Driven Design

UML Klassendiagramm

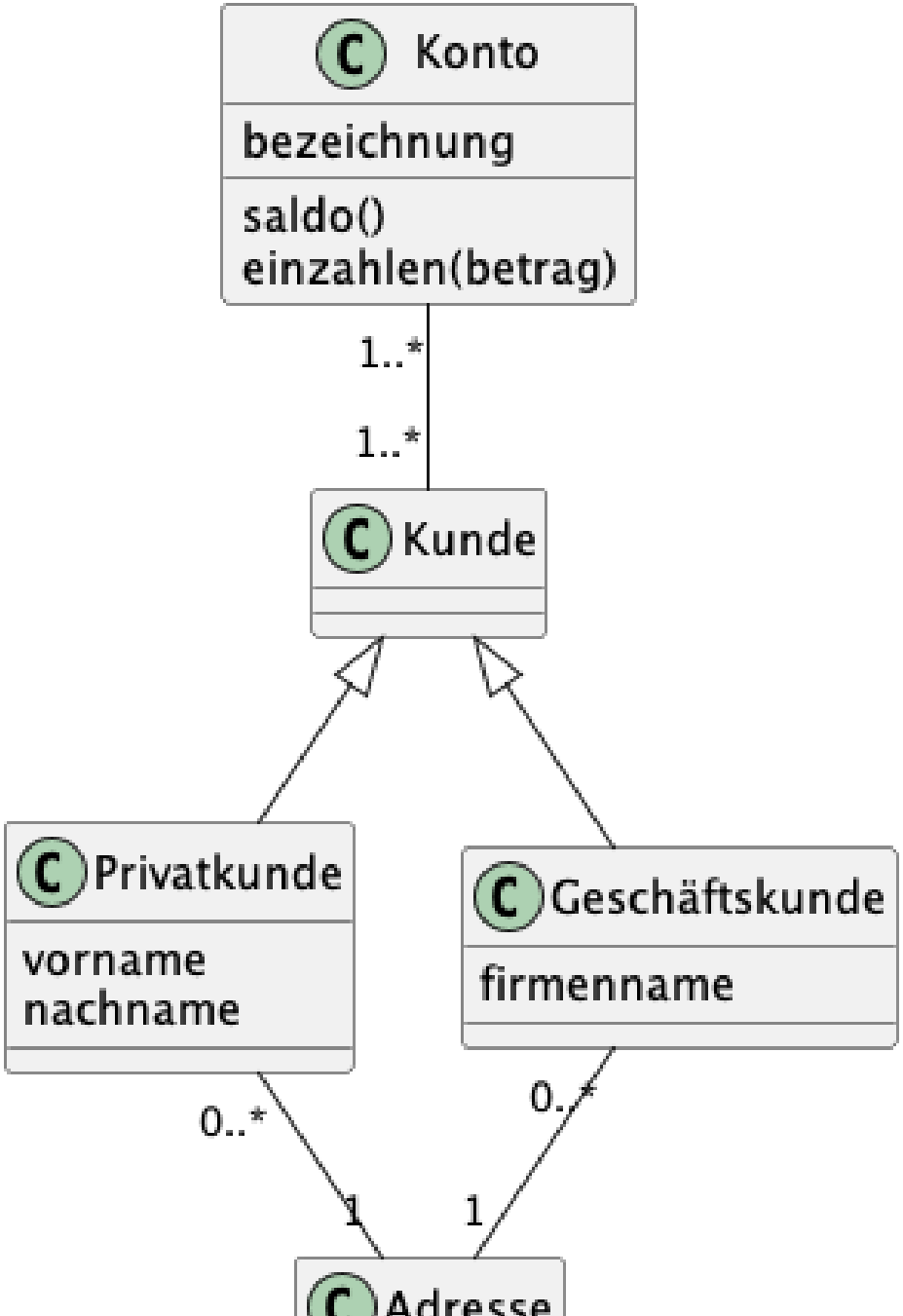
UML Klassendiagramm



PlantUML

UML Klassendiagramm

UML Klassendiagramm



PlantUML

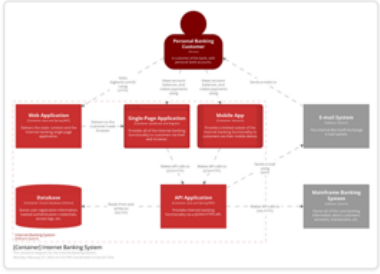
PlantUML

```
@startuml
class Konto {
    bezeichnung
    saldo()
    einzahlen(betrag)
}

class Kunde {
}

class Privatkunde {
    vorname
    nachname
}

class Geschäftskunde {
    firmenname
}
```

C4 Model

C4 Model

Level 1: A **System Context** diagram provides a starting point, showing how the software system in scope fits into the world around it.



Level 2: A **Container** diagram zooms into the software system in scope, showing the high-level technical building blocks.

Level 3: A **Component** diagram zooms into an individual container, showing the components inside it.

Level 4: A **code** (e.g. UML class) diagram can be used to zoom into an individual component, showing how that component is implemented.

