Curriculum for

Certified Professional for Software Architecture (CPSA)®
Advanced Level

Module
FLEX

Flexible Architecture Models - Microservices and Self-Contained Systems

2024.1-RC0-EN-20240909

Table of Contents

The abbreviation "e. V." is part of the iSAQB's official name and stands for "eingetragener Verein" (registered association), which describes its status as a legal entity according to German law. For the purpose of simplicity, iSAQB e. V. shall hereafter be referred to as iSAQB without the use of said abbreviation.

List of Learning Goals

Introduction: General information about the iSAQB Advanced Level

What is taught in an Advanced Level module?

- The iSAQB Advanced Level offers modular training in three areas of competence with flexibly designable training paths. It takes individual inclinations and priorities into account.
- The certification is done as an assignment. The assessment and oral exam is conducted by experts appointed by the iSAQB.

What can Advanced Level (CPSA-A) graduates do?

CPSA-A graduates can:

- Independently and methodically design medium to large IT systems
- In IT systems of medium to high criticality, assume technical and content-related responsibility
- Conceptualize, design, and document actions to achieve quality requirements and support development teams in the implementation of these actions
- Control and execute architecture-relevant communication in medium to large development teams

Requirements for CPSA-A certification

- Successful training and certification as a Certified Professional for Software Architecture, Foundation Level® (CPSA-F)
- At least three years of full-time professional experience in the IT sector; collaboration on the design and development of at least two different IT systems
  - Exceptions are allowed on application (e.g., collaboration on open source projects)
- Training and further education within the scope of iSAQB Advanced Level training courses with a minimum of 70 credit points from at least three different areas of competence
- Successful completion of the CPSA-A certification exam

Essentials

What does the module “FLEX” convey?

The module presents FLEX to the participants how to build very flexible systems with modern architecture approaches. We start by looking at which quality goals actually motivate a flexible, distributed architecture at all. We also review different options for modularization and the architectural concepts behind them. The participants learn how communication structures between teams and their organisation impacts their architecture decisions and how to organize macro architecture decision making.

At the end of the module, the participants know the basics of architecture styles like microservices and self-contained systems and are able to design systems based on these concepts. They will be able to suggest a meaningful business decomposition that includes insights stemming from the concepts of Continuous Delivery and sustainable service operations.

Curriculum Structure and Recommended Durations

Content	Recommended duration (minutes)	Exercise time (minutes)
1. Motivation	30	
2. Modularisation	180	75
3. Integration	210	90
4. Installation and Roll Out	120	30
5. Operations, Monitoring, and Failure Analysis	150	45
6. Outlook	90	
Total	780 (13h)	240 (4h)

In this context, "duration" refers to teaching time without exercises. The time breakdown between theory and exercises is only a recommendation. The design of the examples and exercises aren’t specified in this curriculum

Duration, Teaching Method and Further Details

The times stated below are recommendations. The duration of a training course on the FLEX module should be at least 3 days, but may be longer. Providers may differ in terms of duration, teaching method, type and structure of the exercises, and the detailed course structure. In particular, the curriculum provides no specifications on the nature of the examples and exercises.

Licensed training courses for the FLEX module contribute the following credit points towards admission to the final Advanced Level certification exam:

Methodical Competence:	10 Points
Technical Competence:	20 Points
Communicative Competence:	00 Points

Prerequisites

Participants **should** have the following knowledge and / or experience:

- Fundamentals of the description of architectures using various views, comprehen- sive concepts, design decisions, boundary conditions etc., as taught in the CPSA-F (Foundation Level).
- Experience with implementation and architecture in agile projects.
- Experiences from the development and architecture of classical systems with the typical challenges.

