AARHUS UNIVERSITET

Microservices and DevOps

Scalable Microservices

The Microservice Architectural Style

Henrik Bærbak Christensen

Designing MS Architectures?

- The challenge:

- We have the MS definition
  - All those benefits (and liabilities)

- What's next then???

- How to 'do it'?

Our Take: A set of central concepts in the architectural design process.
Overlap with SAiP fagpakke ☺

Architectural Style

- Similar to *architectural pattern* and out of fashion, but I do like the different perspective and the conceptual framework it brings along:
- **Architectural Style:**
  - **Set of element types (components)**
  - **Set of interaction mechanism (connectors)**
  - **Topology of components**
  - **Semantic constraints**

- Exercise:
  - Client-server ?

Relation To Def

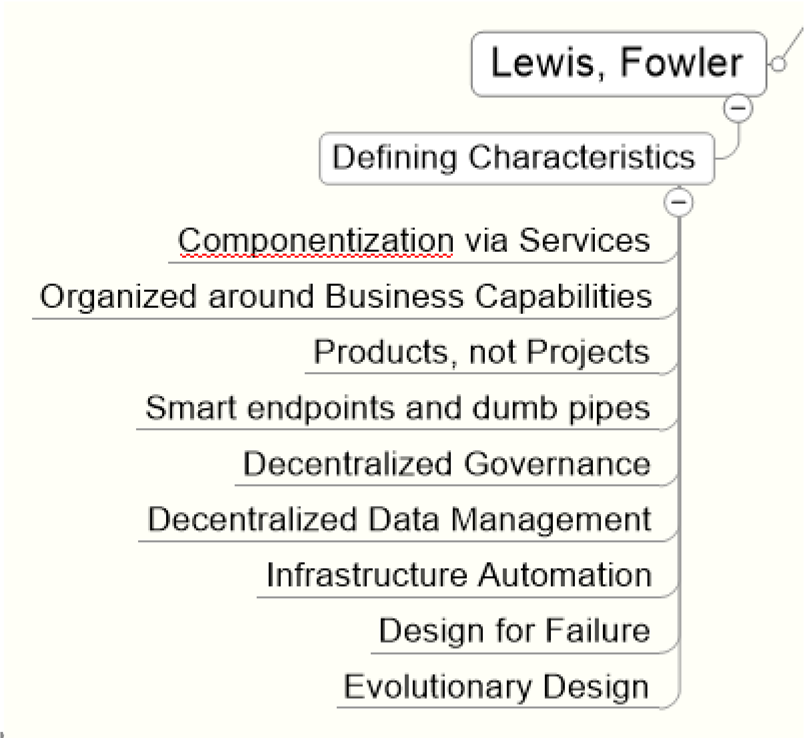
- Actual the definition is quite precise about a subset of the defining characteristics of the

- **The Microservice Architectural Style**
  - **Components, Connectors, Topology, Semantic constraints**

AARHUS UNIVERSITET

Relation to Def

- **The Microservice Architectural Style**
  - **Components, Connectors, Topology, Semantic constraints**



Lewis, Fowler
Defining Characteristics
Componentization via Services
Organized around Business Capabilities
Products, not Projects
Smart endpoints and dumb pipes
Decentralized Governance
Decentralized Data Management
Infrastructure Automation
Design for Failure
Evolutionary Design

AARHUS UNIVERSITET

Aspects

AARHUS UNIVERSITET

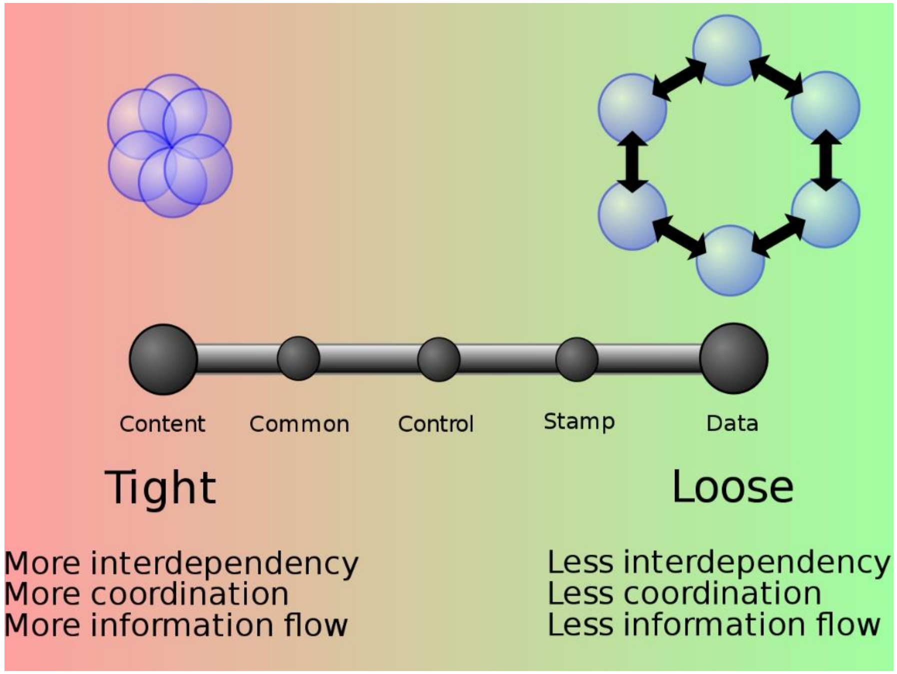
Coupling and Cohesion

So old, and so difficult to get right

‘it is all about role boundaries’

Larry Constantine

- Late 1960'ies formulation
  - Coupling
- The antidote:
  - Information Hiding
    - David Parnas
  - Encapsulation
    - An *inside* and *outside*



- Shaw
  - *It is not the functionality per se, it is the packing*

Packaging

- To rephrase the previous *software reuse* historical treatment, much of every new tech-hype has been about a *new scheme for packing software into a deploy unit*

– Modules (modula 2) :	Language packaging
– Objects (simula/smalltalk) :	Language packaging
– CBS:	Static deployments
– SOA:	Dynamic deployments
– Microservices:	Dynamic deployments at scale

