Microservices: Patterns & Antipatterns

Stefan Tilkov
stefan.tilkov@innoq.com
@stilkov

innoQ

Let’s talk about words

Let’s talk about patterns

Pattern: ‹Name›

Description

…

Approach

- …
-

Consequences

- …

Pattern: Microservices

Description

Design modules as separate deployment and operation units, with large degrees of freedom for their implementation

Approach

- Former technical detail (deployment architecture) as first class architectural design principle
- Network communication as hard-to-cross boundary, enforcing encapsulation

Consequences

- Isolation
- Autonomy
- Scalability
- Resilience
- Speed
- Experimentation
- Rapid Feedback
- Flexibility
- Replaceability

Pattern: Evolutionary Architecture

Pattern: Evolutionary Architecture

Description

Architecture is constructed so it can evolve as much as possible over the course of (ideally indefinite) time

Approach

- Separation of large domain into “islands of change”
- Design for replacement, not for re-use
- Minimization of shared dependencies

Consequences

- Cell metaphor: Renewal over time
- Experimentation with different micro architecture approaches possible

Antipattern: ‹Name›

Description

…

Reasons

- …
-

Consequences

- …
-

Antipattern: Distributed Monolith
(a.k.a. “Microservices Gone Bad”)

Description

System made up of arbitrarily sized, tightly coupled modules communicating over network interfaces

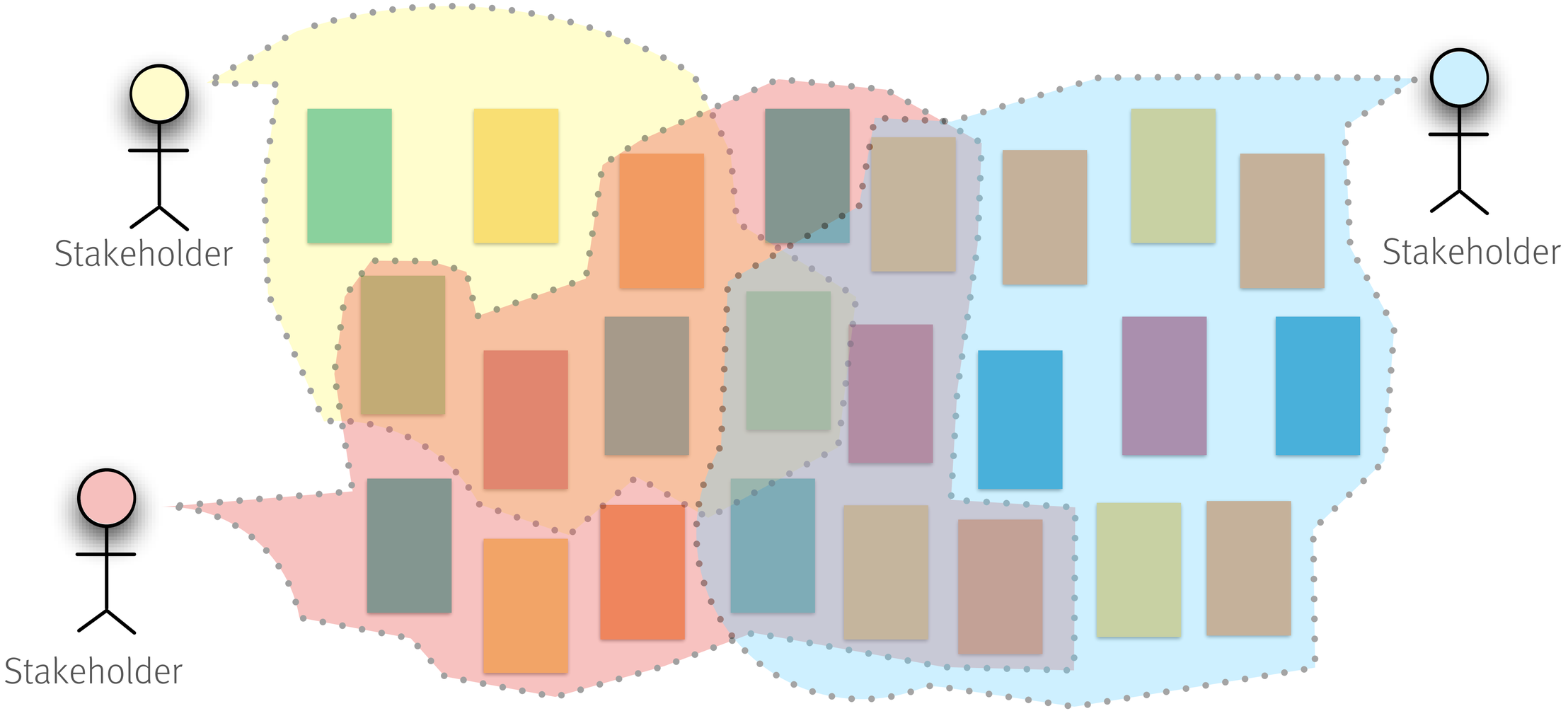
Reasons

- Hype-driven architecture
- Conference-driven development
- Missing focus on business domain
- Infrastructure over-engineering

Consequences

- “Ripple” effect of changes
- Complex environment
- Massive network overhead
- Performance issues
- Wild mix of technologies, products & frameworks
- Hard to understand & maintain

Antipattern: Decoupling Illusion



Antipattern: Decoupling Illusion

Description

Functional changes required by different stakeholders require changes to overlapping services

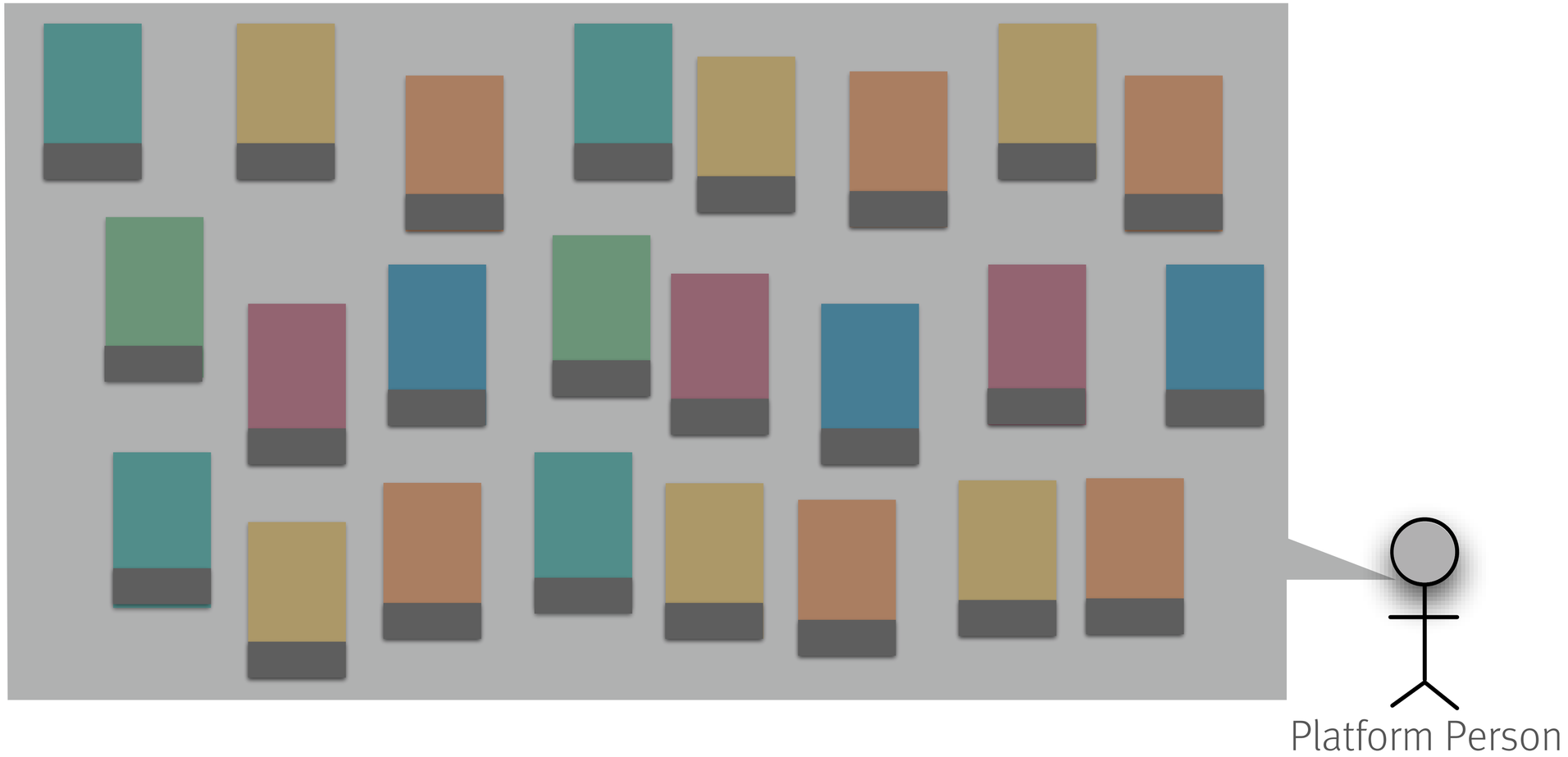
Reasons

- No alignment of organization and architecture
- Fine-grained services
- Too much focus on re-use