Useful for understanding some concepts are also:

- Distributed systems
  - Problems and challenges in the implementation of distributed systems
  - Typical distributed algorithms
  - Internet protocols
- Knowledge about modularisations
  - Functional modularisation
  - Technical implementations like packages or libraries
- Classical operation and deployment processes

Structure of the Curriculum

The individual sections of the curriculum are described according to the following structure:

- **Terms/principles**: Essential core terms of this topic.
- **Teaching/practice time**: Defines the minimum amount of teaching and practice time that must be spent on this topic or its practice in an accredited training course.
- **Learning goals**: Describes the content to be conveyed including its core terms and principles.

This section therefore also outlines the skills to be acquired in corresponding training courses.

Supplementary Information, Terms, Translations

To the extent necessary for understanding the curriculum, we have added definitions of technical terms to the iSAQB glossary and complemented them by references to (translated) literature.

1. Motivation

Duration: 120 min	Practice time: 0 min

1.1. Terms and Principles

Availability, resilience, time-to-market, flexibility, predictability, reproducibility, homogenization of stages, internet/web scale, distributed systems, parallelizing feature development, evolution of architecture (build for replacement), heterogenity, automation.

1.2. Learning Goals

LG 1-1: Quality goals of flexible architectures

Architectures can be optimised to meet different quality goals.

In this module participants learn how to develop architectures that allow fast deployment and fast feedback from using the system. Such architectures support developers in achieving functional suitability efficiently and improve usability. They also foster modularity and can serve to achieve high reliability und replace individual components in isolation, which serves flexibility.

LG 1-2: Interaction of architecture types and organisation

They have understood the drivers for the architecture types taught in this curriculum module - the desire to bring feature to production fast and reduce the Time-to-Market. They understood the implications of these drivers for the architectures, and the interaction of the architectures with the organisation, processes, and technologies.

LG 1-3: Communicating and adapting compromises of the presented architecture types

They have understood the trade-offs of the architecture types presented (at least Microservices, Self-Contained Systems and Deployment Monoliths) and are able to communicate them as well as apply them in the context of concrete projects / system developments in order to make appropriate architectural decisions.

LG 1-4: Understanding the constraints of distributed systems

Participants understand the constraints of distributed systems, such as the following:

- The architecture has crucial influence on the ability to quickly bring new features into production.
- Dependencies between components of different development teams influence the duration until software can be put into production because they increase the communication costs and delays are propagated.
- Automation of activities (such as test and deployment processes) increases the reproducibility, predictability, and quality of results of these processes. This can lead to an improvement of the overall development process.
- Unification of the different environments (e.g., development, test, QA, production) reduces the risk of lately detected and (in other environments) non-reproducible errors due to different configurations.
- Unification and automation are essential aspects of continuous delivery.
- Continuous integration is a prerequisite for continuous delivery.
- A suitable architecture is the prerequisite for the parallelization of the development as well as the

independent commissioning of independent modules. This can, but do not have to be "services".

- Some change scenarios are easier to implement in monolithic architectures. Other change scenarios are easier to implement in distributed service architectures. Both approaches can be combined.
- There are different types of isolation with different advantages. A failure, for example, can be limited to a single component or changes can be limited to a single component.
- Certain types of isolation are much easier to implement between processes with remote communication.
- Remote communication, however, has disadvantages - for example many new sources of errors.

LG 1-5: Contextualizing topics and buzzwords

- Conway's law
- Partitionability as a quality feature
- Round trip times with the IT supply chain as a competitive factor
- Building a continuous delivery pipeline
- The different test phases of a continuous delivery pipeline

1.3. References

[Humble, et al. 2010], [Wolff 2016], [Humble, et al. 2014]

2. Modularization

Duration: 120 min	Practice time: 30 min

2.1. Terms and Principles

- Motivation for decompositioning into smaller systems
- Different kinds of modularisation, coupling
- System limits as a way of isolation
- Hierarchical structure
- Application, Self-Contained System, Microservice
- Domain-Driven Design Concepts and "Strategic Design", Bounded Contexts

2.2. Learning Goals

LG 2-1: Decomposing into building blocks

What shall participants be capable of?