- Note that packing does not really help on coupling

AARHUS UNIVERSITET

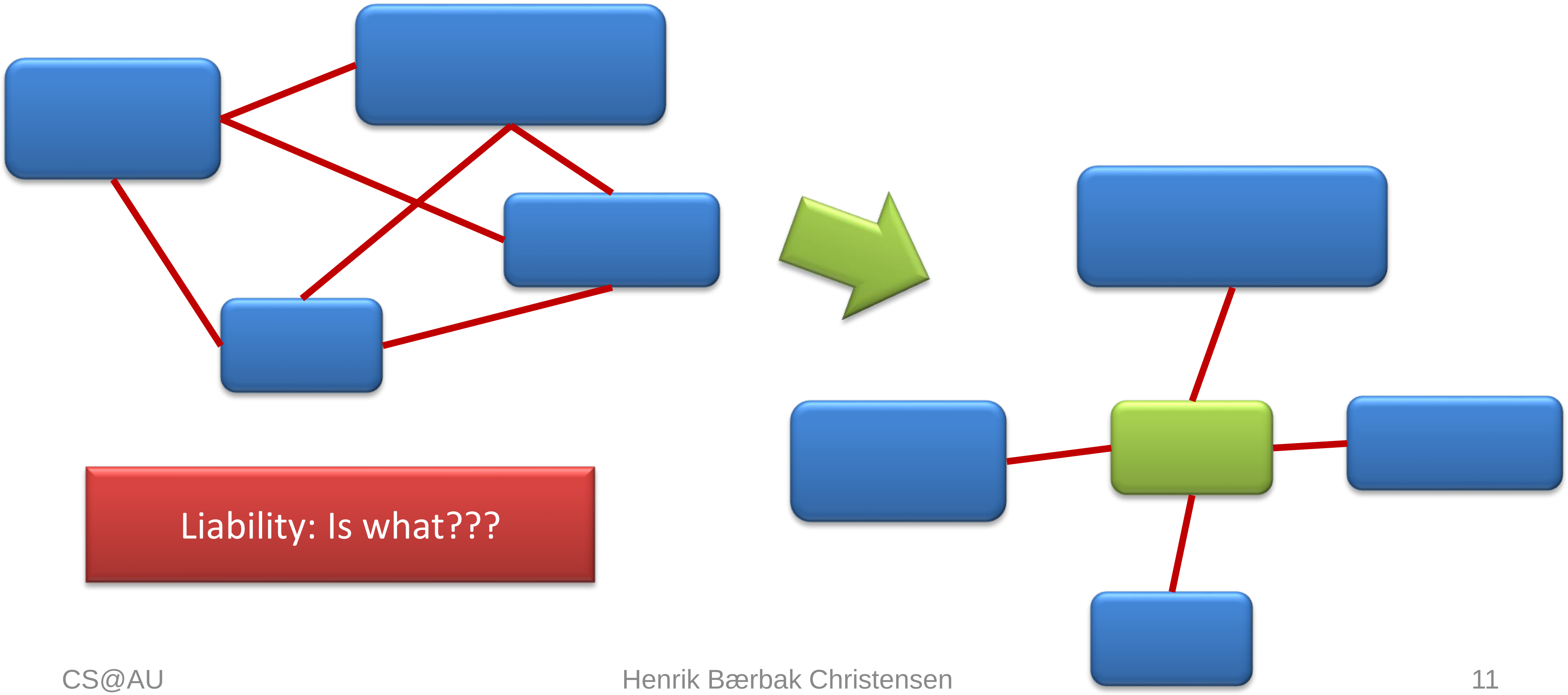
An OO Experience

- My early OO experience usually ended in one of two places
  - A mesh of highly interconnected objects ☹
  - A god class that starved any other object from behavior ☹
- The **Mediator** Pattern
  - Intent: *Object that encapuslates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and lets you vary their interaction independently.*
  - Head On book: *Centralize complex control and data flow between related objects*

AARHUS UNIVERSITET

Visually

- Replace high interconnectivity with central Mediator

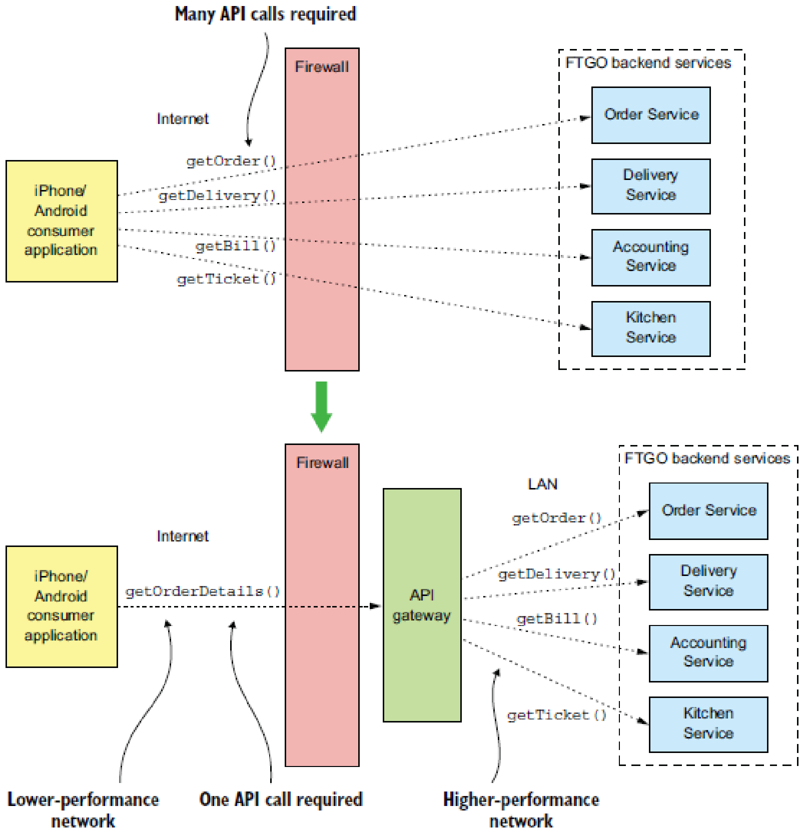


Liability: Is what???

AARHUS UNIVERSITET

API Gateway pattern

- API Gateway patterns is essentially the ‘out-of-process’ equivalent to the Mediator.
- SkyCave: Who plays the Mediator role?
- But note:
  - Smart services, dumb connectors !



Richardson, p 261

AARHUS UNIVERSITET

Compositional Principles

The old GoF
principles of reusable design

Worth Mentioning

- Principle 1:
- *Program to an interface, not implementation*
  - Make a clear contract between supplier and consumer

- Principle 2:
- *Favor object composition, over class inheritance*
  - Make system as composition of services

Bounded Contexts

From Domain Driven Design (DDD) by Eric Evans…

Newman §3

AARHUS UNIVERSITET

Bounded Contexts

- *Domain = Set of bounded contexts.*
  - *Bounded context = model elements that are private* ***plus*** *model elements that are* ***shared externally*** *with other BCs.*
  - *Bounded context = specific responsibility enforced by explicit boundaries*

  - Request to BC through *shared models* on explicit boundary

  - Analogy with living cells – membranes define what can pass

Eric Evans: *Domain-Driven Design* Book

MusicCorp Example

- BC: *Finance* and *Warehouse*
- Shared model: *Stock Item*
  - *However:* full stock object not relevant for finance
    - *Shelf location has no meaning to finance, only value and count*
  - Stock has both an internal (**hidden**) and external (**shared**) representation

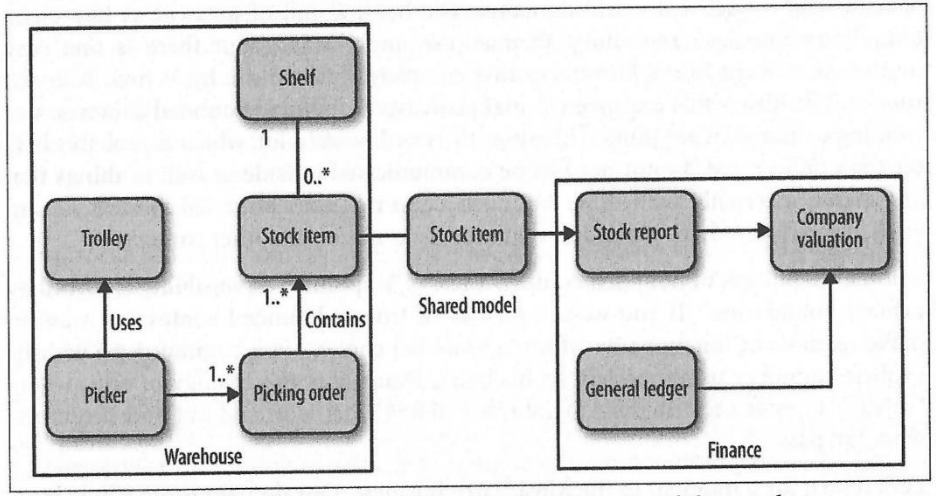


Figure 3-1. A shared model between the finance department and the warehouse

AARHUS UNIVERSITET

Definition?

