1.– 4. September 2014
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Moderne Zeiten

Architekturen für eine Next Generation IT

Uwe Friedrichsen

codecentric AG

@ufried

Uwe Friedrichsen | uwe.friedrichsen@codecentric.de | http://slideshare.net/ufried | http://ufried.tumblr.com

Why do we need a “Next Generation IT”?

Economic Darwinism

Economic Darwinism

Everyone is affected by Economic Darwinism

- All sectors
- Growing globalization on all levels
- Internet business
- More competitors per customer
- Higher customer expectations
- Lower customer loyalty

→ In the long run only those will survive who meet the customer needs and demands best

Nice, but how does this relate to IT?

IT is the nervous system

IT is vital

- All companies
- IT is not just supporter or „cost center“ …
- … but it is the central nervous system
- Even short IT outages considered critical
- No business change without IT
- No new products without IT

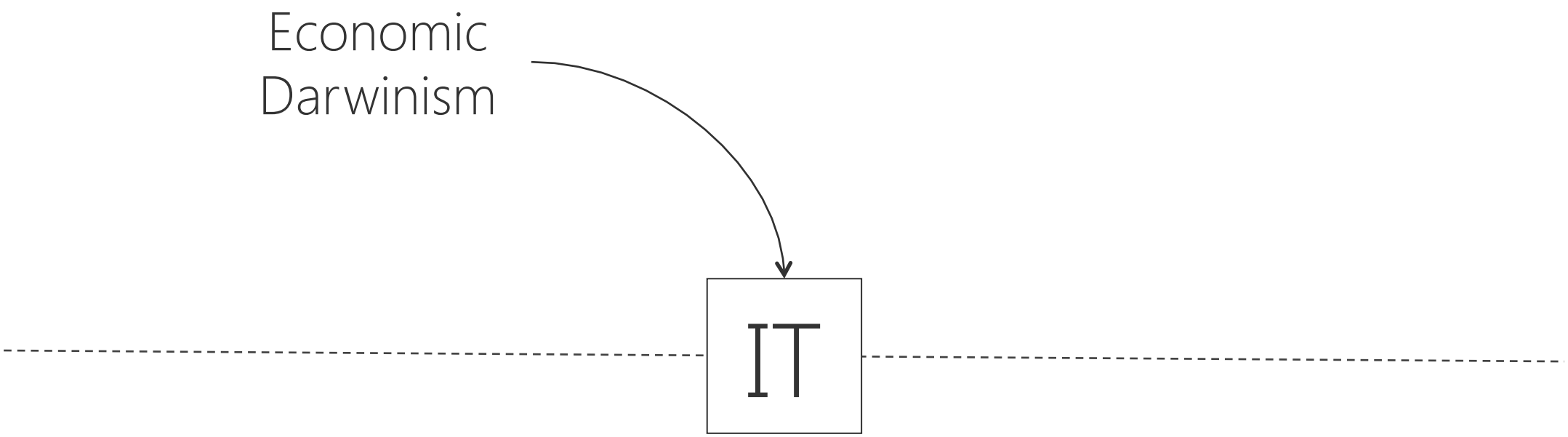
→ IT limits the maximum possible adaption rate of a company

IT is a key success factor for belonging to the survivors of the economic darwinism