- Participants can design a decomposition into individual blocks for a given problem.
- The participants should consider the organisation’s communication structure when setting the module boundaries (Conway’s law).
- The participants can draw up a plan to divide a deployment monolith into small services.

What should participants understand?

- Participants understand that each type of building blocks requires a handy label, as well as a description,
  - what makes up this kind of building block
  - how such a building block is integrated at runtime
  - how such a building block is built (in the sense of the build system)
  - how such a building block is deployed
  - how such a building block is tested
  - how such a building block is scaled
- Different levels of specifications can be useful for the development of a module. Some specifications should better be superior valid for the integration with other building blocks of this type, in general. The overarching decisions that affect all systems can form a macro architecture, including, for example, communications protocols or operating standards. Micro architecture can be the architecture of a single system. It is largely independent of other systems. Excessive limitations at the macro architecture level will lead to an overall architecture that can be applied to fewer problems

What should participants know?

- The participants should know Conway’s law.

LG 2-2: Modularisation concepts

What shall participants be capable of?

- The participants should be able to evaluate and select technical modularisation concepts in a project-specific manner.
- The participants should be able to illustrate and analyse the relationships between modules as well as between modules and subdomains (context mapping).
- Participants can develop a concept to build a system of services.

What should participants understand?

- Modularisation helps to achieve goals such as parallelization of development, independent deployment / interchangeability at runtime, rebuild / reuse of modules and easier understanding of the overall system.
- Therefore, techniques like continuous delivery and the automation of test and deployment are important influences on the modularisation.
- Modularisation means the decomposition of a system into smaller parts. Re-integrating these parts after the decomposition causes organisational and technical efforts. These efforts have to be exceeded by the advantages achieved by the modularisation.
- Depending on the chosen modularisation technology, there is coupling on different levels:
  - Sourcecode (modularisation with files, classes, packages, namespaces etc.)
  - Built target (modularisation with JARs, libraries, DLLs, etc.)
  - Runtime environment (operating system, virtual machine or container)
  - Network protocol (distribution to different processes)
- A coupling at the source code level requires very close cooperation as well as common SCM. A coupling at the level of the built target means that the building blocks must usually be deployed together. Only a distribution to different applications / processes is feasible with regard to independent deployment.
- Participants understand that a complete isolation between building blocks can only be ensured by a separation in all phases (development, deployment and runtime). If this is not the case, undesirable dependencies cannot be excluded. At the same time, the participants also understand that it can be useful to forego complete isolation for reasons such as efficient resource usage or complexity reduction.
- Participants understand that, when distributing to different processes, some dependencies no longer exist in the implementation, but rather arise at runtime. This increases the requirements for monitoring these interfaces.
- Microservices are independent deployment units and therefore independent processes that expose their functions through lightweight protocols, but may also have a UI. Different technology decisions can be made for the implementation of each individual microservice.
- A Self-Contained System (SCS) is a functionally independent system. It usually includes UI and persistence. It may consist of several Microservices. Usually a SCS covers a functional bounded context.

What should participants know?

- The participants should know various technical modularisation options: e.g., source code files, libraries, frameworks, plugins, applications, processes, Microservices, SCS.
- The participants should know various technical modularisation options: e.g., source files, libraries, frameworks, plugins, applications, processes, microservices, self-contained systems

- The participants should know "The Twelve-Factor App".

LG 2-3: Modularisation strategies

What shall participants be capable of?

- Participants can evaluate the consequences of different modularisation strategies and compare the efforts of the modularisation with the expected benefits.
- Participants can assess the impact of the modularisation strategy on the autonomy of building blocks at development time and at run time.
- The participants can choose a suitable modularisation as well as a suitable granularity of the modularisation - depending on the organisation and the quality goals.

What should participants understand?