- *Shared Model:* A subset of a model, containing only properties relevant for a given, external, bounded context [My take on a definition]**.**
- Example:
  - A ‘return’
    - Customer context: print return label, await refund
    - Warehouse context: package that will arrive, restocking
- Discussion
  - I guess it means there can be several ‘shared models’ for a concept
    - Stock Item for Warehouse, Finance, Customer, …

On a more Language Level

- One way of exchange *models* on BC boundary
  - HTTP GET that JSON object from the service
  - Broker ‘List<PODO> getListOfMessages()’ in ChatRoom object, where client has a ChatRoomClientProxy
  - Subscribe to ProtoBuf encoded binary object from RabbitMQ on topic using routing key: ‘*.measurements.aarhus’
- Models are PlainOldDataObjects/PODOs.
  - Optionally with *abstract references* ala foreign keys
    - For further exchange of ‘models’ that represent these sub PODOs

BC as Modules

- Bounded contexts form module (CC view: Component) boundaries
  - Module = classic static/compile time software unit, with explicit interface(s), *Facade pattern*

- Excellent candidates for later microservices!
  - Monolith first tactic. *Avoid premature decomposition*
  - Why
    - API refactorings are extremely slow in the remote case…

Compare OO Perspective

- OOA and OOD
  - The obvious old OO way:
    - StockItem Warehouse::getAllItems()

- Liabilities
  - Expose internal warehouse details
  - Hard binding through object reference(s)

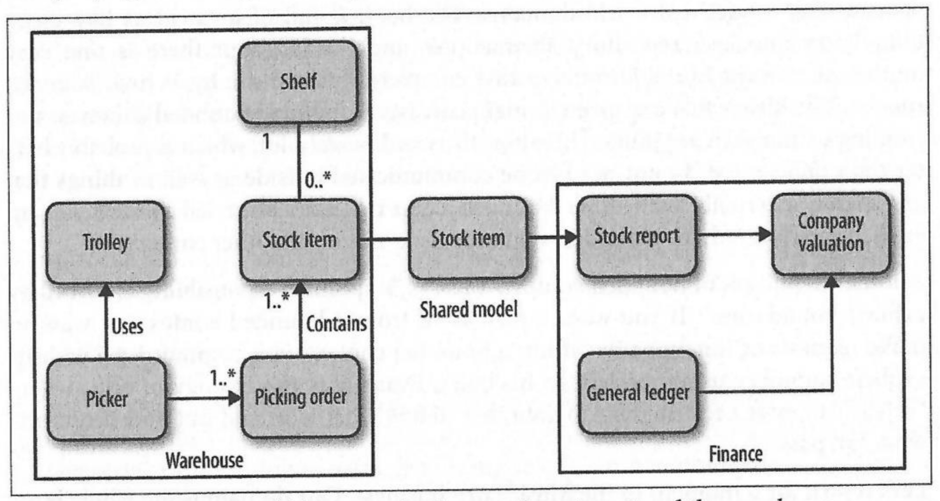
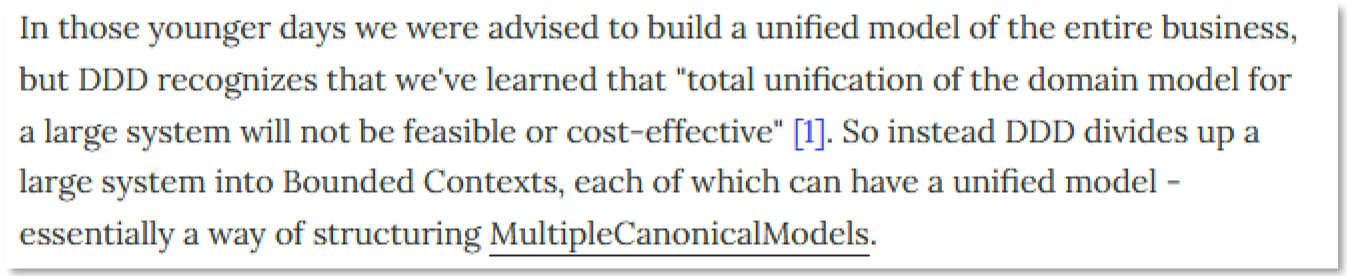


Figure 3-1. A shared model between the finance department and the warehouse

Compare OO Perspective

- Fowler 'BoundedContext' paper



In those younger days we were advised to build a unified model of the entire business, but DDD recognizes that we've learned that "total unification of the domain model for a large system will not be feasible or cost-effective" [1]. So instead DDD divides up a large system into Bounded Contexts, each of which can have a unified model - essentially a way of structuring MultipleCanonicalModels.

- WarStory: EPJ domain…
  - GEPJ
  - HL7 and CDA

Business Capabilities

- Fowler: *Organized around business capabilities*
- Bounded contexts provide services to each other
  - Warehouse: get stock list
  - Finance: set up payroll for new employee

- *Shared models* contain the information exchanged

- Hidden models (= internal/full models) candidates for the *data management layer*

AARHUS UNIVERSITET

Decomposition

- BCs can be a composition of sub BCs
  - Heard it before ☺?
- Consider org. Boundaries
  - If different teams/groups it may make more sense to go for decomposition
  - Conway's law…
- Consider testing
  - Nested appr. means less test doubles…

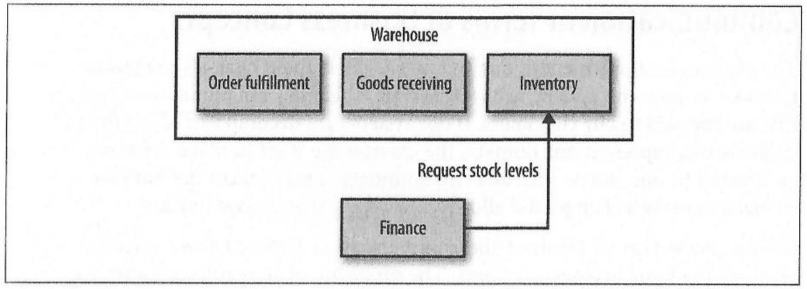


Figure 3-2. Microservices representing nested bounded contexts hidden inside the warehouse

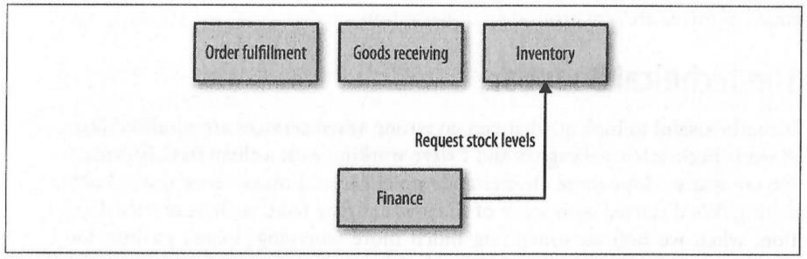


Figure 3-3. The bounded contexts inside the warehouse being popped up into their own top-level contexts

AARHUS UNIVERSITET

Domains are Stable

- One of the first rules of OO design I learned:
  - “Domains are more stable than user requests to UI and functionality”
  - OO modeling was highly ‘domain modeling’
    - OOA
- DDD and BC are founded in the same ‘truth’
  - Business changes more likely to be isolated to a single BC
- I do not completely agree…