What business needs from IT …

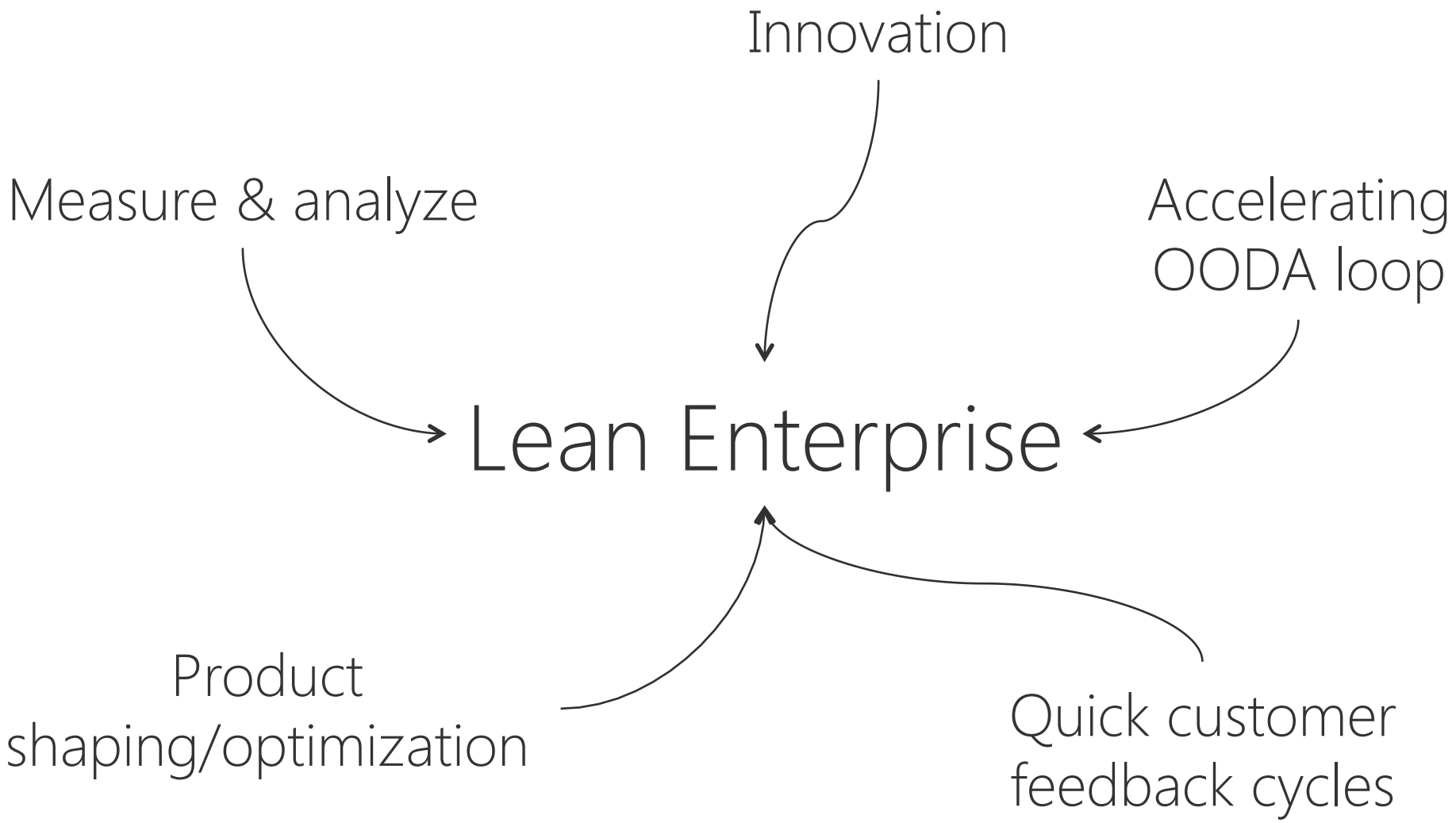
How IT serves business …

Business-related
Change Drivers

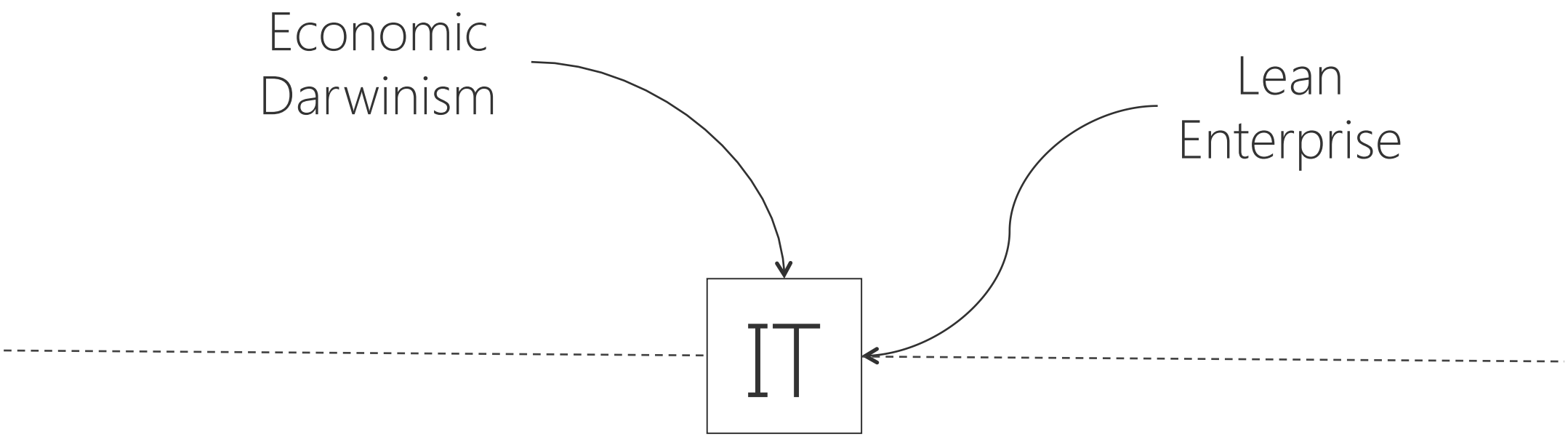


Technology-related