- Participants understand that an integration strategy decides whether a dependency
  - emerges at runtime
  - emerges during development time, or
  - emerges at the deployment.
- The module division can be done along functional or technical boundaries. In most cases, a functional division is recommended, because in this case functional requirements can be assigned more clearly to a concrete module, and therefore it is not necessary to adapt several modules for the implementation of a single functional requirement. Thereby, each module can have its own domain model in the sense of a bounded context and thus different views on a business object with its own data.
- Participants understand that in order to achieve higher autonomy of the development teams, it is better to divide a component along functional boundaries rather than along technical boundaries.
- Transactional consistency through process boundaries can only be achieved via additional mechanisms. So, if a system is divided into several processes, the module boundary often also represents the limit for transactional consistency. Therefore, a DDD aggregate must be managed in one module.
- Participants understand which modularisation concepts can be used not only for transactional but also for batch- and data-flow-oriented systems.

What should participants know?

- The participants should know the following terms from the domain-driven design: Aggregate Root, Context Mapping, Bounded Contexts and relationships between them (e.g., Anti-Corruption Layer).

.

2.3. References

[Eric Evans 2003], [Lewis, Fowler, et al. 2013], [12 Factor App 2012], [Wolff 2018], [Sam Newman 2021], [Parnas 1972]

3. Integration

Duration: 120 min	Practice time: 30 min

3.1. Terms and Principles

Frontend integration, legacy systems, authentication, authorisation, (loose) coupling, scalability, messaging patterns, domain events, decentralised data management.

3.2. Learning Goals

LG 3-1: Strategic Design

1. Participants should have a basic understanding of the implementation of integrations using Strategic Design from Domain Driven Design and know fundamental patterns.
2. For the integration of legacy systems, it has to be defined how to handle old data models. For this purpose, the approach of strategic design can be used with essential patterns such as anti-corruption layers.

LG 3-2: Choosing a technical integration strategy

1. The participants should know the benefits and drawbacks of different integration mechanisms. These include front-end integration with Mash Ups, Middle Tier integration, and integration through databases or database replication.
2. The participants can propose a suitable integration depending on the quality goals and the knowledge of the team.
3. The participants should choose an integration strategy that best suits the particular problem. This can be, for example, front-end integration, integration via RPC mechanisms, message-oriented middleware, REST, or replication of data.
4. Participants should understand the implications and limitations that arise from the integration of systems across different technologies and integration patterns, relating to, for example, security, response time, or latency.
5. RPC means mechanisms for synchronously calling functionality in another process over computer boundaries. This results in coupling in many respects (time, data format, API). This coupling has a negative effect on the availability and response times of the system. REST makes guidelines that can reduce this coupling (Hypermedia, standardised API). However, basically, the temporal coupling remains.
6. With integrating through messaging, systems communicate through the asynchronous exchange of messages. The systems are thus decoupled in time. Technically, this is achieved by means of indirection via a middleware. Messages can optionally be persisted, filtered, transformed, etc. There are different messaging patterns like Request / Reply, Publish / Subscribe or Broadcast.
7. An integration via data enables high autonomy, nevertheless it is bought by the necessity for redundant data storage and the necessary synchronisation. It must not be assumed that other systems use the same schemes, because this prevents an independent development of the schemata. Therefore, an adequate transformation has to be provided for the integration.
8. The participants should be able to name different approaches for front-end integration as well as their benefits and drawbacks and applicability according to context criteria.
9. The participants should know technologies for the integration of services: REST, RPC, message-

oriented middleware.

10. The participants should know database replication mechanisms using ETL tools or other approaches
11. The participants should know messaging Patterns (Request / Reply, Publish / Subscribe, etc.)
12. The participants should know messaging systems (RabbitMQ, Kafka etc.), protocols (AMQP, MQTT, STOMP etc.) and APIs (JMS).

LG 3-3: Explain macro architecture und formulate criteria

1. Based on these approaches, participants should be able to design a macro architecture that covers at least communication and security.
2. The participants can name clear criteria developers can use independently to determine if an (architecture) decision is a decision about micro architecture or macro architecture, if they can make that decision on their own or if they need to involve external stakeholders in the decision process.