https://c4model.com/

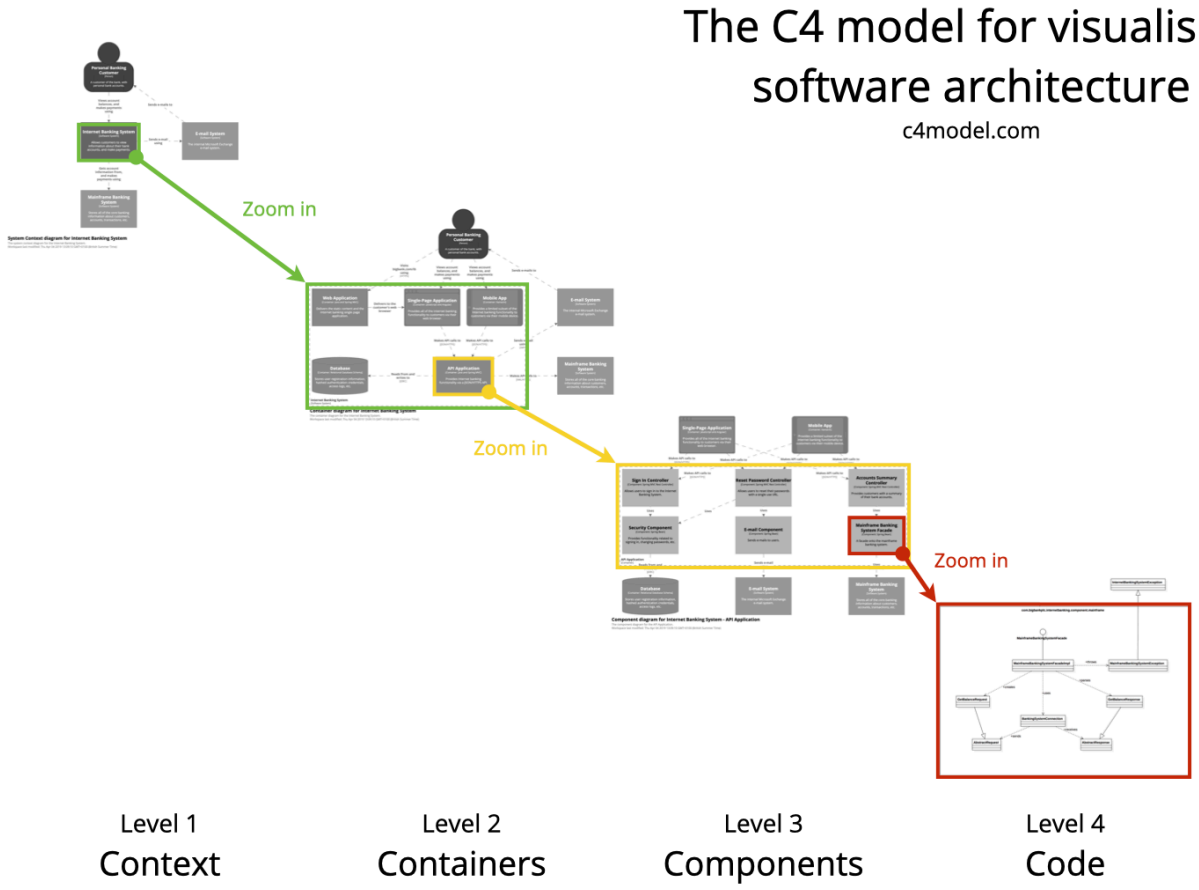
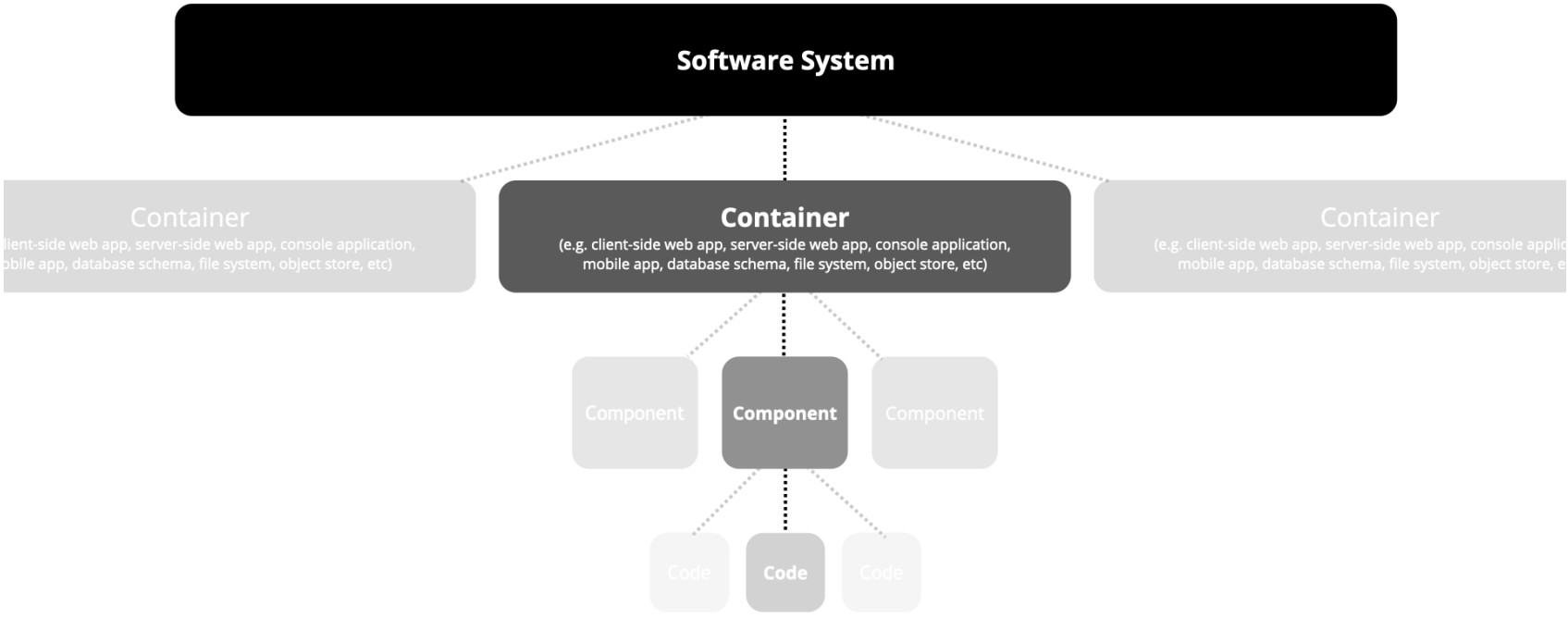


Figure 9: img.png



A **software system** is made up of one or more **containers** (applications and data stores), each of which contains one or more **components**, which in turn are implemented by one or more **code** elements (classes, interfaces, objects, functions, etc).

Figure 10: img.png

Architectural Decision Records

Architectural Decision Records

```
# <!-- short title, representative of solved problem

## Context and Problem Statement

## Considered Options

## Decision Outcome

### Consequences
```

- https://github.com/adr/madr/blob/4.0.0/template/adr-template-bare-minimal.md
- https://github.com/adr/madr/blob/4.0.0/template/adr-template-bare.md

Templates

- Nygard: https://github.com/joelparkerhenderson/architecture-decision-record/blob/main/locales/en/templates/decision-record-template-by-michael-nygard/index.md
- MADR: https://github.com/adr/madr/blob/4.0.0/template/adr-template.md

Tools

- https://github.com/npryce/adr-tools
- https://github.com/opinionated-digital-center/pyadr