AARHUS UNIVERSITET

Roles and Responsibilities

Loosening the Boundaries from the Domain

Much Bounded Context Hype

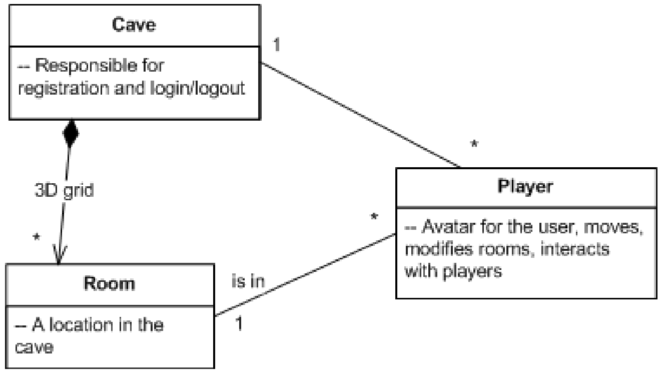
- Every Microservice book I look into has chapters on DDD and Bounded Contexts.
- Which is **rooted in the domain model!**
- I am old enough to have seen that before:

- Object Oriented Analysis and Design
  - *Talk to customers and write down the nouns. These are the classes. Then write down the verbs that are associated with these nouns. These are the methods.*

- Experience then showed that it was a lot of bullsh…

Domain is not All

- Domain classes are important but there are so much more to complex software systems than domain concepts.
- Example:
  - 2016 SkyCave consists of 97 classes. Only three of them are domain classes!
  - Rest is about:
    - Distribution
    - Persistence
    - Variability Management
    - Dependency injection
    - QA, like availability, performance, …

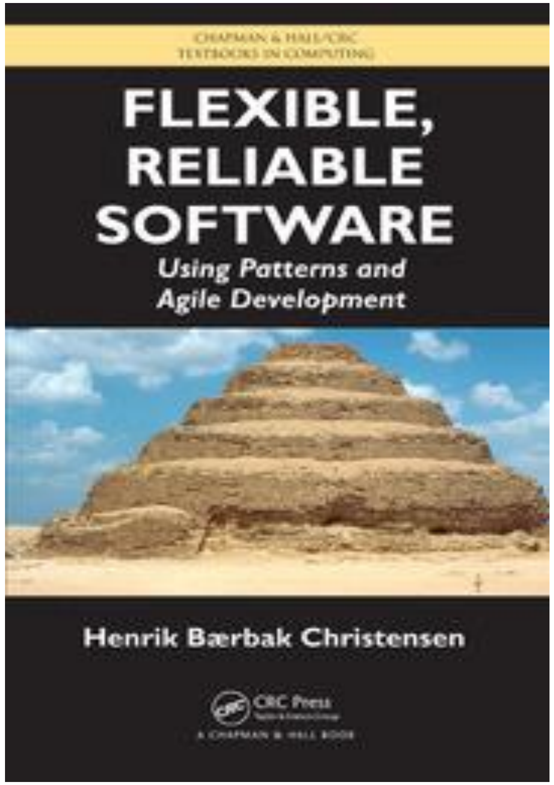

Cave
-- Responsible for registration and login/logout
1
*
3D grid
*
Player
-- Avatar for the user, moves, modifies rooms, interacts with players
*
Room
-- A location in the cave
is in
1

Domain is not All !

- Large classes of important software designs cannot be explained from a purely Domain focus!
- Example: *Design Patterns*
  - Strategy pattern:
    - Encapsulate algorithm and let it vary independently of consumer
- Absolutely no counterpart in any domain
  - And still a central software architectural tool in the architect’s toolbox

Stronger Conceptual Framework

- Roles, Responsibilities, Behavior, and Protocol
- Not
  - **The Order Concept**
- But
  - **The responsibilities associated with handling an Order**
- *Leads to more fine-grained abstractions*
- *More '-able' than noun interfaces*


CHAPMAN & HALL/CRC TEXTBOOKS IN COMPUTING
FLEXIBLE, RELIABLE SOFTWARE
Using Patterns and Agile Development
Henrik Bærbak Christensen
CRC Press
A CHAPMAN & HALL BOOK

The Central definitions

- The central concepts:
  - **Behaviour:** *What actually is being done*
    - ”Henrik sits Sunday morning and writes these slides”
  - **Responsibility:** *Being accountable for answering request*
    - ”Henrik is responsible for teaching responsibility-centric design”
  - **Role:** *A function/part performed in particular process*
    - ”Henrik is the course teacher”
  - **Protocol:** *Convention detailing the expected sequence of interactions by a set of roles*
    - ”Teacher: ‘Welcome’ => Students: stops talking and starts listening”

I see the Trend Already

- **FaaS: Function as a Service**
  - **A Responsibility that is set free from its prison in a bounded context, or a domain concept.**
- The Strategy Pattern in a new Packaging!

- Roles are *often invented independent of domain*, as just a cohesive collections of responsibilities
  - Kindergarten example: The *Flyer role…*
  - Car example: Is it an umbrella? Is it umbrella behavior?
  - (anyone read Larman? 'Pure Fabrication' principle)

AARHUS UNIVERSITET

So…

- Bounded Contexts and DDD are fine starting points
  - Just as domain understanding was it in OOA and OOD
- But do not let it restrict you

- Think
  - **Roles = responsible for cohesive part in particular process (or responsible for a *service* to the community!)**

- Microservices should perform *roles,* and use a *protocol* to exchange control and data (*shared models*)

AARHUS UNIVERSITET

DRY or Not DRY

Do not Repeat Yourself

Or maybe you should ☺ !

AARHUS UNIVERSITET

DRY

- Step five of the TDD rhythm
  - *Step 5: Refactor to remove duplication*

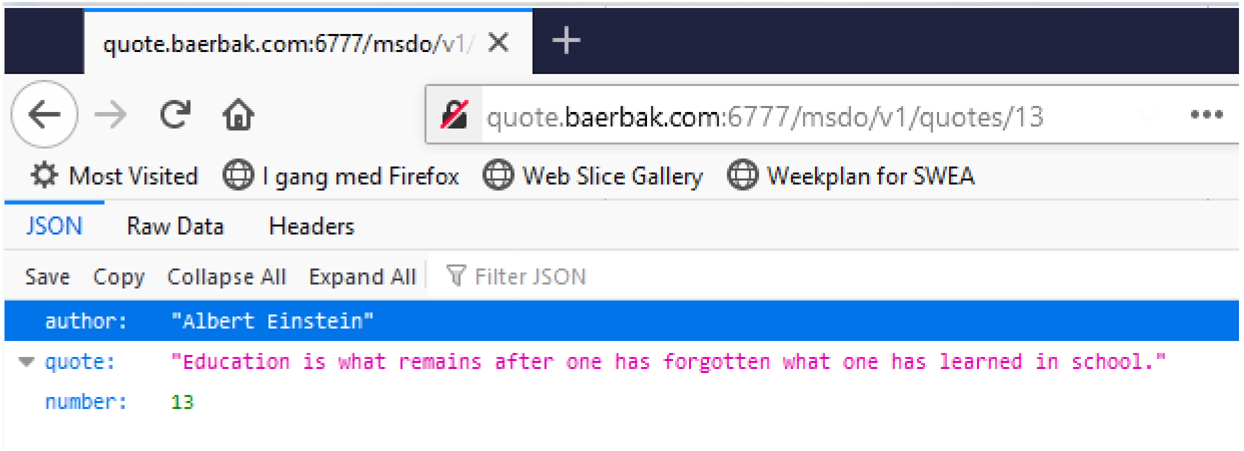
- Duplicated code is root of all evil
  - Dual maintenance, fixing a bug *once-and-for-all* is very difficult, etc. etc. etc.