Consequences

- High cost
- High technical complexity
- Reduced or no benefits in terms of increased agility

Antipattern: Micro Platform



Antipattern: Micro Platform

Description

Standardization of service-internal runtime aspects

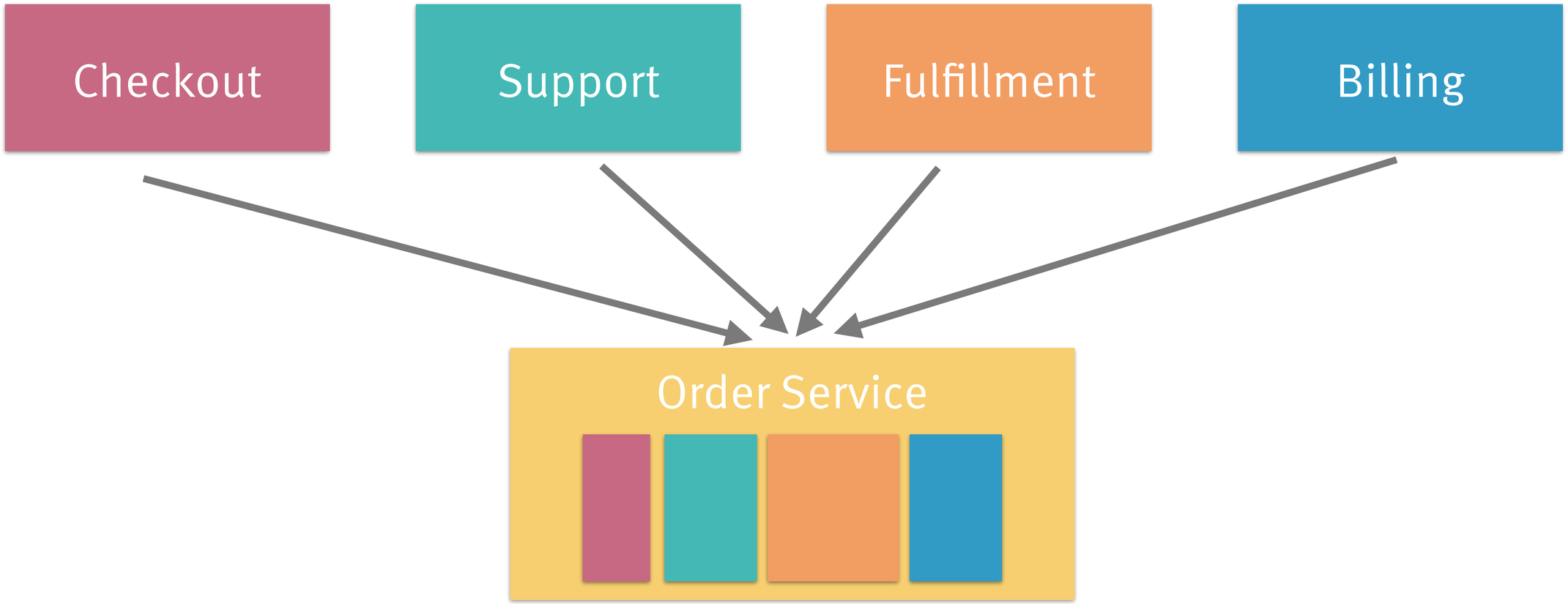
Reasons

- (Perceived) Increased efficiency
- Easier handling of cross-cutting concerns
- “Domain allergy”

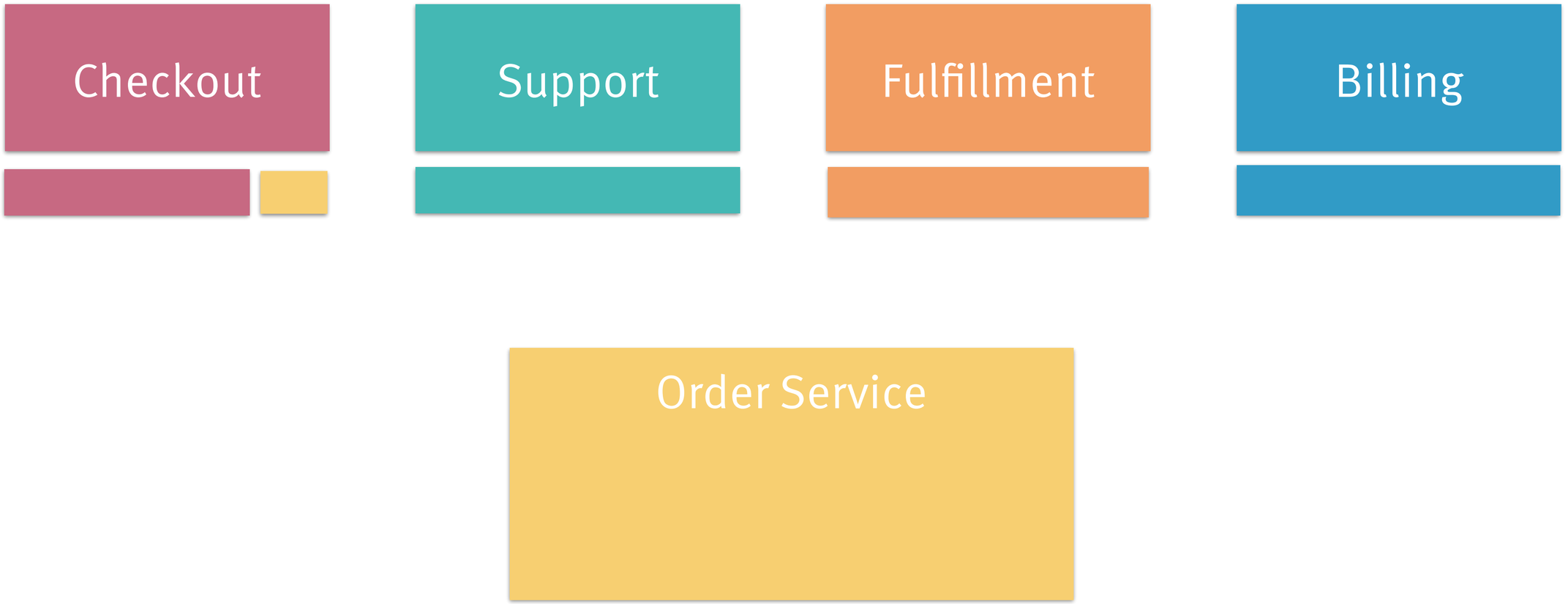
Consequences

- Shared dependency on implementation details
- Bottleneck for changes
- Co-ordinated updates, evolution, deployment

Antipattern: Entity Service



Antipattern: Entity Service (resolved)