Komplexität

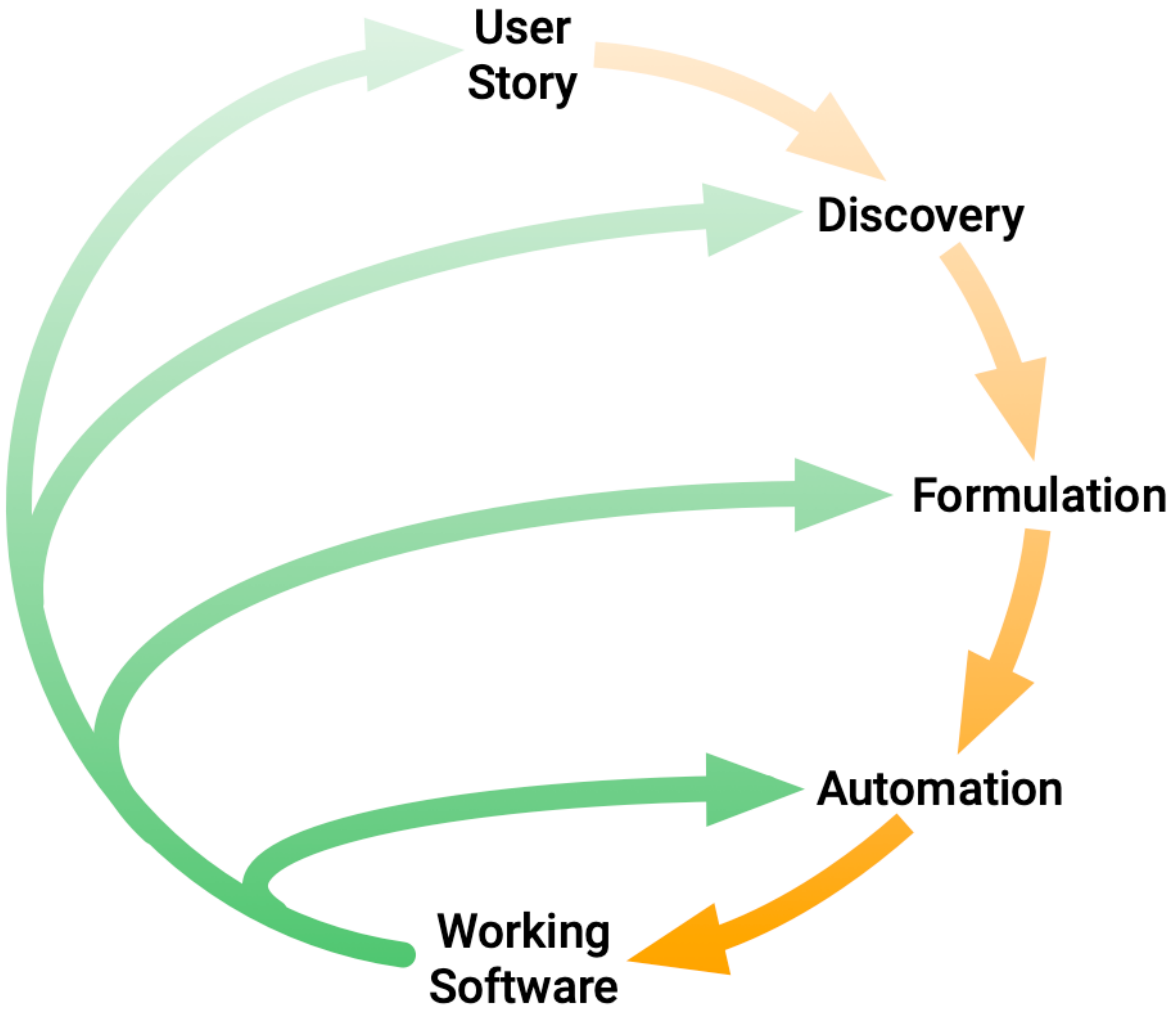
Modularity & Separation of Concerns

Testing

Testing

> *The hardest single part of building a software system is deciding precisely what to build.*

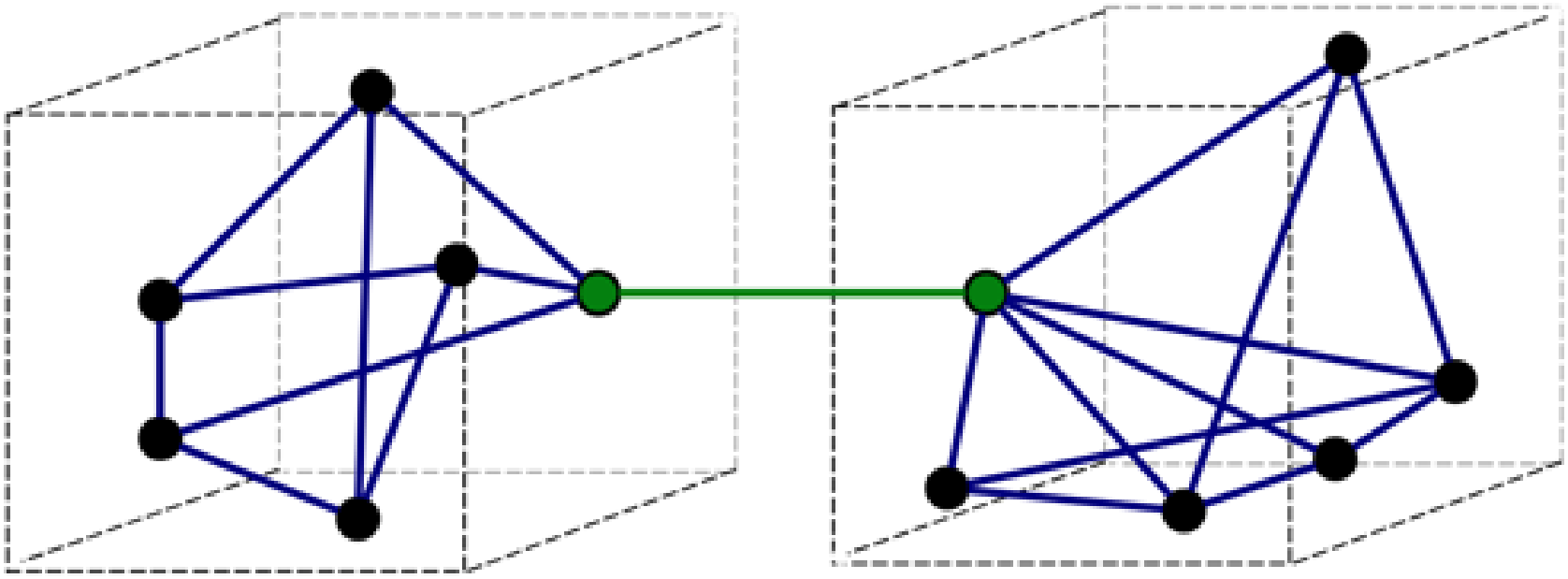
– Fred Brooks, The mythical man-month



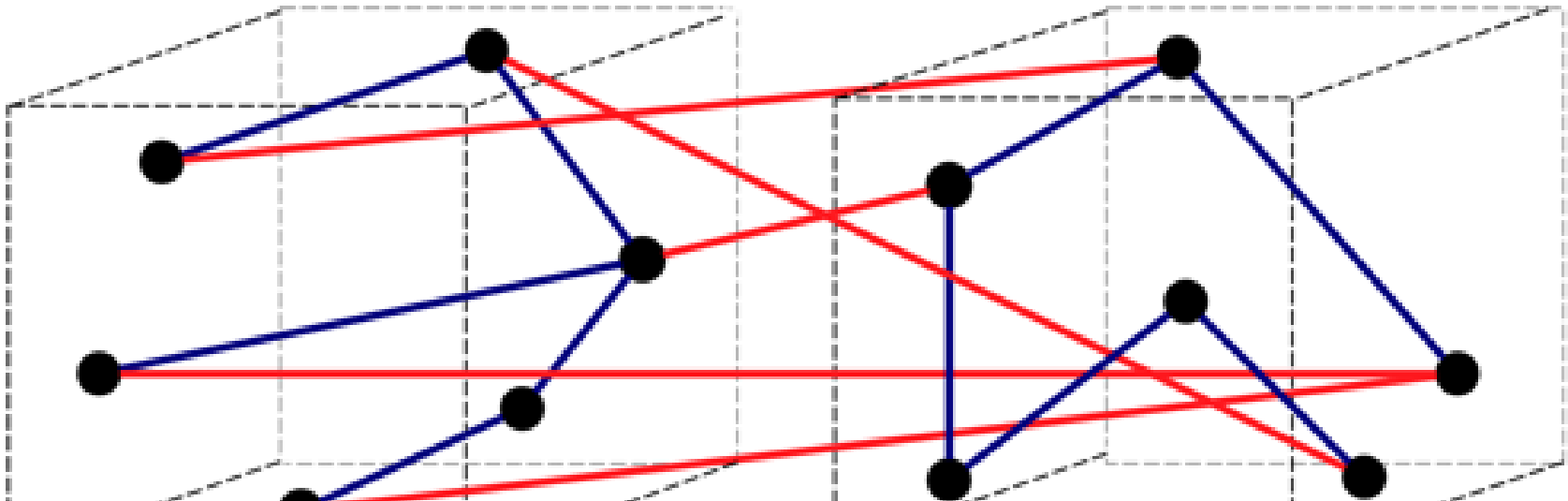
https://cucumber.io/docs/bdd/

Cohesion & Coupling

Cohesion & Coupling



a) Good (loose coupling, high cohesion)