Change Drivers

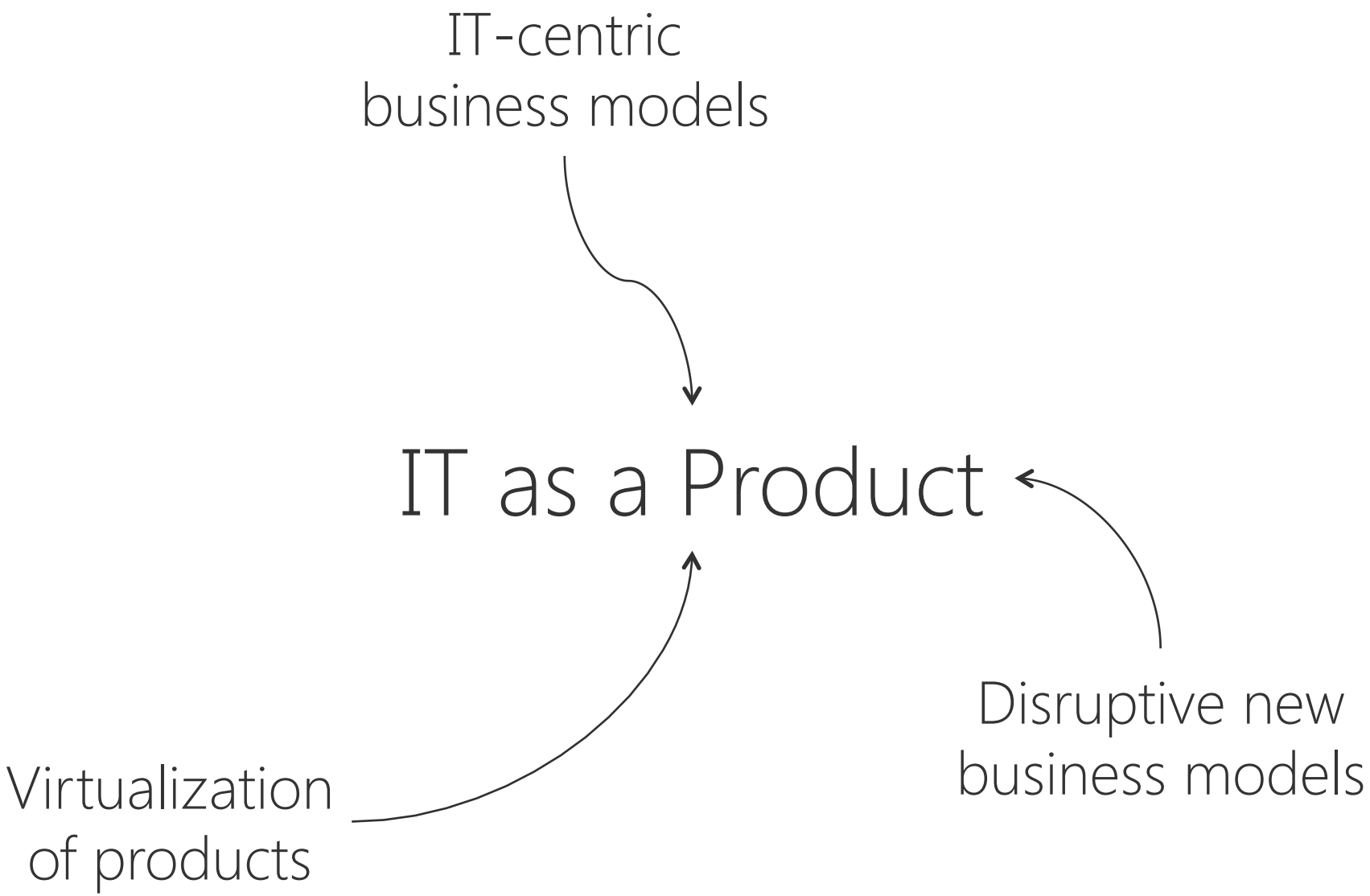
But there is more …


Innovation
Measure & analyze
Accelerating OODA loop
Lean Enterprise
Product shaping/optimization
Quick customer feedback cycles