- Then *what about the shared models? And connector implementations/client libraries*

AARHUS UNIVERSITET

SkyCave Example

- The battle-hardened *quote service*



```
author:   "Albert Einstein"
quote:    "Education is what remains after one has forgotten what one has learned in school."
number:   13
```

- Why not have a *shared library* with the quote PODO?
  - Class quotePodo { String author; String quote; int index; int statuscode; }

DRY downside

- You create *hard bindings* in the form of *shared library dependencies*
  - Both producer and consumer depend upon *quote-lib:1.7.3*
  - May create ‘lock-step’ versioning
    - Library update propagages across all services
  - AntiPattern: *The Distributed Monolith*
  - Begins to dictate the programming language ☺
    - Cannot import maven libs in python, to my knowledge

- Bottomline: *Relaxed about DRY across service boundaries*
  - Programming level: Do not put the ‘shared models’ in shared libs !

Connectors

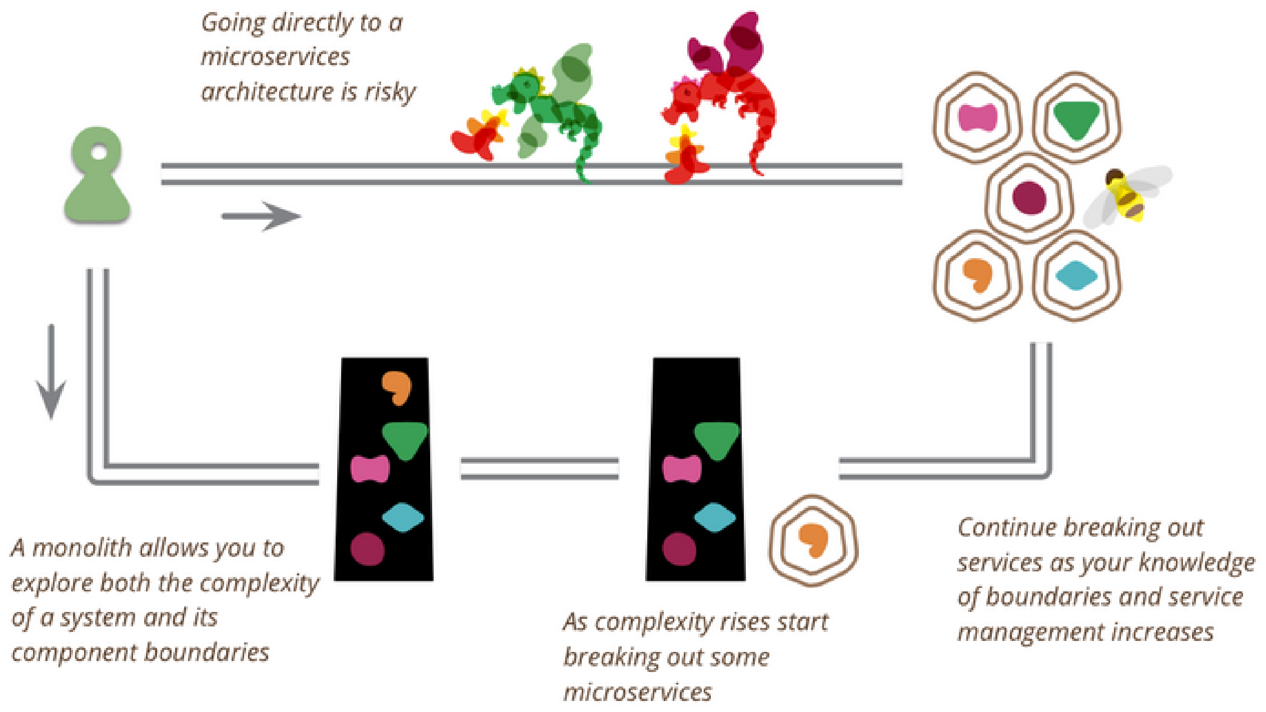
- What about library implementations of connectors?
  - Aka ‘client libraries’?
- Example: Our ‘RealQuoteService’ implementation – why did I not just hand it out to you?
  - (Would be an easy exercise, then ☺)
- Newman
  - Risk of server logic creeping into the connector logic
- Example of ‘it works’
  - Amazon Web Service SDK
    - Client lib for the AWS REST api
  - But developed by *other team + community,* not the AWS team !!!

AARHUS UNIVERSITET

Monolith First Pattern

Fowler

This pattern has led many of my colleagues to argue that **you shouldn't start a new project with microservices, even if you're sure your application will be big enough to make it worthwhile.** .


Going directly to a microservices architecture is risky
A monolith allows you to explore both the complexity of a system and its component boundaries
As complexity rises start breaking out some microservices
Continue breaking out services as your knowledge of boundaries and service management increases

My Opinion

- Right, I agree totally
  - Tool support so much better for in-process communication
  - YAGNI principle (you ain’t gonna need it)
- But, still
  - As service boundaries starts to materialize, I think it makes sense to start to apply the *architectural style* along the boundary, alas
    - Order order = coffeeshop.createOrder(Type.Latte, …);
    - *May become something like (REST’ifying the communication)*
    - OrderModel orderModel = new OrderModel(Type.Latte, …);
    - OrderResult result = coffeeshop.serviceCall(ORDER, orderModel);
    - Order order = result.getOrder();
  - That is, ‘Order’ without any deep references. Only a PODO!

AARHUS UNIVERSITET

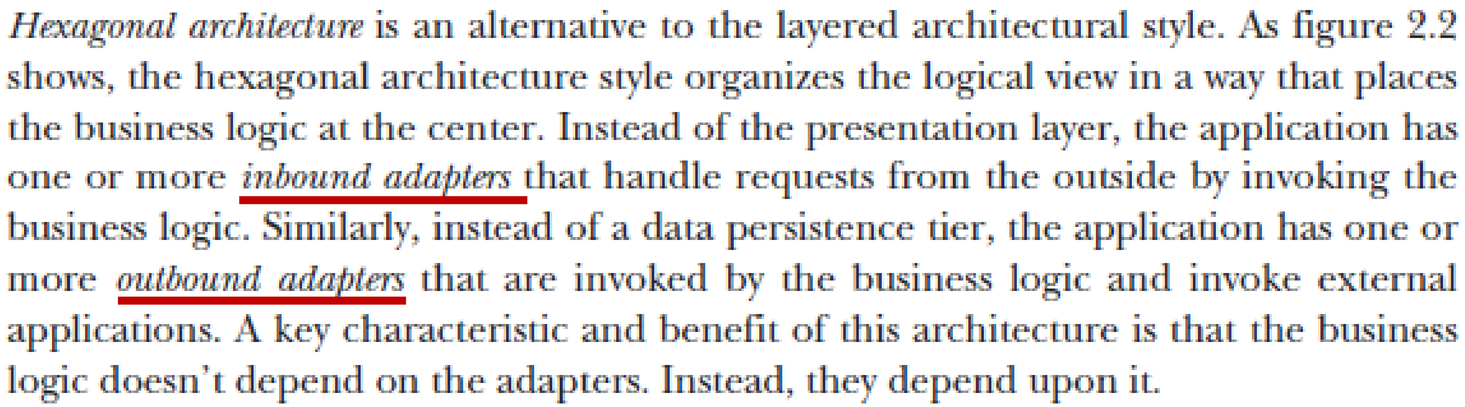
My Opinion

- Objection to this opinion
  - *You have no transparency, why not encapsulate the concrete connector style? This code smells RESTish, right?*
- It is a really good objection, but…
- *From an architectural standpoint it is highly important to be explicit when a connector is out-of-process*
  - Nygard: **Integration point**
    - Must be analyzed in terms of availability, performance, security, …
- I have to code time-outs, circuit breakers, chunky interface, …, for these connectors! *No transparency anyway!*