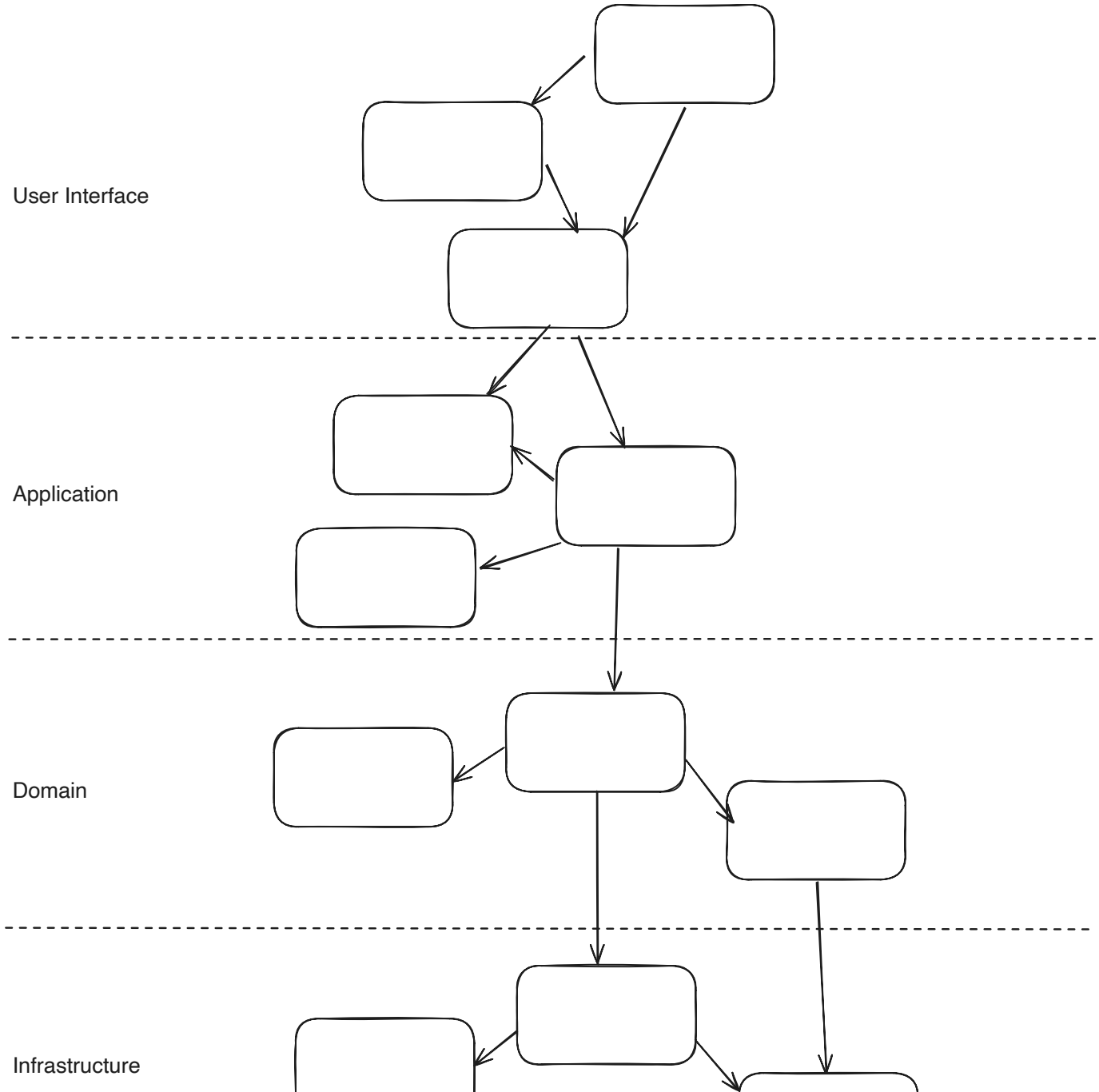


Abstraction

Architekturen

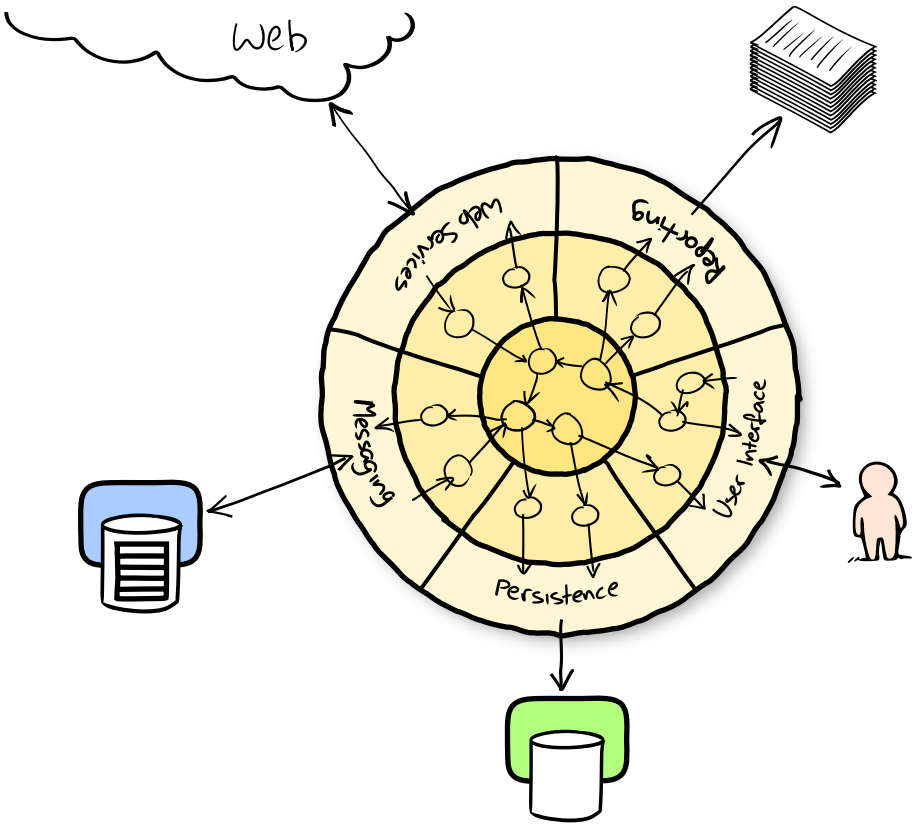
Schichtenarchitektur

Schichtenarchitektur



Ports and Adapters

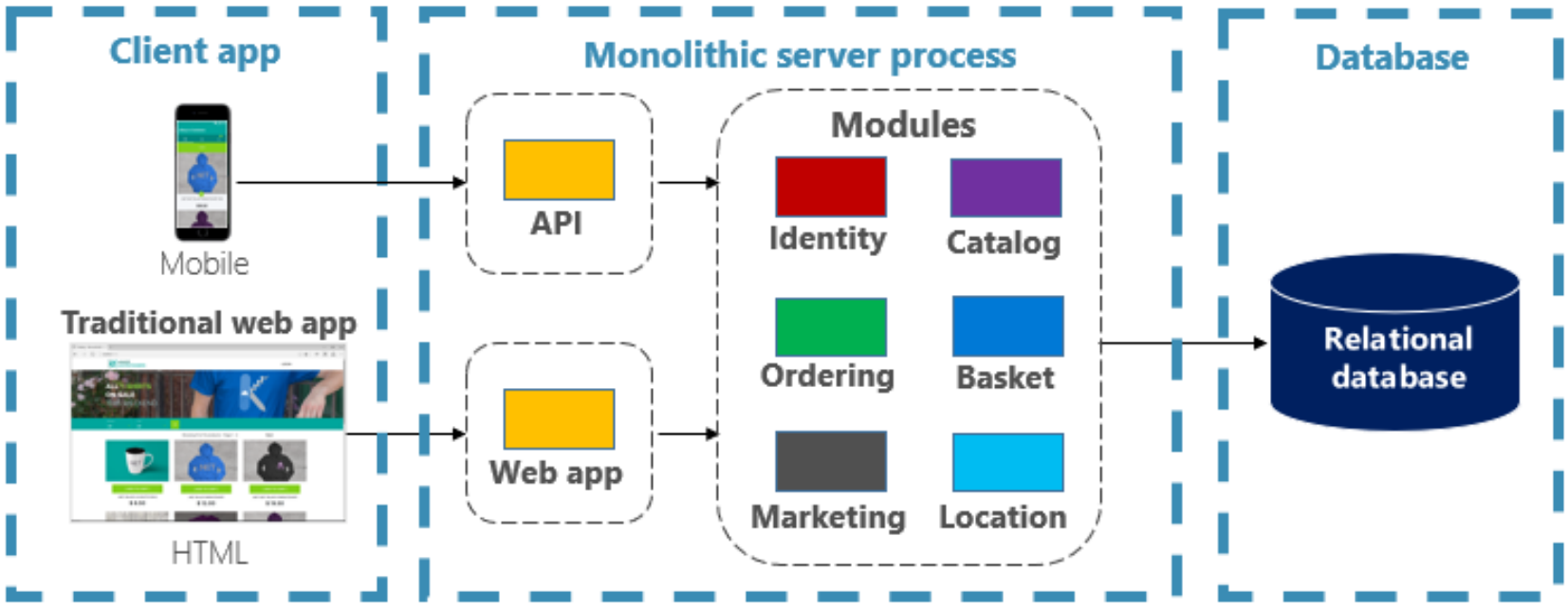
Ports and Adapters



growing-object-oriented-software.com

Traditional Monolithic Design

Traditional Monolithic Design



Schichtenarchitektur im Client Server Modell

Schichtenarchitektur im Client Server Modell

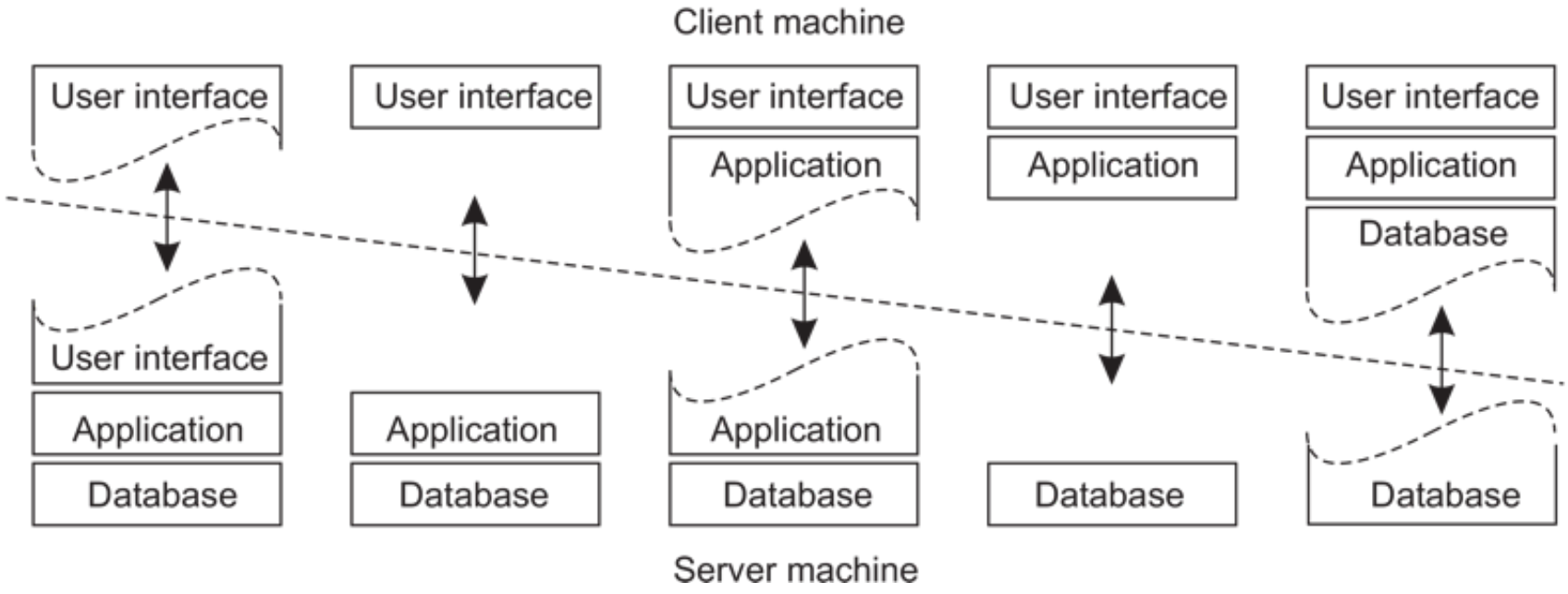


Figure 13: w:1000px

Microservices

Microservices