Antipattern: Entity Service

Description

Services boundaries are chosen to encapsulate “wide” business entities

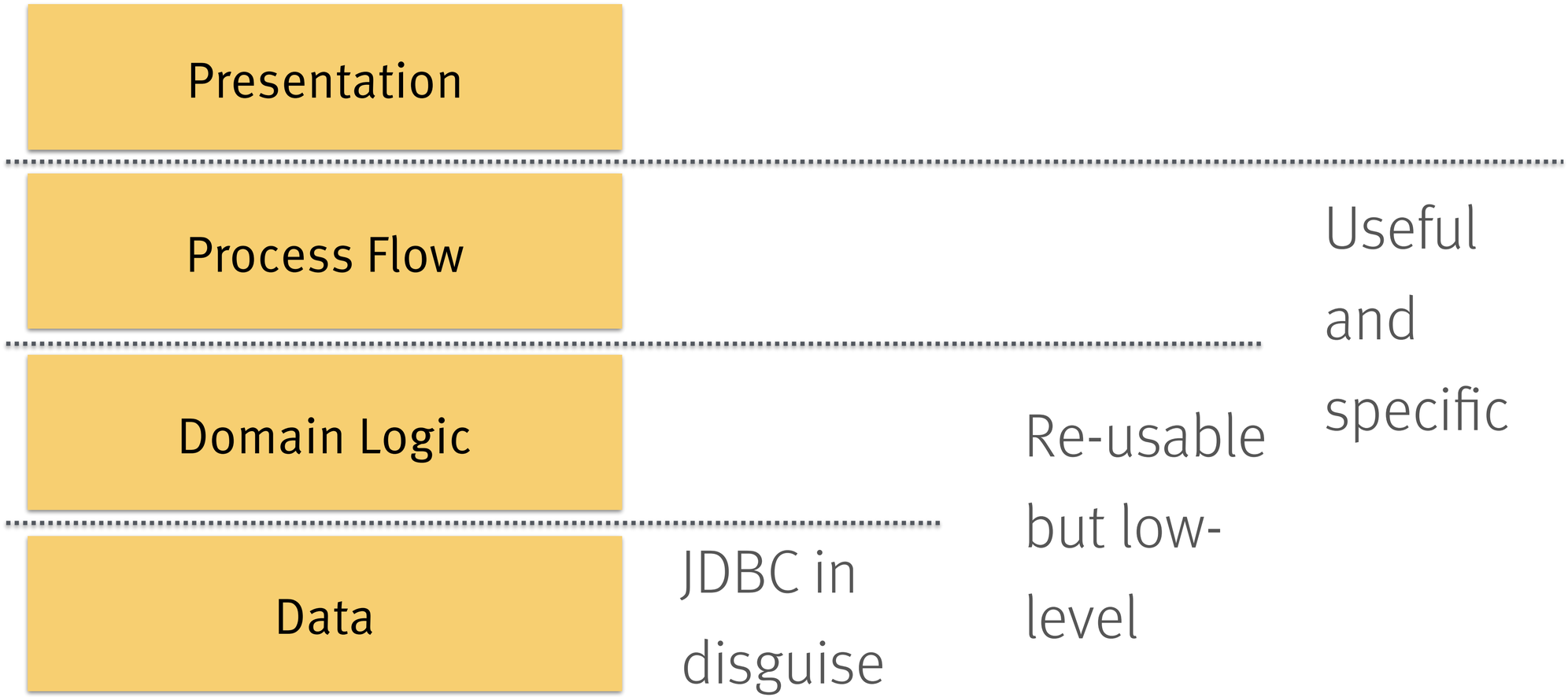
Reasons

- Naive domain modeling
- Perceived benefits of canonical models
- Violation of interface segregation principle

Consequences

- Distributed responsibility Process bottlenecks
- Performance & scalability issues
- High network overhead

Antipattern: Anemic Service



Antipattern: Anemic Service

Description

Services designed to solely encapsulate data, with logic left to the caller

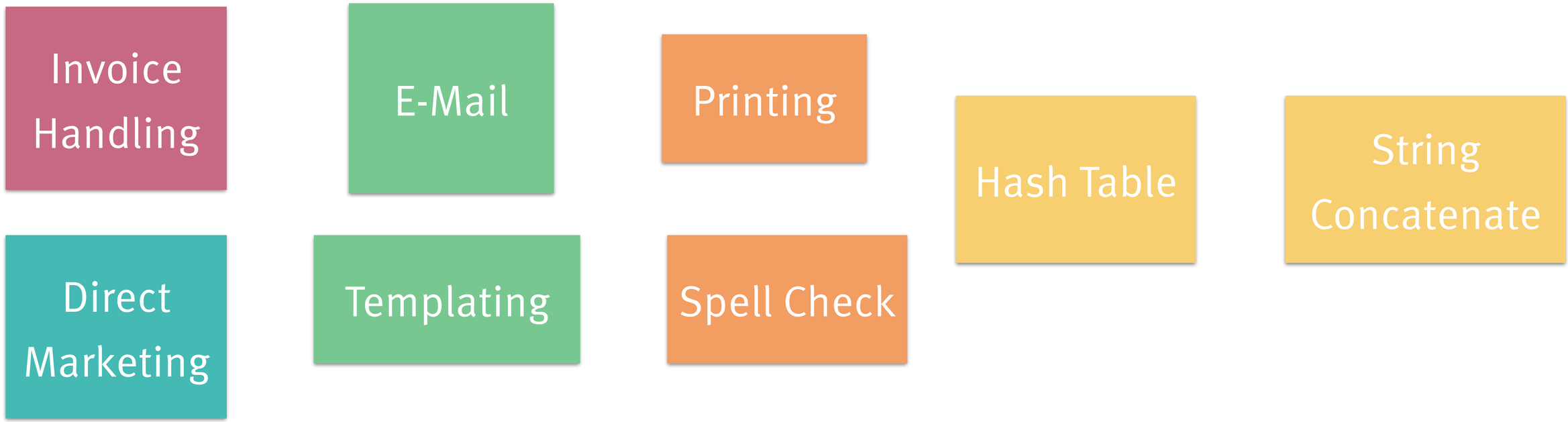
Reasons

- Easily derived from simple domain (E/R) models
- Reduces effort to agree on specifics
- Maximizes re-use

Consequences

- Increased network overhead
- Useless bottlenecks
- Performance issues

Antipattern: Unjustified Re-Use



Antipattern: Unjustified Re-Use

Description

Extremely generic utility functions to reduce logic redundancy

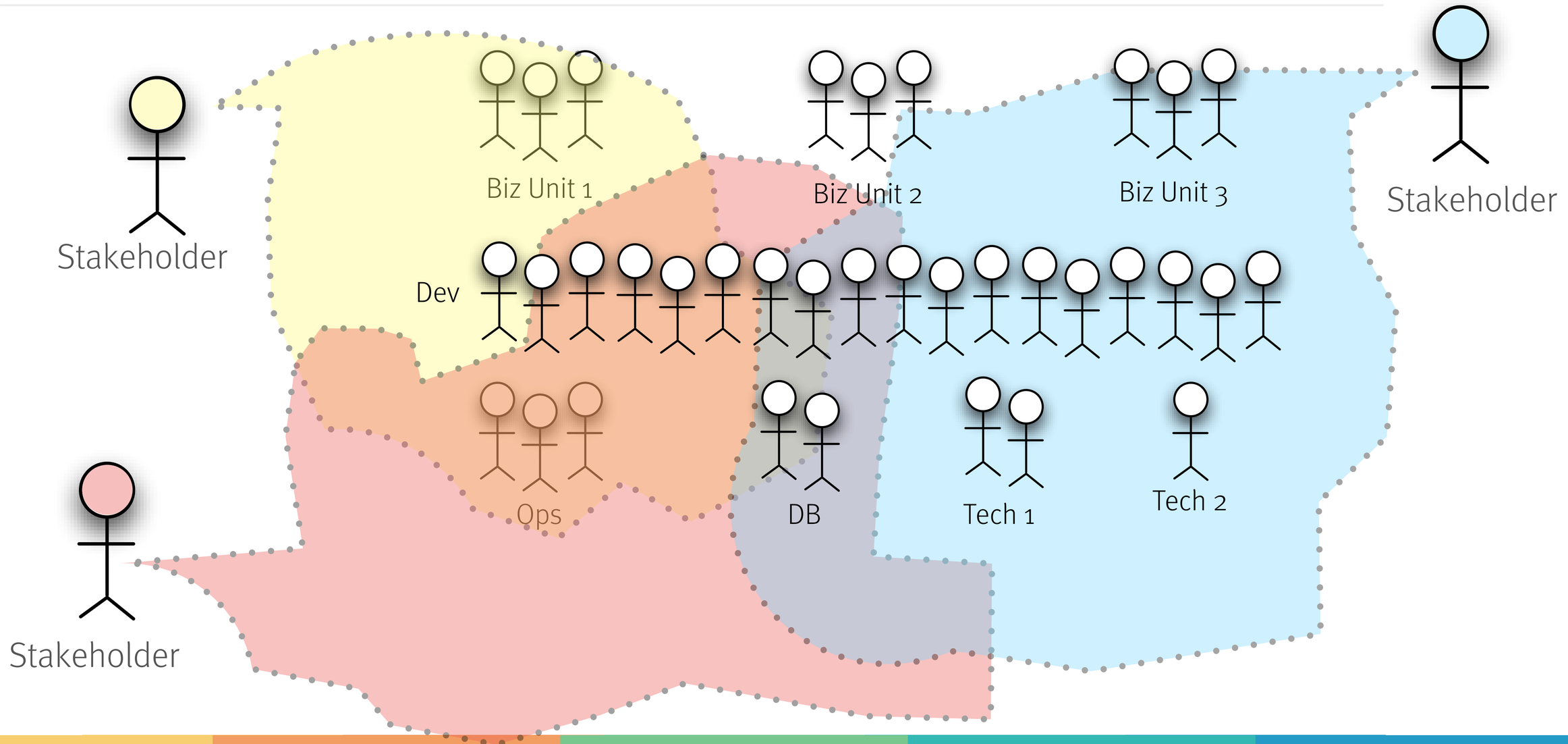
Reasons

- Generalization drive
- Domain allergy

Consequences

- Network overhead
- Increased complexity
- Bottlenecks

Antipattern: Domain-last Approach



Antipattern: Domain-last Approach

Description

Major driver for organizational structure is roles and technical capabilities, not business domain