LG 3-4: Domain events instead of RPC for integrating bounded contexts

1. In an event-driven architecture (EDA), RPC is avoided or reduced by publishing domain events. Domain events describe state changes. Interested systems can process these messages (Publish / Subscribe). This procedure affects how the state is stored. While, in an RPC-based integration the server has to store the data, with EDA this is the responsibility of the subscriber. Thus, replicas arise (decentralised data storage). Thereby, the subscribers act as a cache for the publisher. Additional subscribers can be added without affecting the publisher (except by polling). Monitoring of the event flows is important.
2. Domain events can be published via messaging. The publisher pushes the messages into a messaging system. Alternatively, the messages can be polled from the publisher (e.g., Atom / RSS). When using a messaging system, subscribers can receive the messages by push or pull. This has implications for dealing with backpressure.
3. The participants should know challenges of the usage of shared data.

LG 3-5: Security

1. Participants should be able to design a suitable approach for implementing security (authorisation / authentication) in a distributed system.
2. The participants should know typical distributed security mechanisms such as OAuth or Kerberos

3.3. References

[Eric Evans 2003], [Parecki et al. 2006], [Hohpe, Woolf 2003]

4. Installation and Roll Out

Duration: 90 min	Practice time: 30 min

4.1. Terms and Principles

Modern operations, DevOps, Infrastructure as Code, Configuration Management.

4.2. Learning Goals

LG 4-1: Aspects of rollout and continuous delivery

What shall participants be capable of?

- The participants should be able to roughly sketch a concept and understand how to automatically deploy a system as simple as possible. They should be able to weigh between the different technological approaches.
- Participants should be able to design and evaluate deployment automation. For example, they have to be able to assess the quality and testing of these approaches. They have to be able to select an appropriate approach for a project scenario.
- The participants should be able to sketch a team structure based on the DevOps organisation model.

What should participants understand?

- The foundation for automating deployment is virtualization or the cloud with Infrastructure as a Service (IaaS). A lightweight alternative are Linux containers as implemented by Docker.
- Deployment of a large number of servers and services is virtually impossible without a deployment automation.
- Modern deployment tools allow to automatically install software on computers. In addition to the application itself, the complete infrastructure can also be created automatically. In this case installations are idempotent, which means that they always lead to the same result regardless of the initial state of the system.
- Immutable servers are never changed. If a new version of the software has to be taken into operation, the server is rebuilt from scratch. This can be simpler and more reliable than to rely on idempotent tools.
- PaaS (Platform as a Service) provides a complete platform where applications can be deployed in. Since in this case the infrastructure is not built up by oneself, the approach is simpler, but also less flexible.
- Concepts such as tests or code reviews are also indispensable for deployment automation. Infrastructure becomes code that must meet the same requirements as productive code (Infrastructure as Code).
- To support the deployment, the result of a build process can be packages for an operating system or even images for virtual machines.
- The environments of a developer should ideally match the production environments. With modern tools, it is possible to create and maintain such an environment at the push of a button.
- The complexity of the deployment becomes a further quality feature of the system and affects architecture tools.
- With DevOps, development and operations are integrated in a single team. Consequently, operations

must be considered already during development. This affects provisioning but also continuous delivery (see chapter "Continuous Delivery").

What should participants know?

- Basic concept of modern infrastructure such as IaaS, PaaS and virtualization concepts of deployment tools like Chef, Puppet, Ansible or Salt Organisation forms for DevOps
- Concepts of deployments with package managers or Linux containers Various PaaS platforms and their concepts

4.3. References

[Vossen, Haselmann, Hoeren 2012], [Wolff, Müller, Löwenstein 2013], [Humble, et al. 2010], [Wolff 2016]

5. Operations, Monitoring, and Failure Analysis

Duration: 90 min	Practice time: 30 min

5.1. Terms and Principles

Monitoring, Operations, Logging, Tracing, Metrics, Alerting.

5.2. Learning Goals

LG 5-1: Fault analysis in distributed systems

What shall participants be capable of?