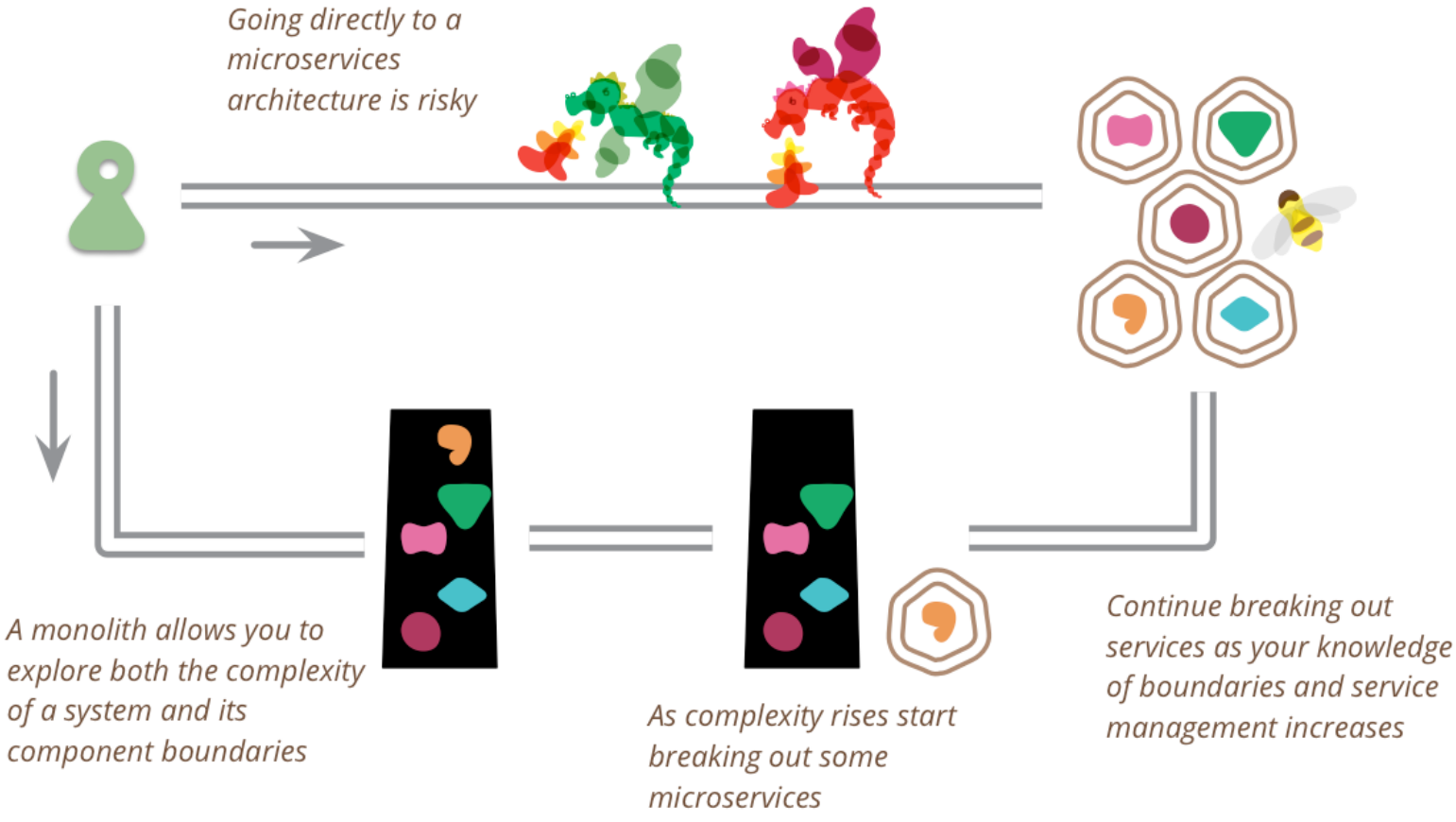
- Maximale Skalierbarkeit
- Einzelne Services können von **kleinen[^1]** Teams **unabhängig entwickelt und deployed** werden
- Bessere Wart- und Erweiterbarkeit
- Unterschiedliche Technologien können eingesetzt werden
- Kommunikation nicht trivial
- Höhere Wahrscheinlichkeit eines Ausfalls
- **Hohe Komplexität**

martinfowler.com/articles/microservices.html [^1]: "We try to create teams that are no larger than can be fed by two pizzas"

Monolith First

Monolith First