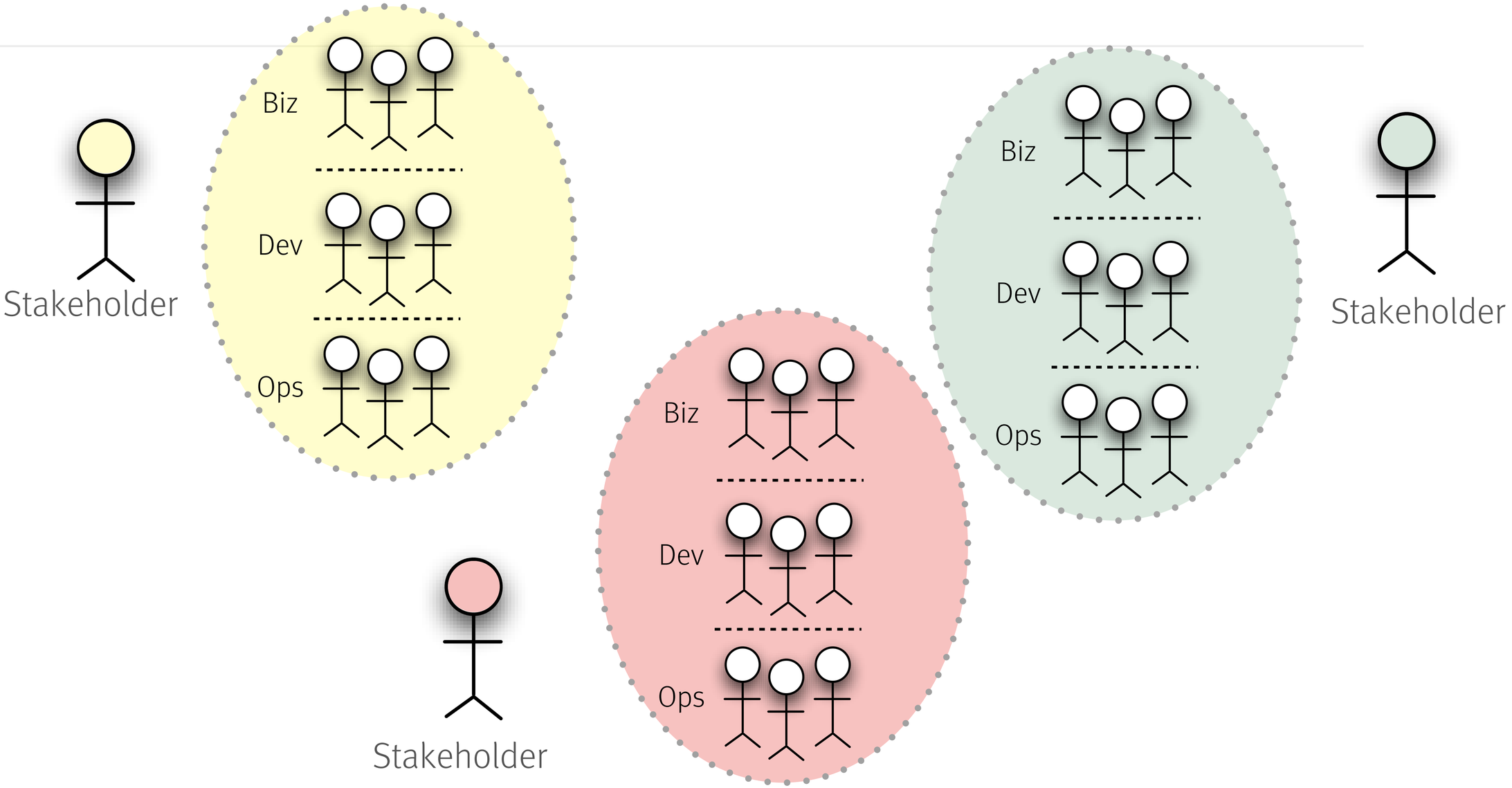
Reasons

- Matches classical company structure
- Division of labor in divisions, department, teams
- Projects as exceptions to change something that works

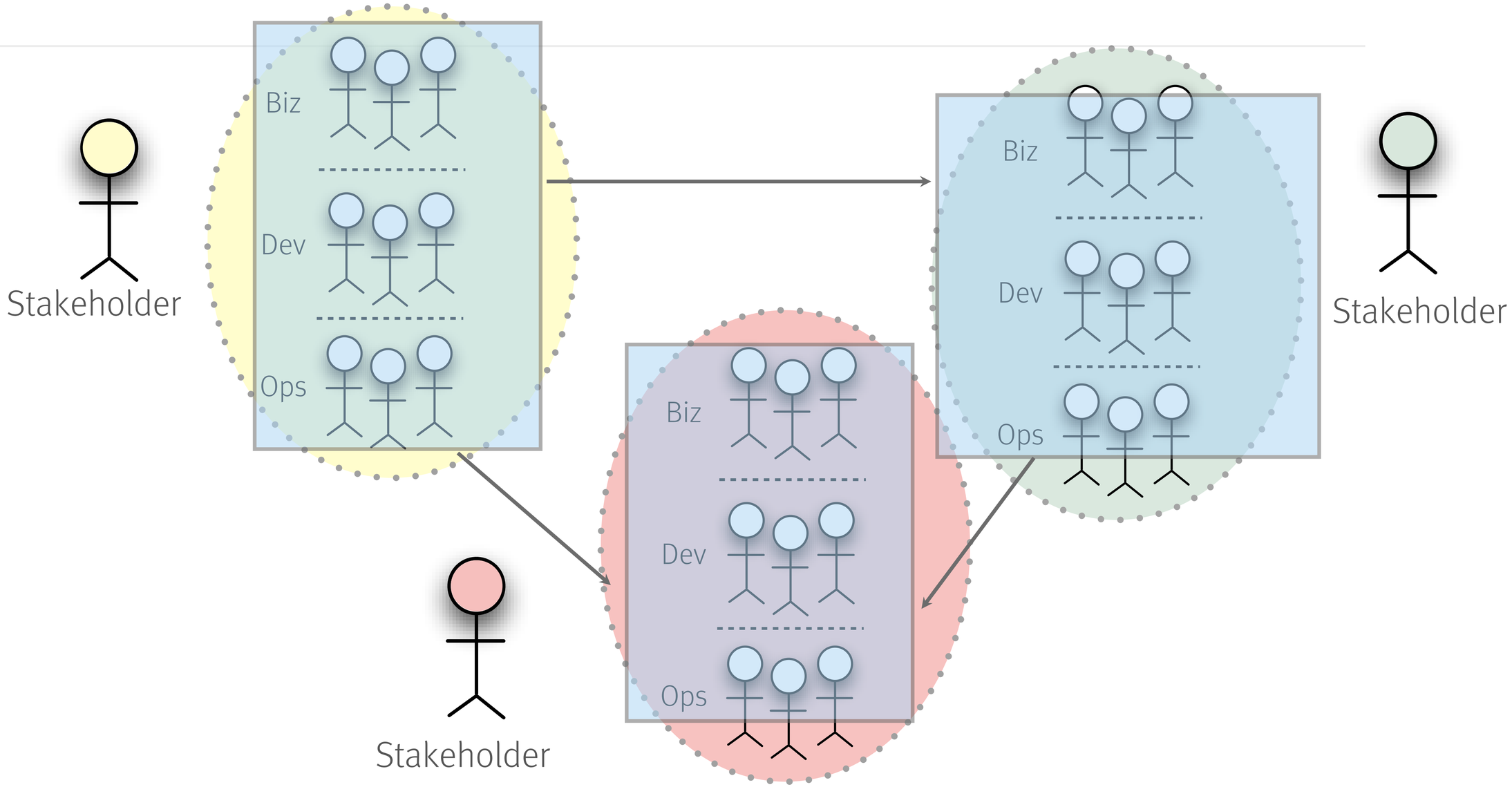
Consequences

- Inter-departmental politics over business needs
- Conflicting project and disciplinary hierarchies and stakeholders
- Blameshifting

Pattern: Autonomous Cells



Pattern: Autonomous Cells



Pattern: Autonomous Cells

Description

Decentralized, domain-focused cells with maximum authority over all aspects of a set of capabilities

Approach

- Decisions are made locally on all aspects of a solution
- Success is measured via customer-oriented KPIs
- Cross-functional team with biz, dev, ops skills