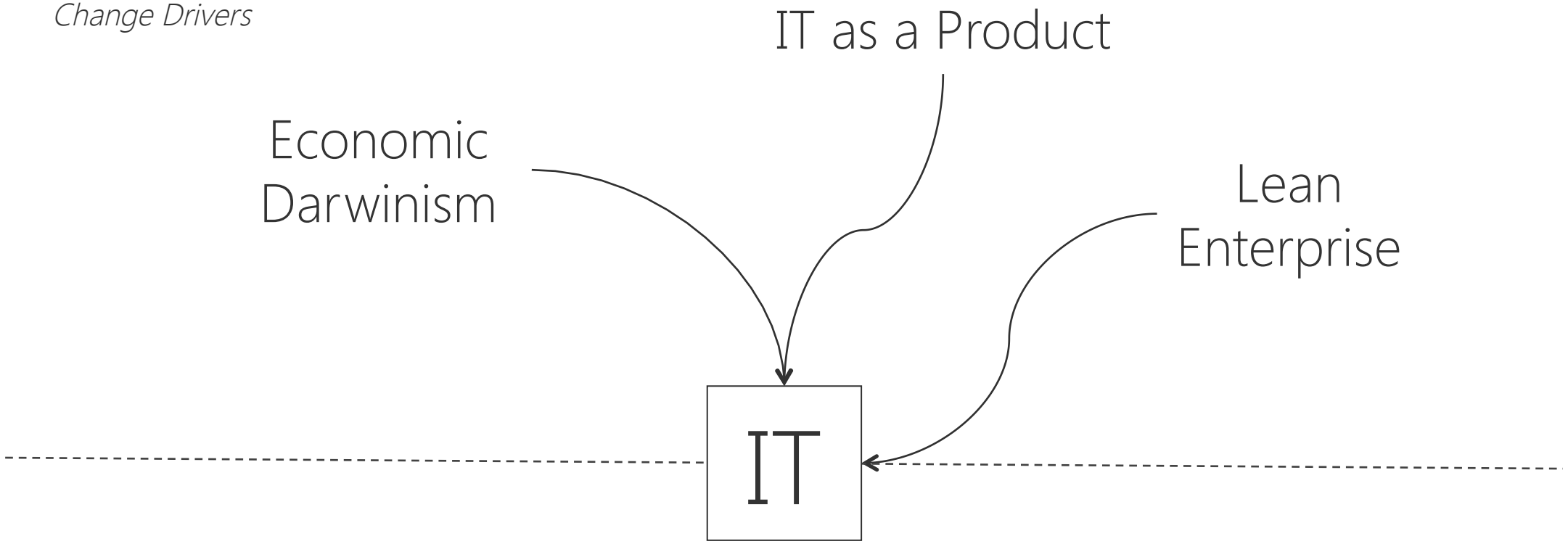
Business-related
Change Drivers



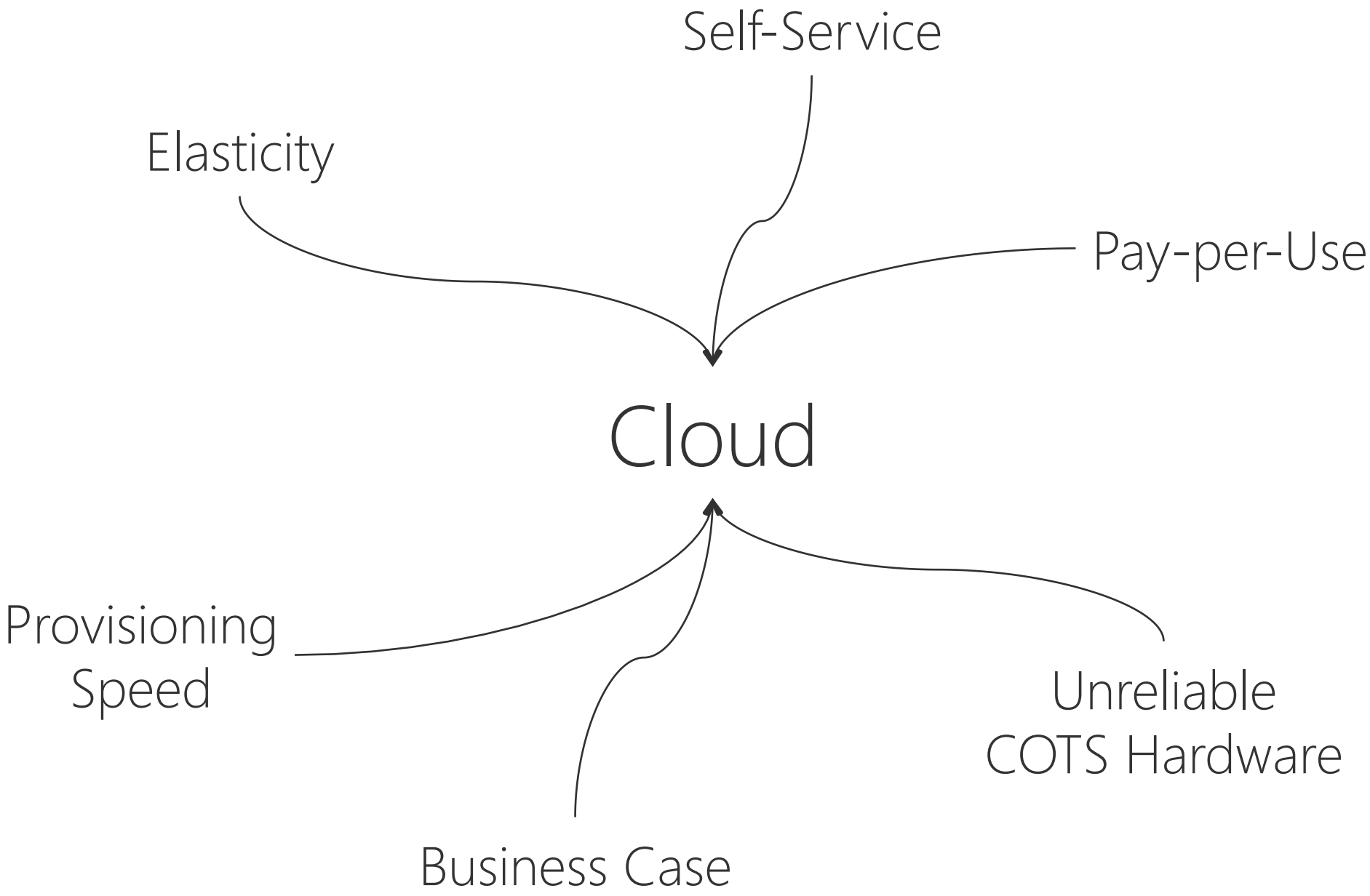
Technology-related
Change Drivers


IT-centric
business models
IT as a Product
Virtualization
of products
Disruptive new
business models