AARHUS UNIVERSITET

Outlook

Hexagonal Architectures



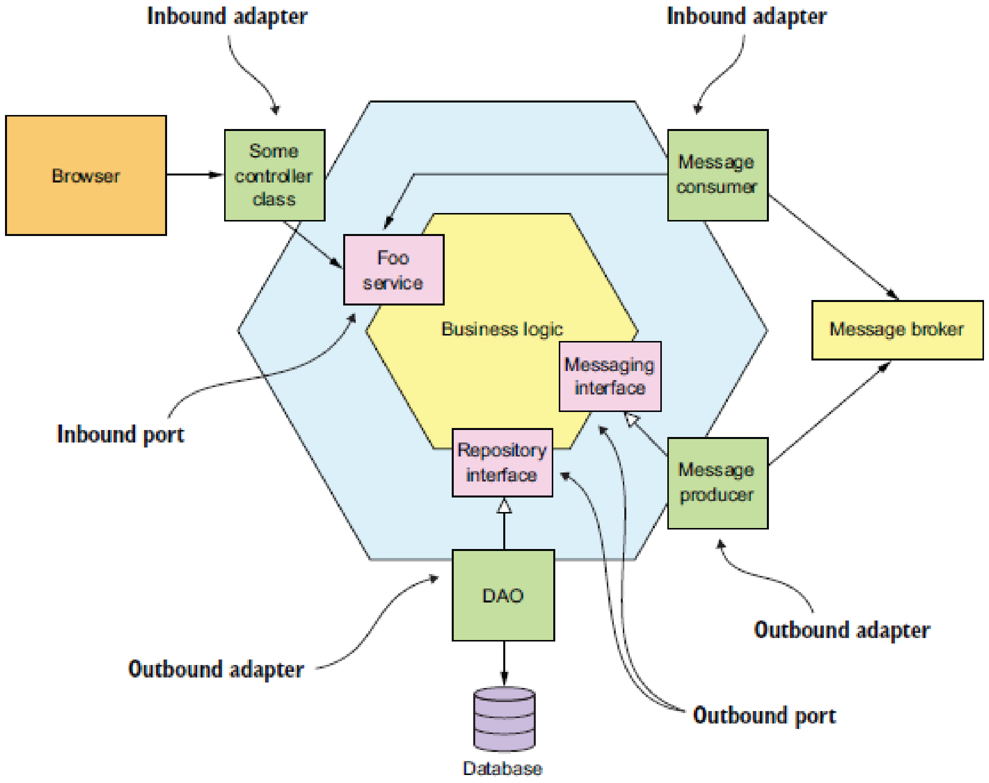
Hexagonal architecture is an alternative to the layered architectural style. As figure 2.2 shows, the hexagonal architecture style organizes the logical view in a way that places the business logic at the center. Instead of the presentation layer, the application has one or more *inbound adapters* that handle requests from the outside by invoking the business logic. Similarly, instead of a data persistence tier, the application has one or more *outbound adapters* that are invoked by the business logic and invoke external applications. A key characteristic and benefit of this architecture is that the business logic doesn't depend on the adapters. Instead, they depend upon it.

- Alternative to layered style

AARHUS UNIVERSITET

Visually

- Benefits
  - BL independent of presentation and data access
  - Easier testing / decoupling
  - Often need for many e.g. UI adapters (web, mobile app, legacy, ..) anyway



So…

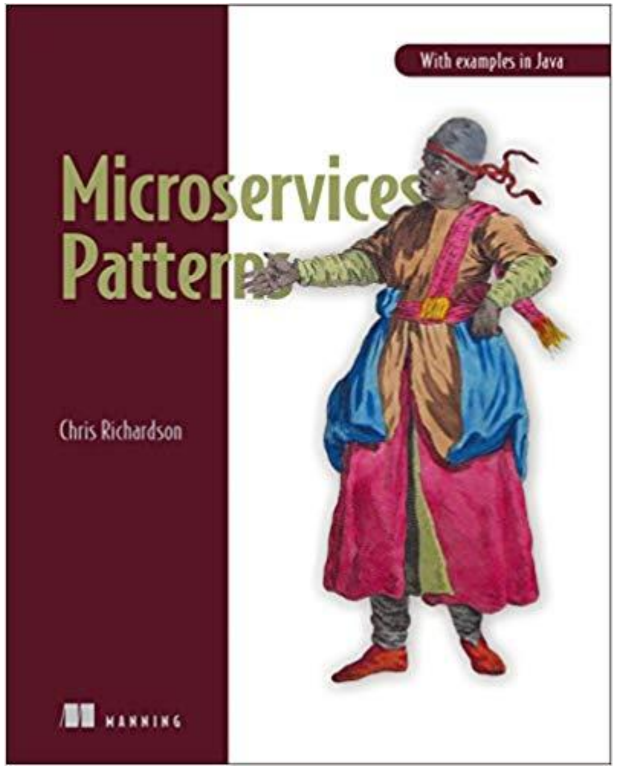
- What has that to do with microservice architectures?

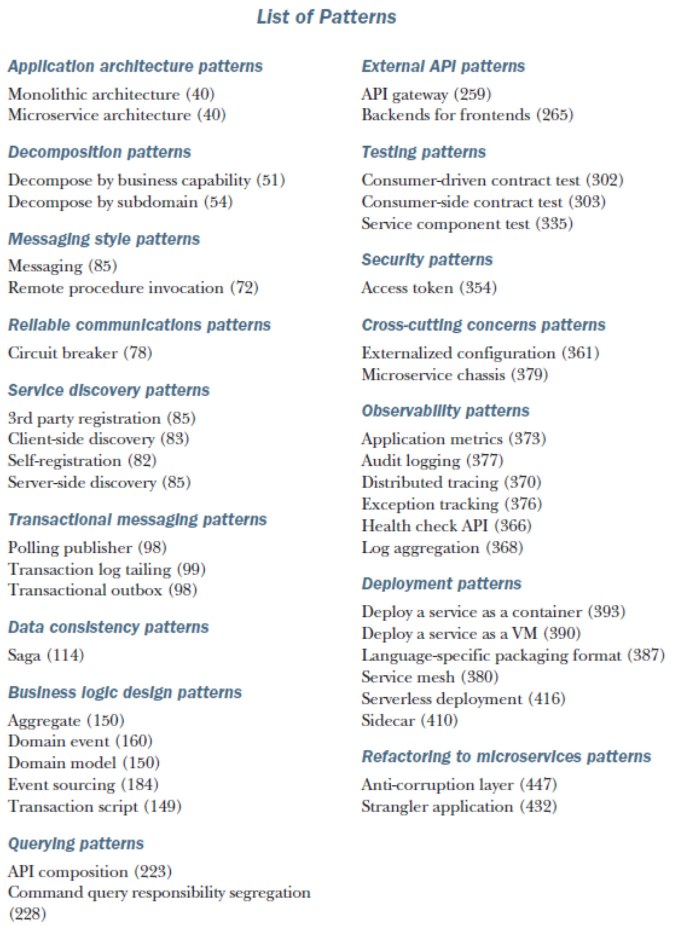
- Nothing, excepts as another way of structuring the kind of applications that most often are candiates for micro service architectures ☺