Consequences

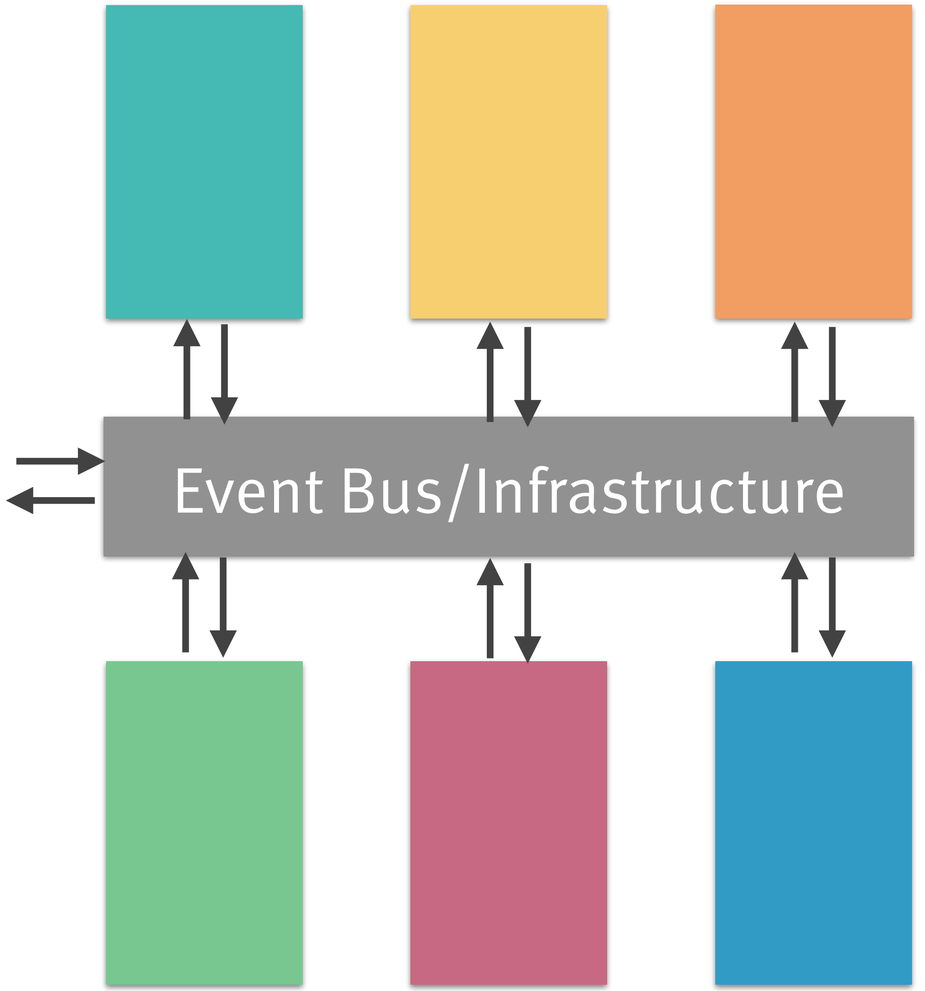
- Customer/end user focus
- Decentralized delivery capability
- Speed as #1 priority
- “Full-stack” requirement for developers and other roles
- Redundancy instead of centralization

Sizing Patterns

Example: Pricing Engine

- Default product prices
- General discounts
- Customer-specific discounts
- Campaign-related rebates

→FaaS



Pattern: FaaS (Function as a Service)

Description:

- As small as possible
- A few hundred lines of code or less
- Triggered by events
- Communicating asynchronously

As seen on:

- Any recent Fred George talk
- Serverless Architecture(*)
- AWS Lambda

(*) https://leanpub.com/serverless

Pattern: FaaS (Function as a Service)

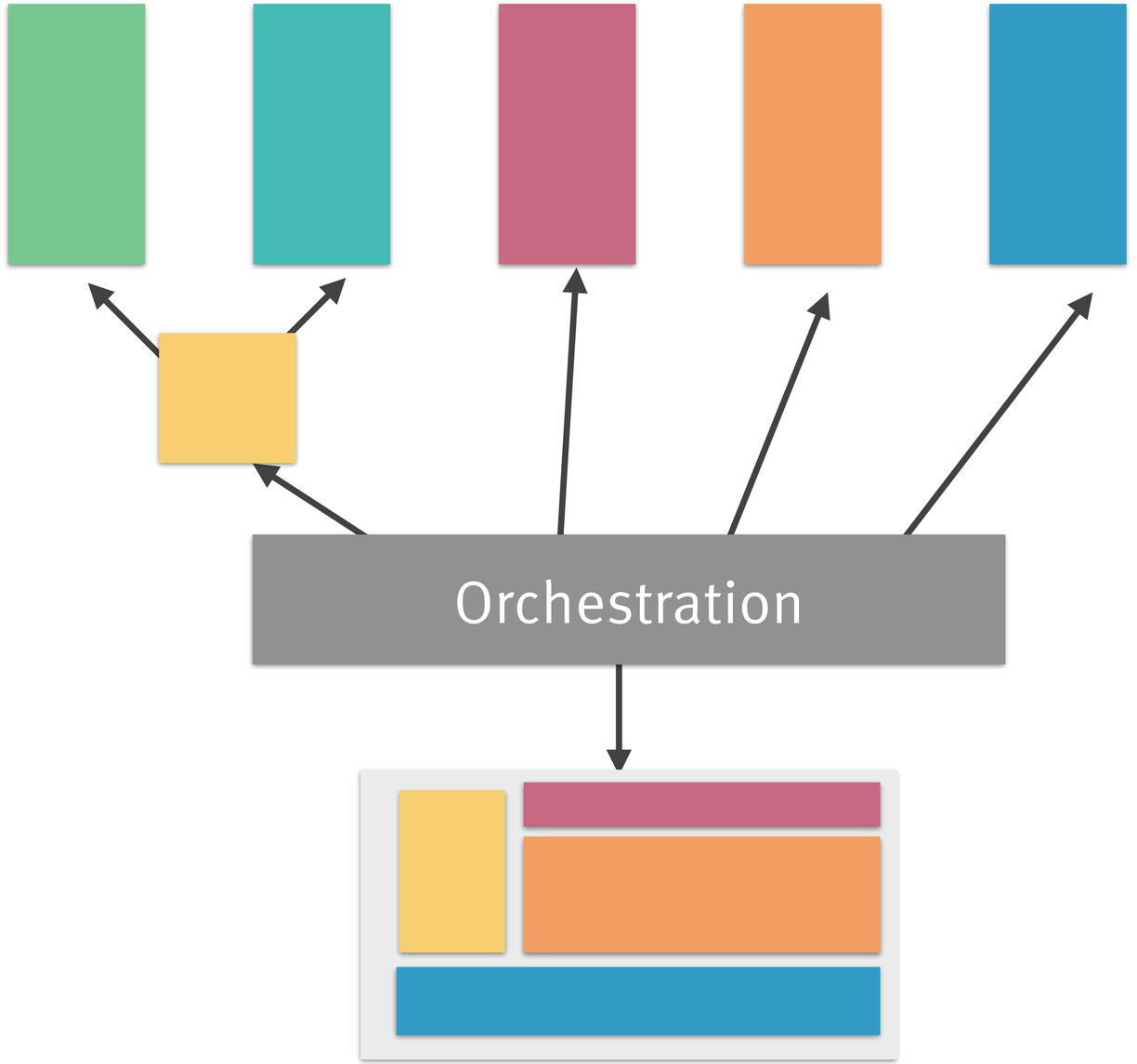
Consequences:

- Shared strong infrastructure dependency
- Common interfaces, multiple invocations
- Similarity to actor-based environments
- Emerging behavior (a.k.a. “what the hell just happened?”)
- Well suited to decomposable/“fuzzy” business problems

Example: Product Detail Page

- Core product data
- Prose description
- Images
- Reviews
- Related content

→ µSOA



Pattern: µSOA (Microservice-oriented Architecture)

Description:

- Small, self-hosted
- Communicating synchronously
- Cascaded/streaming
- Containerized

As seen on:

- Netflix
- Twitter
- Gilt

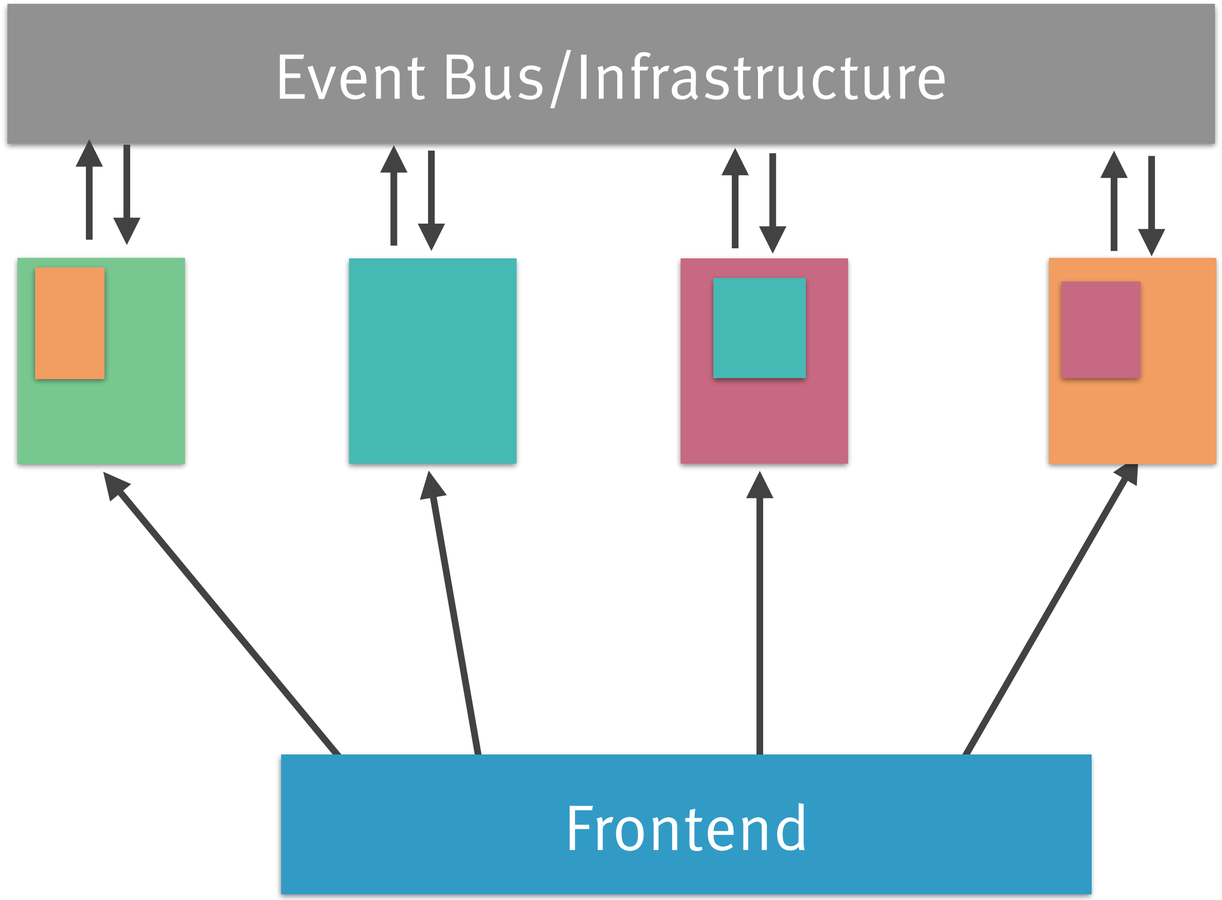
Pattern: µSOA (Microservice-oriented Architecture)