- The participants should be able to roughly sketch and understand a concept that allows to monitor a system, that means assess the status, avoid errors and deviations from the regular operation as far as possible or at least detect and handle them as early as possible.

What should participants understand?

- The appropriate selection of data is essential for reliable and meaningful monitoring and logging.
- In order to achieve systems that are ready to operate, in particular those composed of many individual subsystems, the support of operations has to be an integral part of the architecture concepts.
- In order to achieve the highest possible transparency, a great deal of data has to be collected, but also be pre-aggregated and made evaluable.
- The participants should understand which information can be obtained from log data and which may (better) be obtained by instrumentation of the code with metric probes.

LG 5-2: Difference between metrics, logs, and traces

What should participants understand?

- Logging and monitoring can include both functional and technical data.
- Logs are events, a metric is a state at a specific time

What should participants know?

- Tools for centralised log data management
- Tools for centralised metrics processing
- Difference between business, application and system metrics
- The meaning of important, system-independent application and system metrics

LG 5-3: Different models of operations

What shall participants be capable of?

- According to context - such as organisation und skill set - decide on centralised or decentralised operation.
- In the concept, they may focus on logging, monitoring and the data that is required for that purpose, depending on the specific project scenario
- Participants should be able to meet architecture requirements in a way that supports the usage of appropriate tools, while reasonably dealing with system resources.

What should participants understand?

- The participants should understand how a typical centralised log data administration is built and what impact it has on the architecture.
- The participants should understand how a typical centralised metrics pipeline is built (capture, collect & sample, persist, query, visualise) and the impact it has on the architecture (performance overhead, memory consumption, …).
- The participants should understand the various options of logging, monitoring and an Operations DB (see [Nygard 2018]), which to use wherefore and how to meaningfully combine these tools.

5.3. References

[Wolff 2016], [Nygard 2018]

6. Case Study

Duration: 90 min	Practice time: 60 min

During a curriculum-compliant training a case study has to be used to explain and practice the curriculum's concepts.

6.1. Terms and Principles

The Case Study does not introduce new terms, principles, or concepts

6.2. Learning Goals

The Case Study shall not introduce additional learning goals, but intensify the topics via practically relevant exercises.

6.3. References

None. Training providers are responsible for selecting and creating examples and exercises.

7. Outlook

Duration: 120 min	Practice time: 0 min

The lookout presents advanced topics that participants may dive into. So, they gain a deeper understanding of the challenges of implementing flexible systems. In addition, they learn other factors influencing the choice of technologies.

7.1. Terms and Principles

- Consistency models: ACID, BASE, Partitioning, CAP
- Resilience: Resilient Software Design, Stability, Availability, Graceful Degradation, Circuit Breaker, Bulkhead

7.2. Learning Goals

LG 7-1: Consistency Models

What shall participants be capable of?

- The participants should know various consistency models. They should basically know the trade-offs of the different consistency models.
- Depending on the requirements and general conditions, they should be able to decide whether traditional stability approaches are adequate or resilient software design is required.

What should participants understand?

- The need for ACID transactions is much lower than is often assumed.
- Different scaling, distribution, and availability requirements require different consistency models.
- The CAP theorem describes a spectrum in which, depending on the given requirements, a suitable consistency model can be selected very fine-grained.
- BASE transactions guarantee consistency, however, they may not be atomic and isolated as ACID transactions, which may lead to temporary inconsistencies becoming visible.

What should participants know?

- Characteristics of and differences between ACID and BASE transactions
- Some product examples from different categories (e. g., NoSQL, configuration tools, service discovery)
- CAP for describing and explaining consistency models

LG 7-2: Resilience Patterns

What shall participants be capable of?

- They should be able to decide, depending on requirements and constraints, if traditional resilience approaches are sufficient or if resilient software design is required.

What shall participants understand?

- Traditional stability approaches (error avoidance strategies) on infrastructure levels are generally no longer sufficient for today's distributed, highly networked system landscapes.