- Vorsicht vor Cargo-Kult: Amazon, Google, Meta etc. haben heute andere Herausforderungen als Startups
- Technologien oder Architekturen wählen, “weil Google macht das auch so” ist ein schlechter Grund



martinfowler.com/bliki/MonolithFirst.html

Event Driven Architecture

Reactive Systems

Reactive Systems

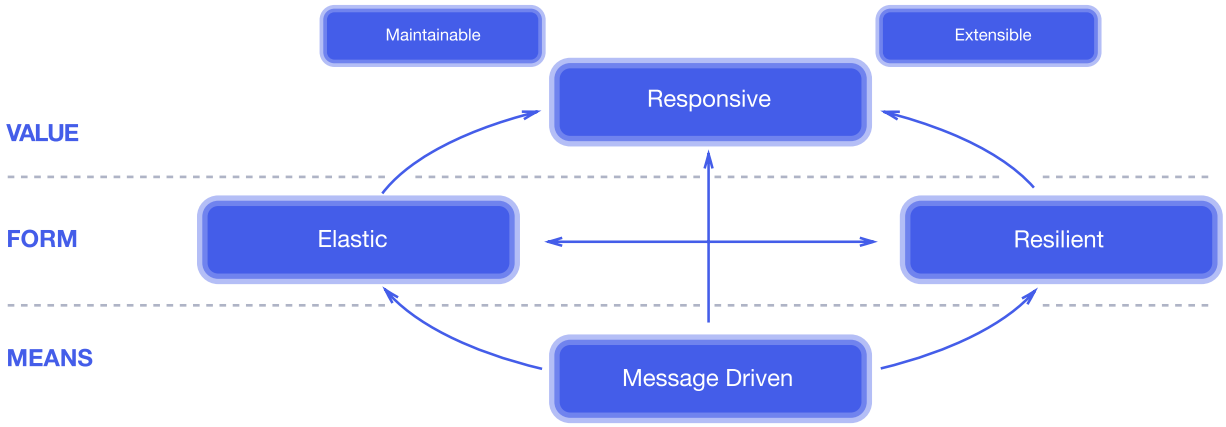


Figure 14: w:1000px

Fallstudie

Fallstudie

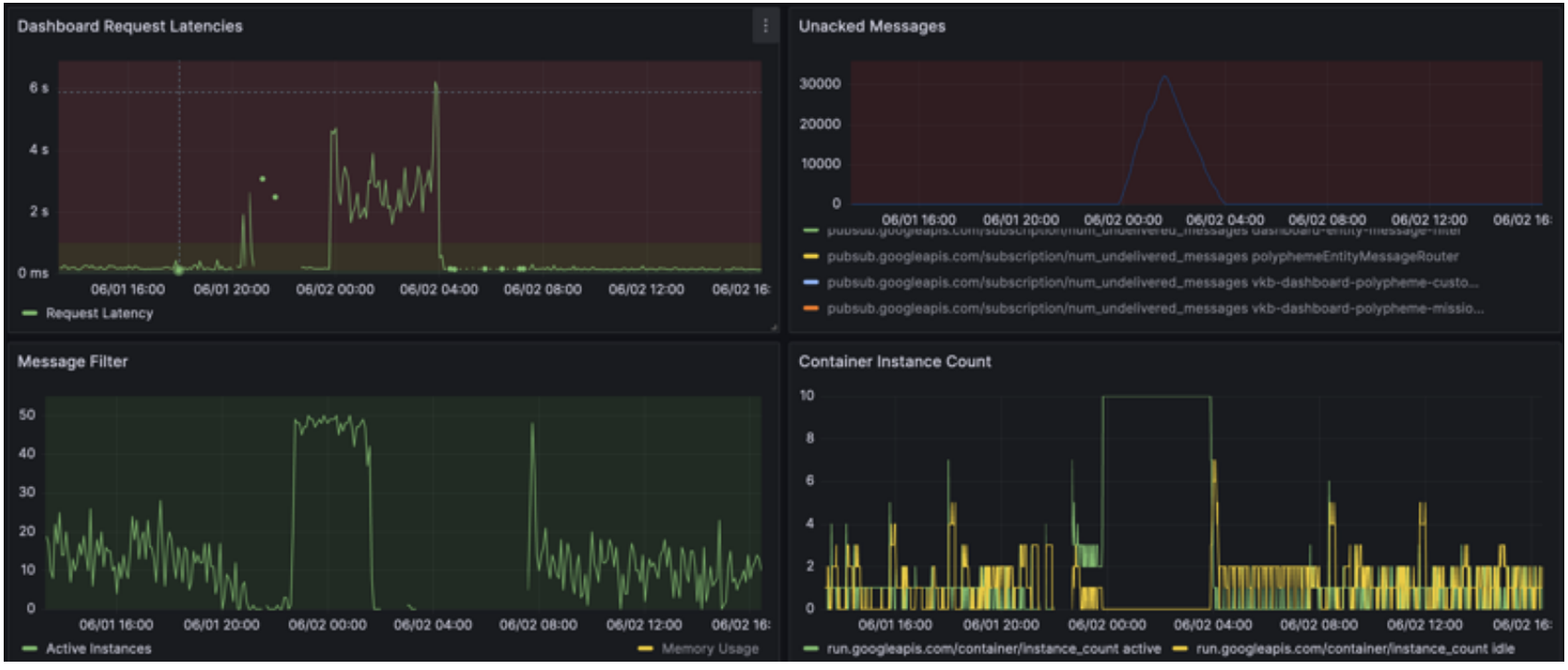


Figure 15: w:1000px

Cloud Computing

Cloud Computing

The entire history of software engineering is that of the rise in levels of abstraction.