Consequences:

- Close collaboration – common goal
- Need for resilience/stability patterns for invocations
- High cost of coordination (versioning, compatibility, …)
- High infrastructure demand
- Often combined with parallel/streaming approach
- Well suited to environments with extreme scalability requirements

Example: Logistics Application

- Order management
- Shipping
- Route planning
- Invoicing



→DDDD

Pattern: DDDD (Distributed Domain-driven Design)

Description:

- Small, self-hosted
- Bounded contexts
- Redundant data/CQRS
- Business events
- Containerized

As seen on:

- (undisclosed)

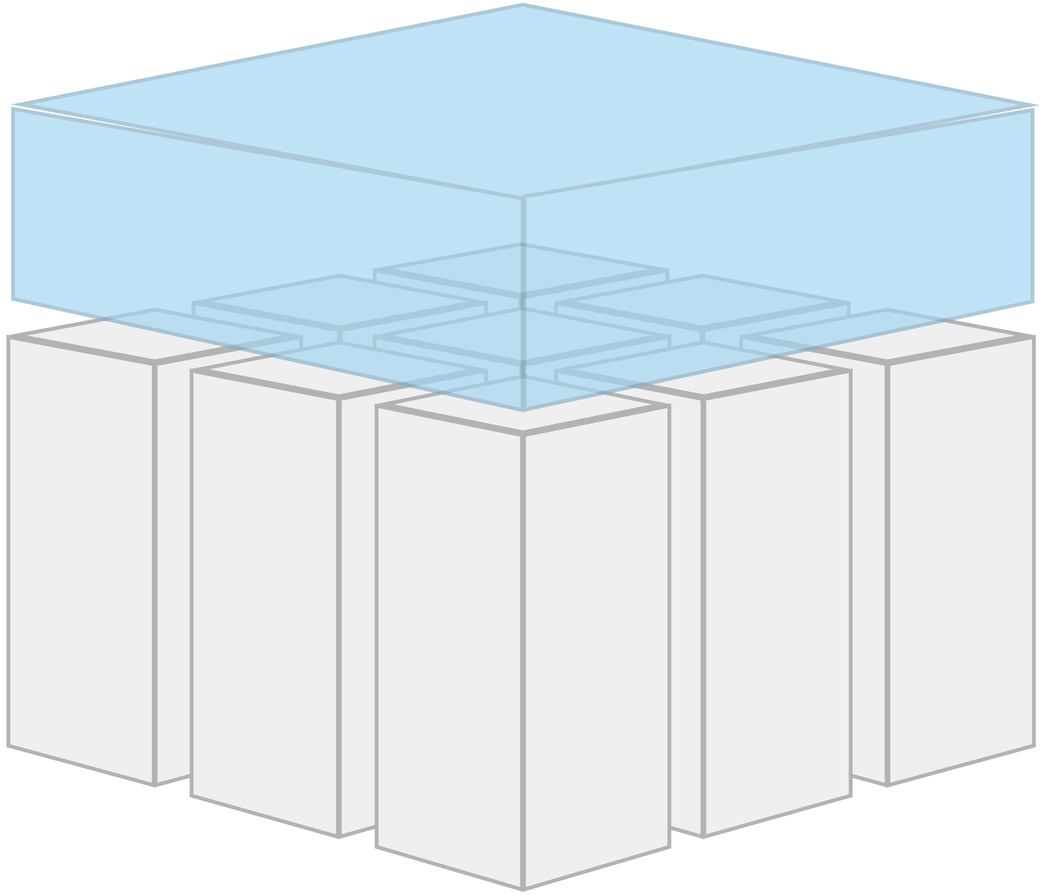
Pattern: DDDD (Distributed Domain-driven Design)

Consequences:

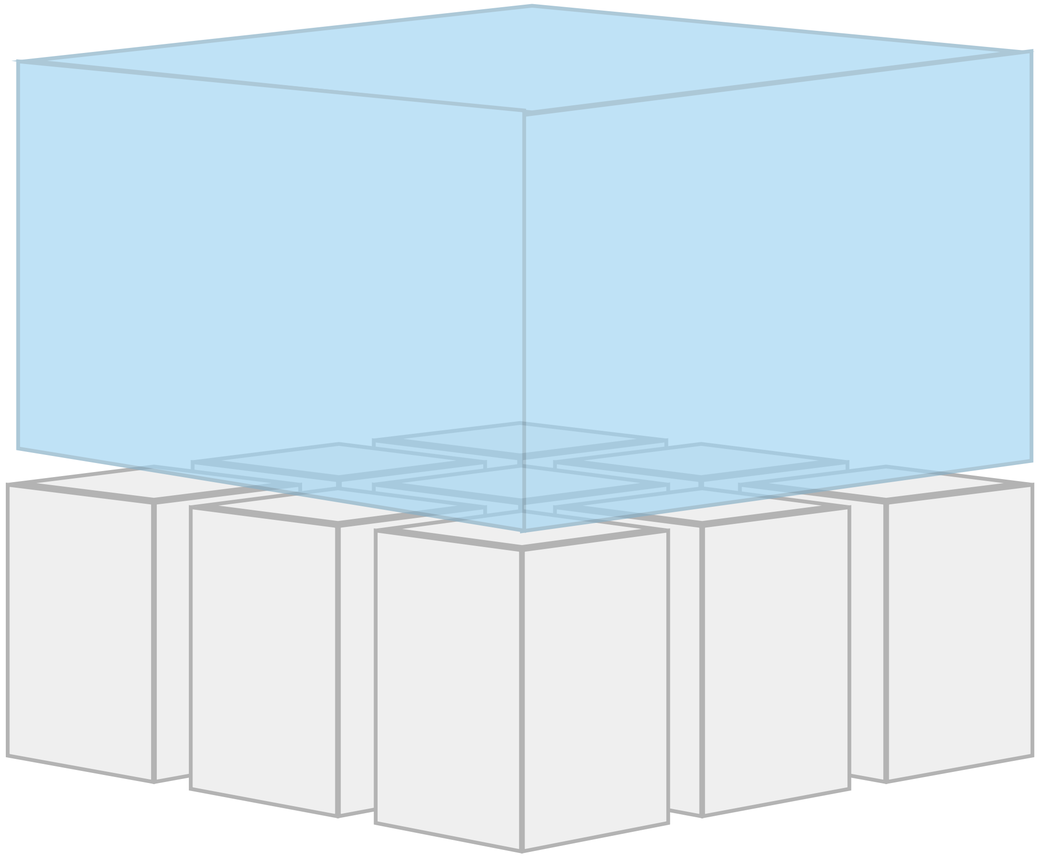
- Loose coupling between context
- Acknowledges separate evolution of contexts
- Asynchronicity increases stability
- Well-suited for to support parallel development

That UI thing? Easy!