AARHUS UNIVERSITET

Pattern Languages

Recommendable Book





List of Patterns

Application architecture patterns
Monolithic architecture (40)
Microservice architecture (40)

Decomposition patterns
Decompose by business capability (51)
Decompose by subdomain (54)

Messaging style patterns
Messaging (85)
Remote procedure invocation (72)

Reliable communications patterns
Circuit breaker (78)

Service discovery patterns
3rd party registration (85)
Client-side discovery (83)
Self-registration (82)
Server-side discovery (85)

Transactional messaging patterns
Polling publisher (98)
Transaction log tailing (99)
Transactional outbox (98)

Data consistency patterns
Saga (114)

Business logic design patterns
Aggregate (150)
Domain event (160)
Domain model (150)
Event sourcing (184)
Transaction script (149)

Querying patterns
API composition (223)
Command query responsibility segregation (228)

External API patterns
API gateway (259)
Backends for frontends (265)

Testing patterns
Consumer-driven contract test (302)
Consumer-side contract test (303)
Service component test (335)

Security patterns
Access token (354)

Cross-cutting concerns patterns
Externalized configuration (361)
Microservice chassis (379)

Observability patterns
Application metrics (373)
Audit logging (377)
Distributed tracing (370)
Exception tracking (376)
Health check API (366)
Log aggregation (368)

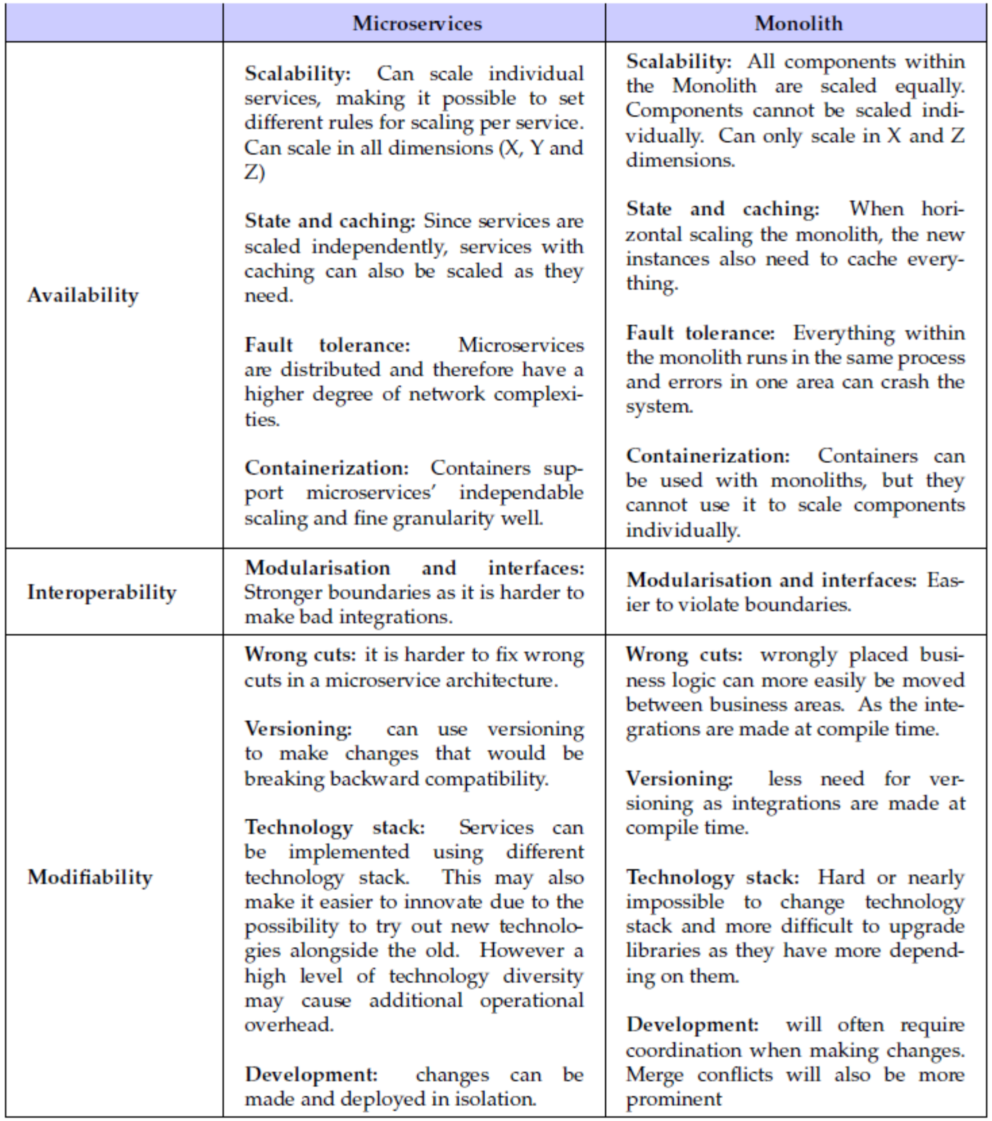
Deployment patterns
Deploy a service as a container (393)
Deploy a service as a VM (390)
Language-specific packaging format (387)
Service mesh (380)
Serverless deployment (416)
Sidecar (410)

Refactoring to microservices patterns
Anti-corruption layer (447)
Strangler application (432)