– Grady Booch

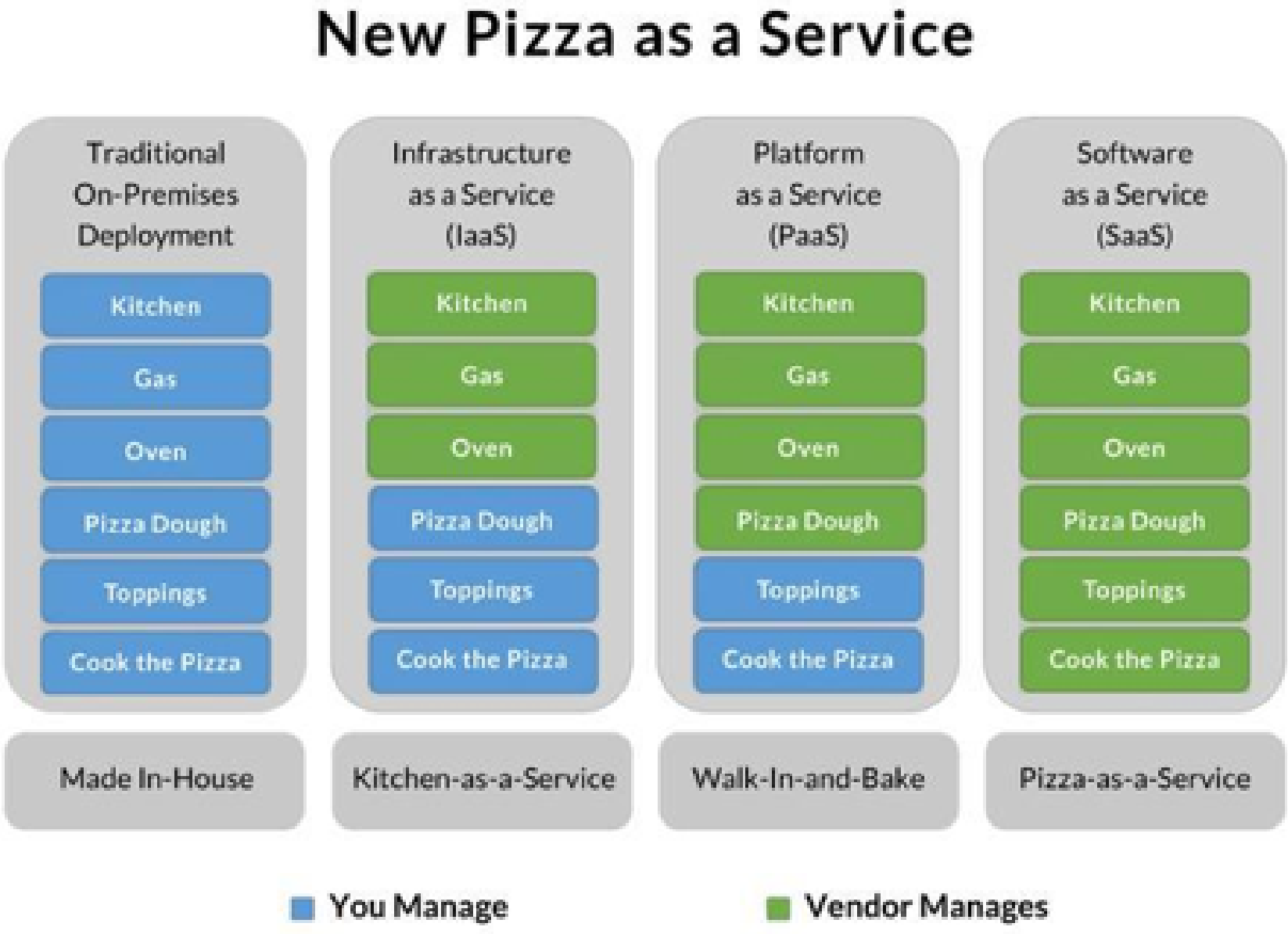
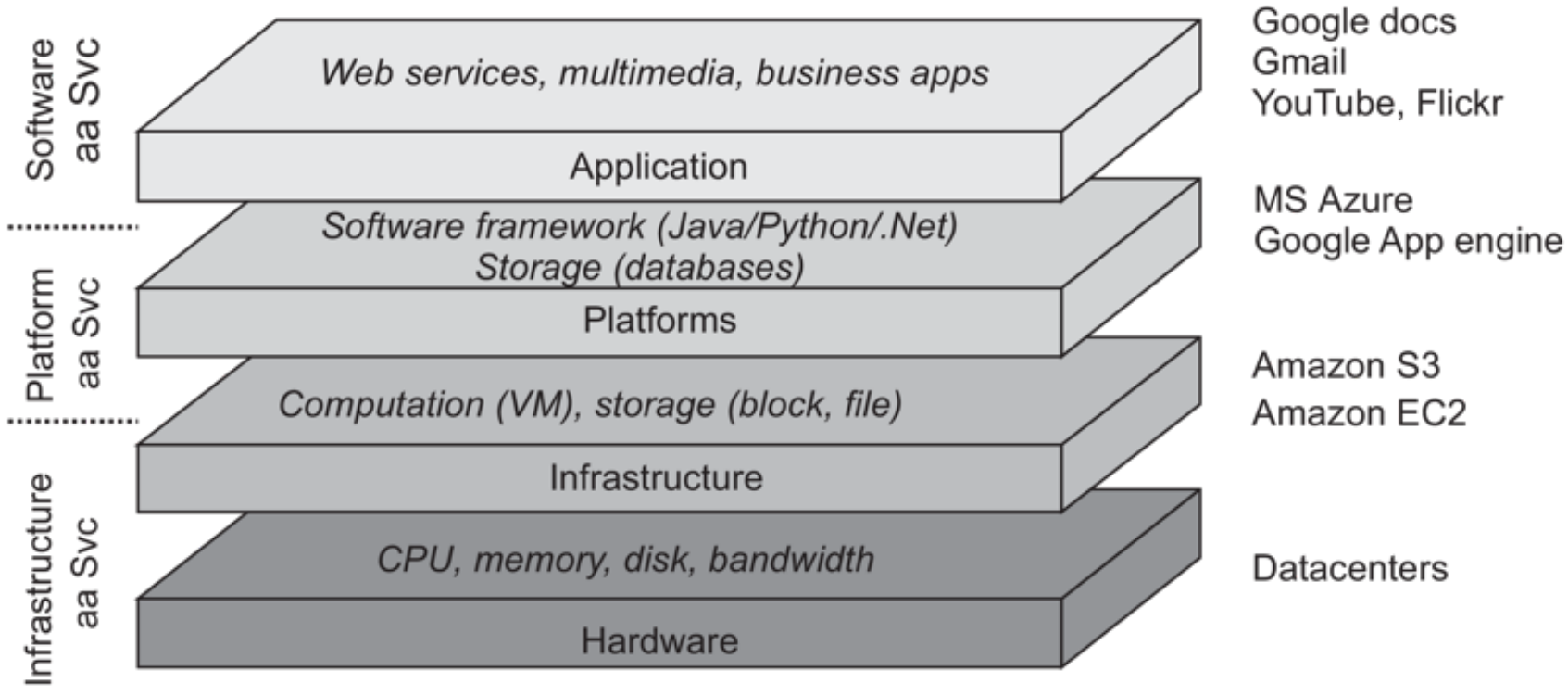