Assumption



Reality



Antipattern: Frontend Monolith

Description

Anemic services joined by a monolithic frontend application that contains most of the business logic

Reasons

- (Perceived) necessary for homogenous UX
- Very few developers with frontend knowledge
- (Perceived) lack of frontend platform standardization
- Architects with focus on backend

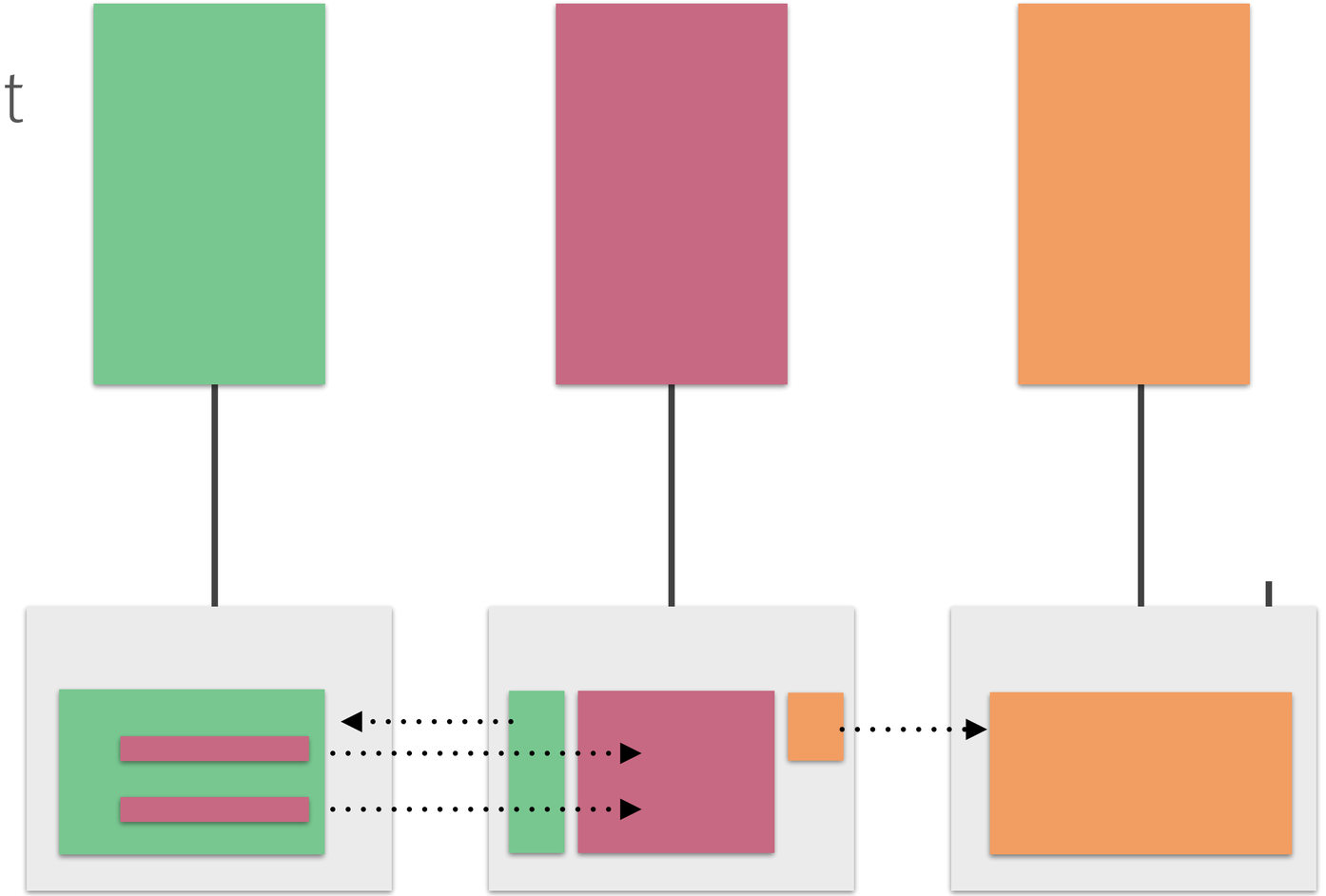
Consequences

- Strong dependency on individual frameworks and/or tooling
- Bottlenecks during development
- Complex and time-consuming evolution due to lack of modularity

Example: E-Commerce Site

- Register & maintain account
- Browse catalog
- See product details
- Checkout
- Track status

→SCS



Pattern: SCS (Self-contained System)

Description:

- Self-contained, autonomous
- Including UI + DB
- Possibly composed of smaller microservices

As seen on:

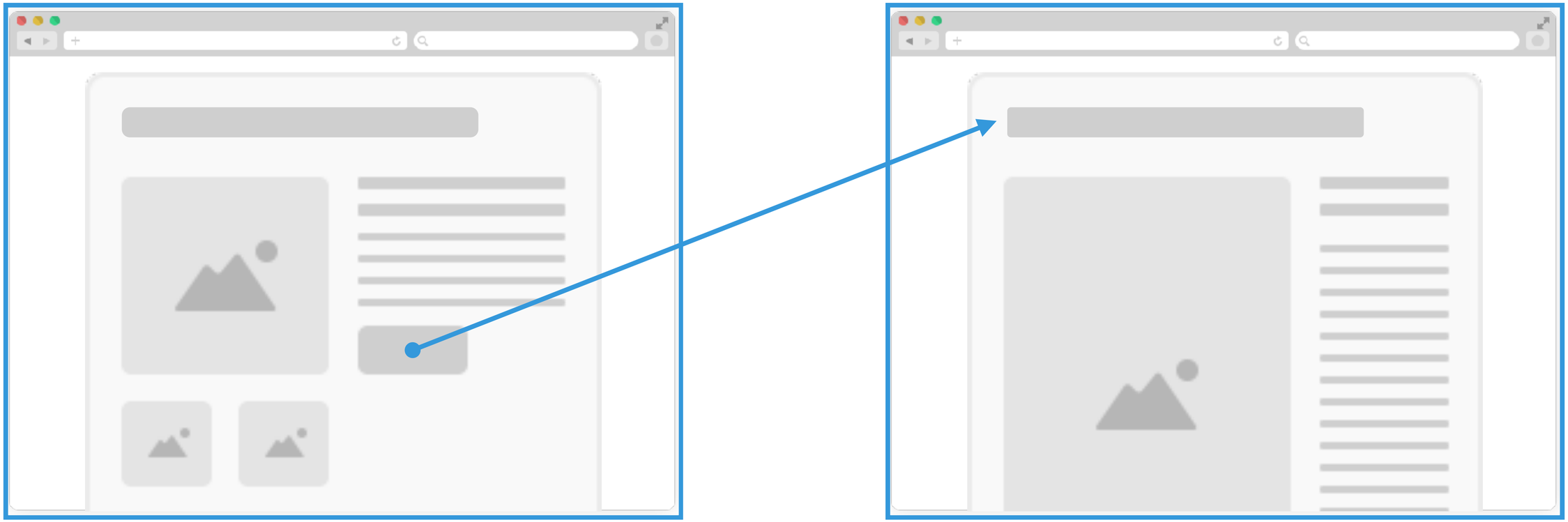
- Amazon
- Groupon
- Otto.de
- https://scs-architecture.org

Pattern: SCS (Self-contained System)

Consequences:

- Larger, independent systems, Including data + UI (if present)
- Able to autonomously serve requests
- Light-weight integration, ideally via front-end
- No extra infrastructure needed
- Well suited if goal is decoupling of development teams