Recent Analysis

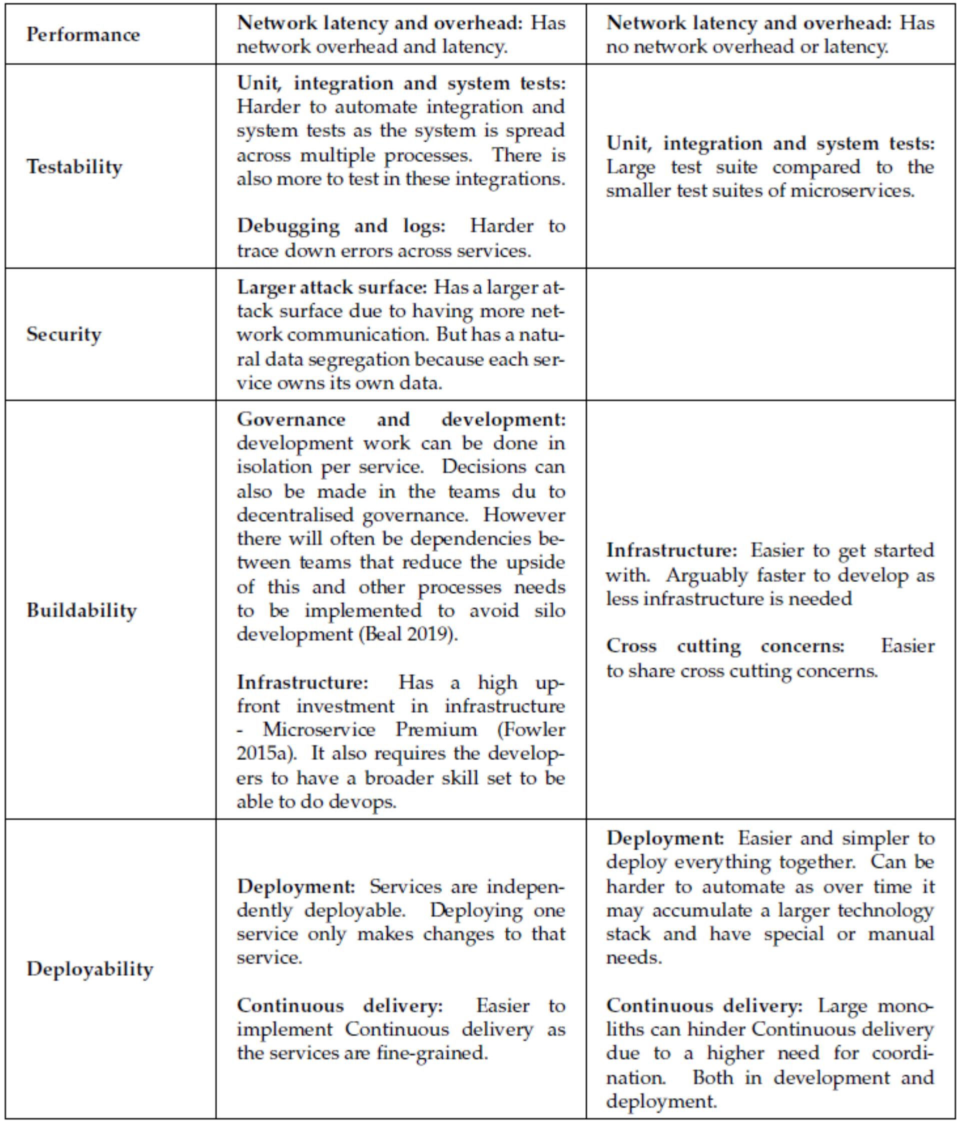
By Peter Daugaard Rasmussen

AARHUS UNIVERSITET

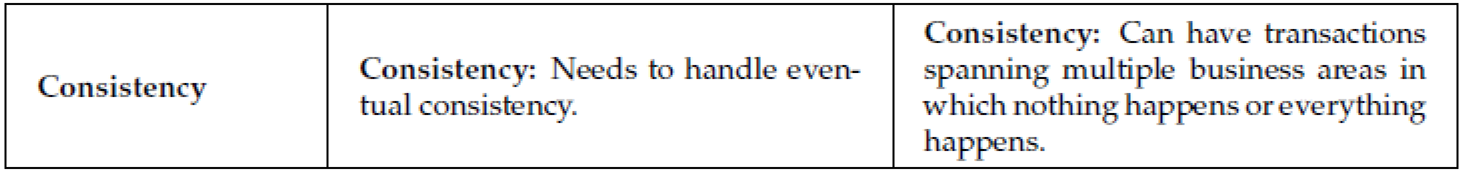


	Microservices	Monolith
Availability	**Scalability:** Can scale individual services, making it possible to set different rules for scaling per service. Can scale in all dimensions (X, Y and Z) **State and caching:** Since services are scaled independently, services with caching can also be scaled as they need. **Fault tolerance:** Microservices are distributed and therefore have a higher degree of network complexities. **Containerization:** Containers support microservices' independable scaling and fine granularity well.	**Scalability:** All components within the Monolith are scaled equally. Components cannot be scaled individually. Can only scale in X and Z dimensions. **State and caching:** When horizontal scaling the monolith, the new instances also need to cache everything. **Fault tolerance:** Everything within the monolith runs in the same process and errors in one area can crash the system. **Containerization:** Containers can be used with monoliths, but they cannot use it to scale components individually.
Interoperability	**Modularisation and interfaces:** Stronger boundaries as it is harder to make bad integrations.	**Modularisation and interfaces:** Easier to violate boundaries.
Modifiability	**Wrong cuts:** it is harder to fix wrong cuts in a microservice architecture. **Versioning:** can use versioning to make changes that would be breaking backward compatibility. **Technology stack:** Services can be implemented using different technology stack. This may also make it easier to innovate due to the possibility to try out new technologies alongside the old. However a high level of technology diversity may cause additional operational overhead. **Development:** changes can be made and deployed in isolation.	**Wrong cuts:** wrongly placed business logic can more easily be moved between business areas. As the integrations are made at compile time. **Versioning:** less need for versioning as integrations are made at compile time. **Technology stack:** Hard or nearly impossible to change technology stack and more difficult to upgrade libraries as they have more depending on them. **Development:** will often require coordination when making changes. Merge conflicts will also be more prominent

AARHUS UNIVERSITET



Performance	**Network latency and overhead:** Has network overhead and latency.	**Network latency and overhead:** Has no network overhead or latency.
Testability	**Unit, integration and system tests:** Harder to automate integration and system tests as the system is spread across multiple processes. There is also more to test in these integrations. **Debugging and logs:** Harder to trace down errors across services.	**Unit, integration and system tests:** Large test suite compared to the smaller test suites of microservices.
Security	**Larger attack surface:** Has a larger attack surface due to having more network communication. But has a natural data segregation because each service owns its own data.	
Buildability	**Governance and development:** development work can be done in isolation per service. Decisions can also be made in the teams du to decentralised governance. However there will often be dependencies between teams that reduce the upside of this and other processes needs to be implemented to avoid silo development (Beal 2019). **Infrastructure:** Has a high upfront investment in infrastructure - Microservice Premium (Fowler 2015a). It also requires the developers to have a broader skill set to be able to do devops.	**Infrastructure:** Easier to get started with. Arguably faster to develop as less infrastructure is needed **Cross cutting concerns:** Easier to share cross cutting concerns.
Deployability	**Deployment:** Services are independently deployable. Deploying one service only makes changes to that service. **Continuous delivery:** Easier to implement Continuous delivery as the services are fine-grained.	**Deployment:** Easier and simpler to deploy everything together. Can be harder to automate as over time it may accumulate a larger technology stack and have special or manual needs. **Continuous delivery:** Large monoliths can hinder Continuous delivery due to a higher need for coordination. Both in development and deployment.



Consistency	**Consistency:** Needs to handle eventual consistency.	**Consistency:** Can have transactions spanning multiple business areas in which nothing happens or everything happens.

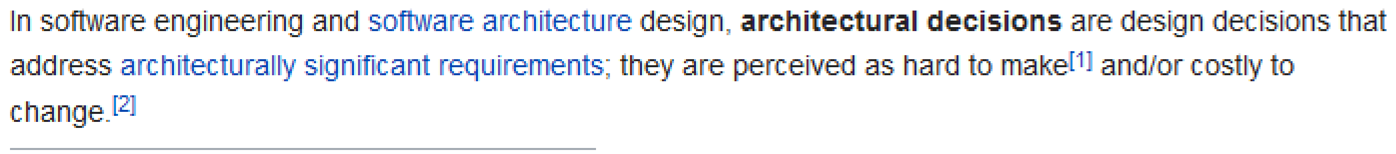
Table 2: Summary of the disadvantages and advantages of microservices

Summary

AARHUS UNIVERSITET

Software Architecture…

- … is difficult !



In software engineering and software architecture design, **architectural decisions** are design decisions that address architecturally significant requirements; they are perceived as hard to make[1] and/or costly to change.[2]

- One definition
  - (forgot the author, and is probably misquoting)
  - *The things we wished we had done differently ☺*

- IMO it is about ‘cutting the cake in the right way’
  - Part the ‘whole’ into ‘suitable’ pieces that collaborate
  - Usually, no ‘right way’ just ‘less bad ways’ to pick from…