Figure 16: w:800px

Abstractions

Abstractions



(VanSteen, 2017, S. 30)

XaaS

XaaS

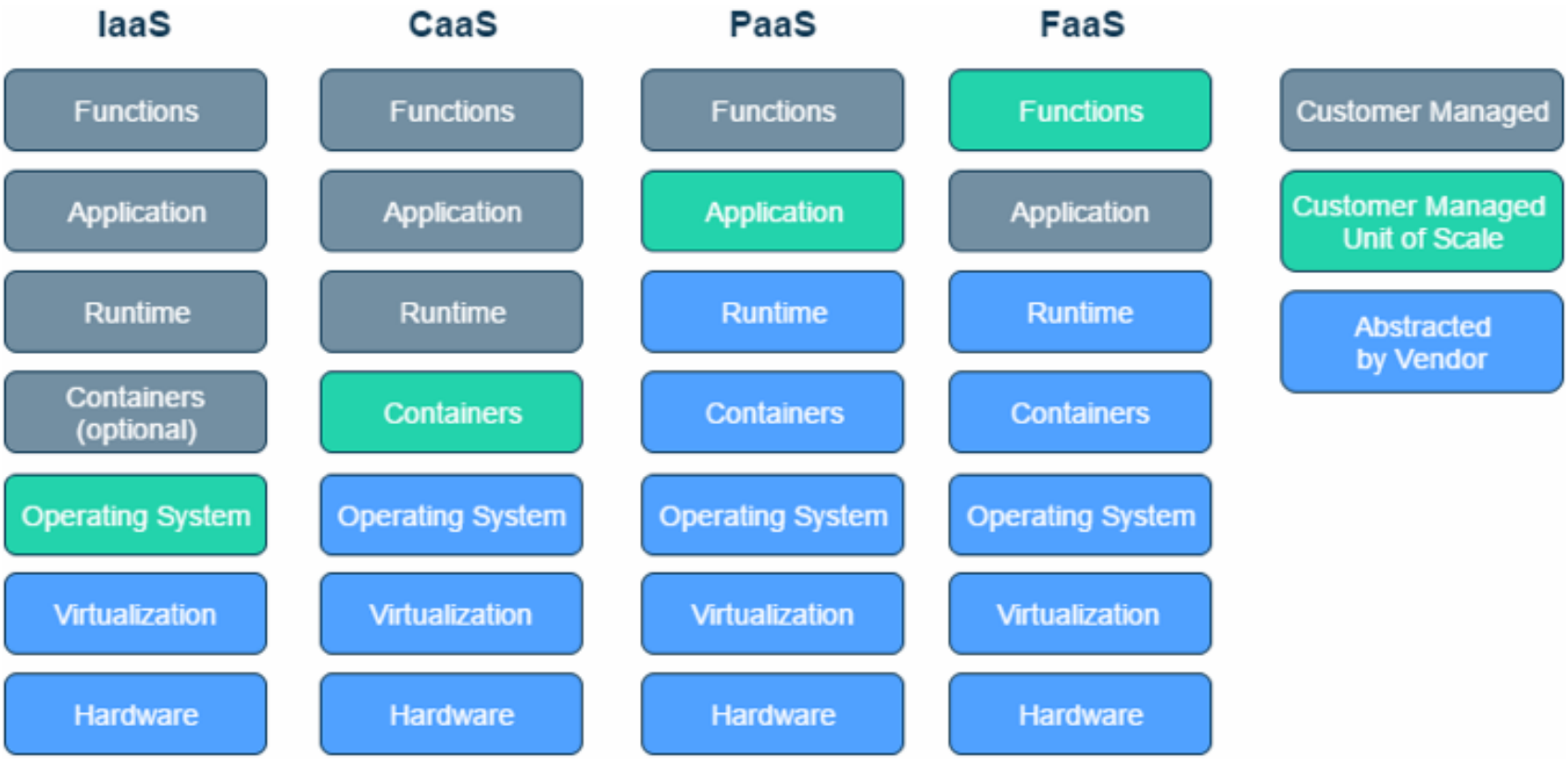


Figure 17: w:1000px

What is a Cloud Native application?

What is a Cloud Native application?

A “cloud native” application, like all native species, has adapted and evolved to be maximally efficient in its environment: the cloud.

The cloud is a harsher environment for applications than those of the past, in particular, than the idealistic environment of a dedicated single node system.

In the cloud, an application becomes distributed. Thus, it is forced to be resilient to hardware/network unpredictability and unreliability, i.e., from varying performance to all-out failure.

https://www.reactiveprinciples.org/cloud-native/index.html

The bad news is that ensuring responsiveness and reliability in this harsh environment is difficult.

The good news is that the applications we build after embracing this environment better match how the real world actually works.

This in turn, provides better experiences for our users, whether humans or software.

https://www.reactiveprinciples.org/cloud-native/index.html

The constraints of the cloud environment, that make up the "cloud operating model," include:

- Applications are limited in the ability to scale vertically on commodity hardware which typically leads to having many isolated autonomous services (often called microservices).
- All inter-service communication takes place over unreliable networks.
- You must operate under the assumption that the underlying hardware can fail or be restarted or moved at any time.
- The services need to be able to detect and manage failure of their peers—including partial failures.
- Strong consistency and transactions are expensive. Because of the coordination required, it is difficult to make services that manage data available, performant, and scalable.

Therefore, a Cloud Native application is designed to leverage the cloud operating model.

It is predictable, decoupled from the infrastructure, right-sized for capacity, and enables tight collaboration between development and operations.

It can be decomposed into loosely-coupled,
independently-operating services that are resilient from failures, driven by data, and operate intelligently across geographic regions.

While Cloud Native applications always have a clean separation of state and compute, there are two major classes of Cloud Native applications: stateful and stateless.

Each class addresses and excels in a different set of use-cases; non-trivial modern Cloud Native applications are usually a combination and composition of the two.

Quellen

Quellen

Farley, 2022 David Farley (2022): Modern Software Engineering: Doing What Works to Build Better Software Faster, Addison-Wesley

Martin, 2018 Robert C. Martin (2018): Clean Architecture: A Craftman's Guide to Software Structure and Design, Prentice Hall

Richards, 2021 Mark Richards, Neal Ford (2021): Handbuch moderner Softwarearchitektur: Architekturstile, Patterns und Best Practices, O'Reilly, 978-3-96009-149-3