Pattern: Web-based UI Integration

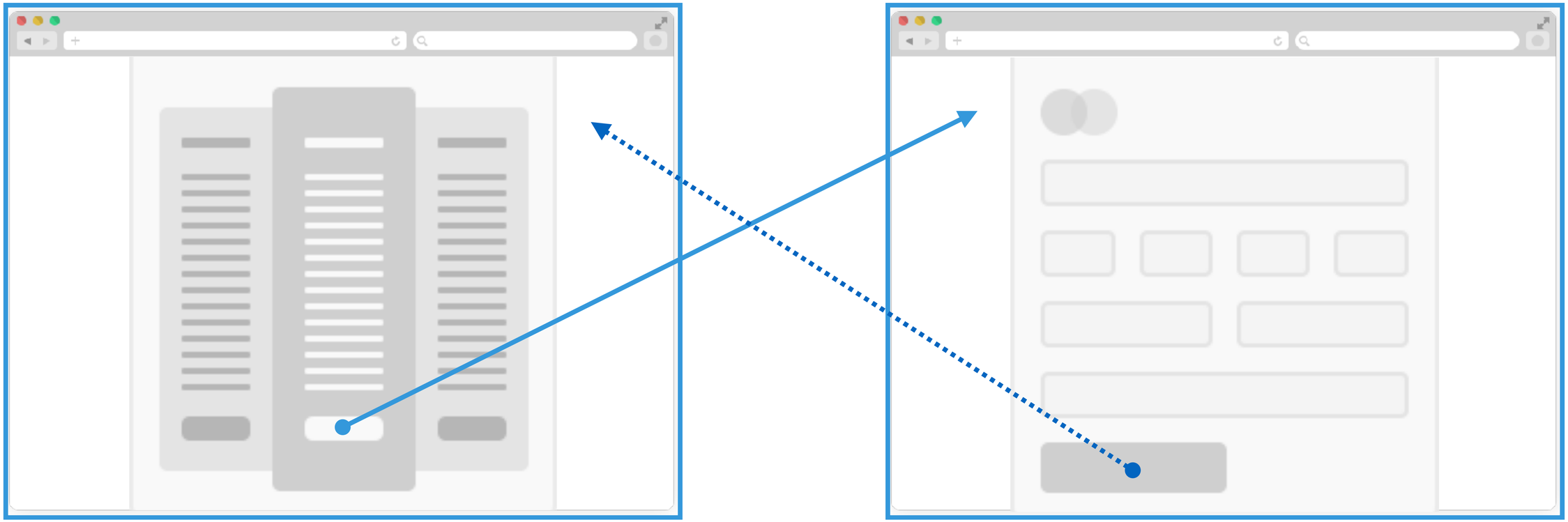


System 1

System 2

→Links

Pattern: Web-based UI Integration

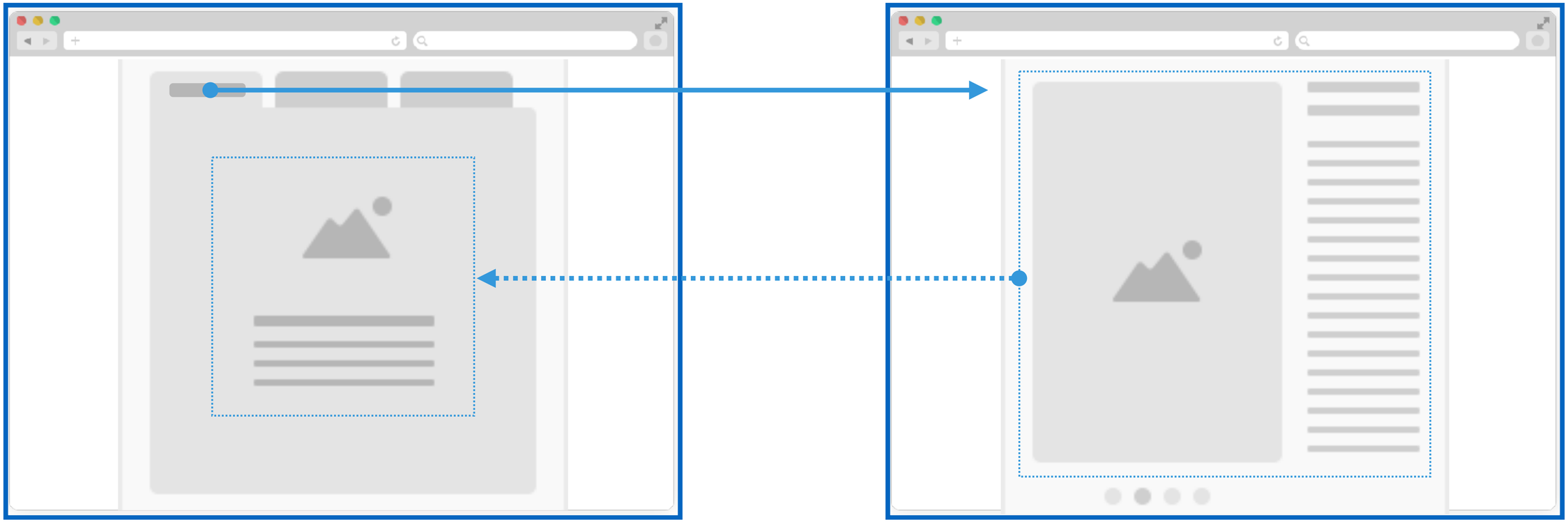


System 1

System 2

→Redirection

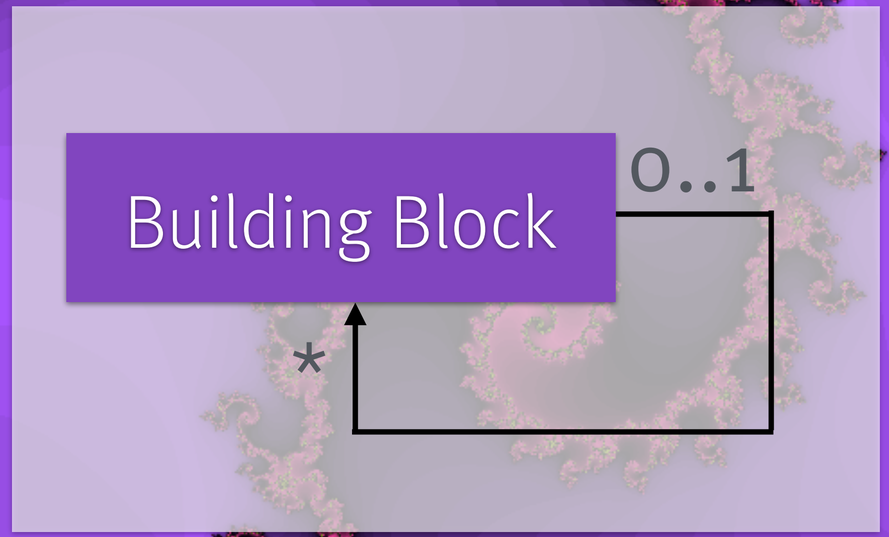
Pattern: Web-based UI Integration



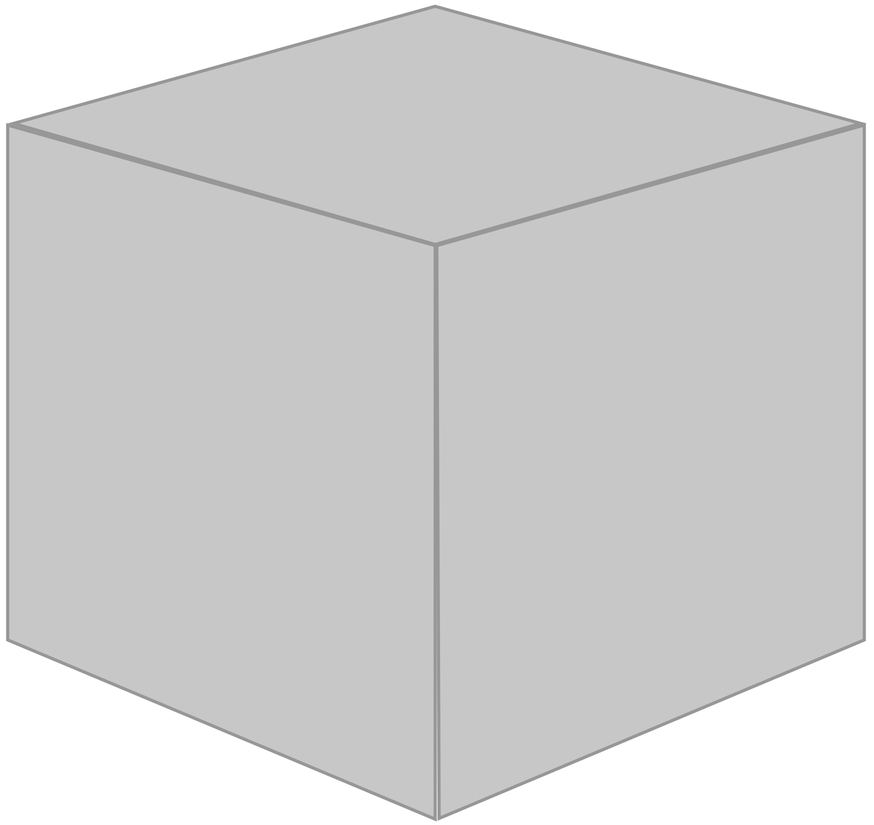
System 1

System 2

→Transclusion


Building Block
0..1
*

The final pattern …



Pattern: Monolith

Description

Highly cohesive, tightly integrated, single unit of deployment application

Approach

- Standard application
- Internal modularity
- No artificially introduced distribution
- Single unit of development and evolution

Consequences

- Straightforward development
- Easy to refactor
- Homogeneous technical choices
- Ideally suited for single small team

We love monoliths –

so let's build a lot of them!

Final Recommendations

1.

Be careful with silver bullets

2.

Prioritize intended benefits, choose matching solutions

3.
Create evolvable structures

That’s all I have.

thanks for listening!

@stilkov

Stefan Tilkov

stefan.tilkov@innoq.com

Phone: +49 170 471 2625

innoQ

www.innoq.com

innoQ Deutschland GmbH

Krischerstr. 100
40789 Monheim am Rhein
Germany
Phone: +49 2173 3366-0

Ohlauer Straße 43
10999 Berlin
Germany
Phone: +49 2173 3366-0

Ludwigstr. 180E
63067 Offenbach
Germany
Phone: +49 2173 3366-0

Kreuzstraße 16
80331 München
Germany
Phone: +49 2173 3366-0

innoQ Schweiz GmbH

Gewerbestr. 11
CH-6330 Cham
Switzerland
Phone: +41 41 743 0116

Image Credit

https://pixabay.com/en/chaos-room-untidy-dirty-messy-627218/

https://commons.wikimedia.org/wiki/File:Wroclaw_Daily_Market.jpg

https://pixabay.com/en/smartphone-face-man-old-baby-1790833/

https://pixabay.com/en/marketing-customer-center-2483856/

About Stefan Tilkov

- CEO/Co-founder & principal consultant at innoQ
- Author and frequent conference speaker
- stefan.tilkov@innoq.com
- @stilkov

About innoQ

- Offices in Monheim (near Cologne), Berlin, Offenbach, Munich, Zurich
- ~125 employees
- Core competencies: software architecture consulting and software development
- Privately owned, vendor-independent
- Clients in finance, telecommunications, logistics, e-commerce; Fortune 500, SMBs, startups

www.innoq.com