- There is no Silver Bullet for Resilient Software Design, i. e., the relevant measures and applied patterns and principles depend on the requirements, the general conditions and the persons involved.

What shall participants know?

- The formula for availability and the different approaches to maximise availability (maximising MTTF, minimising MTTR)
- Isolation and latency monitoring as useful starting principles of Resilient Software Design
- Basic Resilience patterns such as bulkhead, circuit breaker, redundancy, failover

7.3. References

[Tanenbaum, van Steen 2006], [Lamport 1998], [Brewer 2000], [Takada 2013], [Nygard 2018], [Hanmer 2007], [Hamilton 2007]

References

This section contains references that are cited in the curriculum.

B

- [Brewer 2000] Eric Brewer, Towards Robust Distributed Systems, PODC Keynote, July-19-2000

E

- [Eric Evans 2003] Eric Evans: Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison- Wesley Professional, 2003

H

- [Hamilton 2007] James Hamilton, On Designing and Deploying Internet-Scale Services, 21st LISA Conference 2007
- [Hanmer 2007] Robert S. Hanmer, Patterns for Fault Tolerant Software, Wiley, 2007
- [Hohpe, Woolf 2003] Gregor Hohpe, Bobby Woolf: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley, 2003, ISBN 978-0-32120-068-6
- [Humble, et al. 2010] Jez Humble, David Farley: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley, 2010, ISBN 978-0-32160-191-9
- [Humble, et al. 2014] Jez Humble, Barry O'Reilly, Joanne Molesky: Lean Enterprise: Adopting Continuous Delivery, DevOps, and Lean Startup at Scale, O'Reilly 2014, ISBN 978-1-44936-842-5

L

- [Lewis, Fowler, et al. 2013] James Lewis, Martin Fowler, et al.: Microservices - http://martinfowler.com/articles/microservices.html, 2013
- [Lamport 1998] Leslie Lamport, The Part-Time Parliament, ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169

N

- [Sam Newman 2021] Sam Newman: Building Microservices: Designing Fine-Grained Systems, O'Reilly Media, 2nd ed. edition, 2021, ISBN 978-1-49203-402-5
- [Nygard 2018] Michael T. Nygard, Release It!, 2. Auflage, Pragmatic Bookshelf, 2018

O

- [Parecki et al. 2006] Aaron Parecki et al., Explaining OAuth 2.0 http://oauth.net/

P

- [Parnas 1972] David Parnas, On the Criteria To Be Used in Decomposing Systems into Modules. Communications of the ACM 15(12):1053—1058, 1972.

T

- [Takada 2013] Mikito Takada, Distributed Systems for Fun and Profit, http://book.mixu.net/distsys/ (Guter Einstieg und Überblick)
- [Tanenbaum, van Steen 2006] Andrew Tanenbaum, Marten van Steen, Distributed Systems – Principles and Paradigms, Prentice Hall, 2nd Edition, 2006

V

- [Vossen, Haselmann, Hoeren 2012] Gottfried Vossen, Till Haselmann, Thomas Hoeren: Cloud-Computing für Unternehmen: Technische, wirtschaftliche, rechtliche und organisatorische Aspekte, dpunkt, 2012, ISBN 978-3- 89864-808-0

W

- [Wolff 2016] Eberhard Wolff: Continuous Delivery: Der pragmatische Einstieg, 2. Auflage, dpunkt, 2016, ISBN 978-3-86490-371-7
- [Wolff 2018] Eberhard Wolff: Microservices - Grundlagen flexibler Software Architekturen, 2. Auflage, dpunkt, 2018, ISBN 978-3-86490-555-1
- [Wolff, Müller, Löwenstein 2013] Eberhard Wolff, Stephan Müller, Bernhard Löwenstein: PaaS - Die wichtigsten Java Clouds auf einen Blick, entwickler.press, 2013

1

- [12 Factor App 2012] 12 Factor App, Guidelines for building apps on Heroku http://12factor.net/