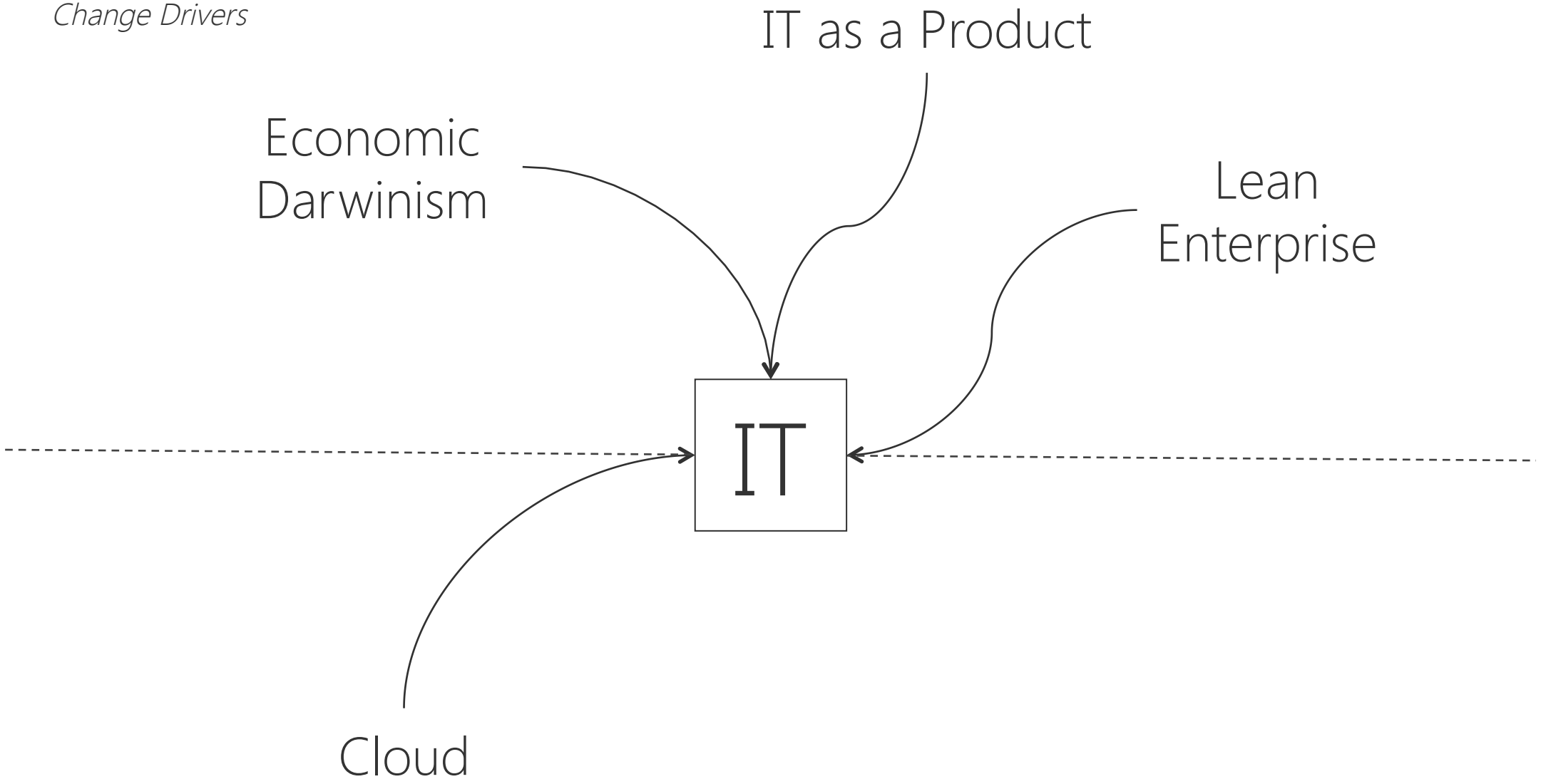
Business-related Change Drivers



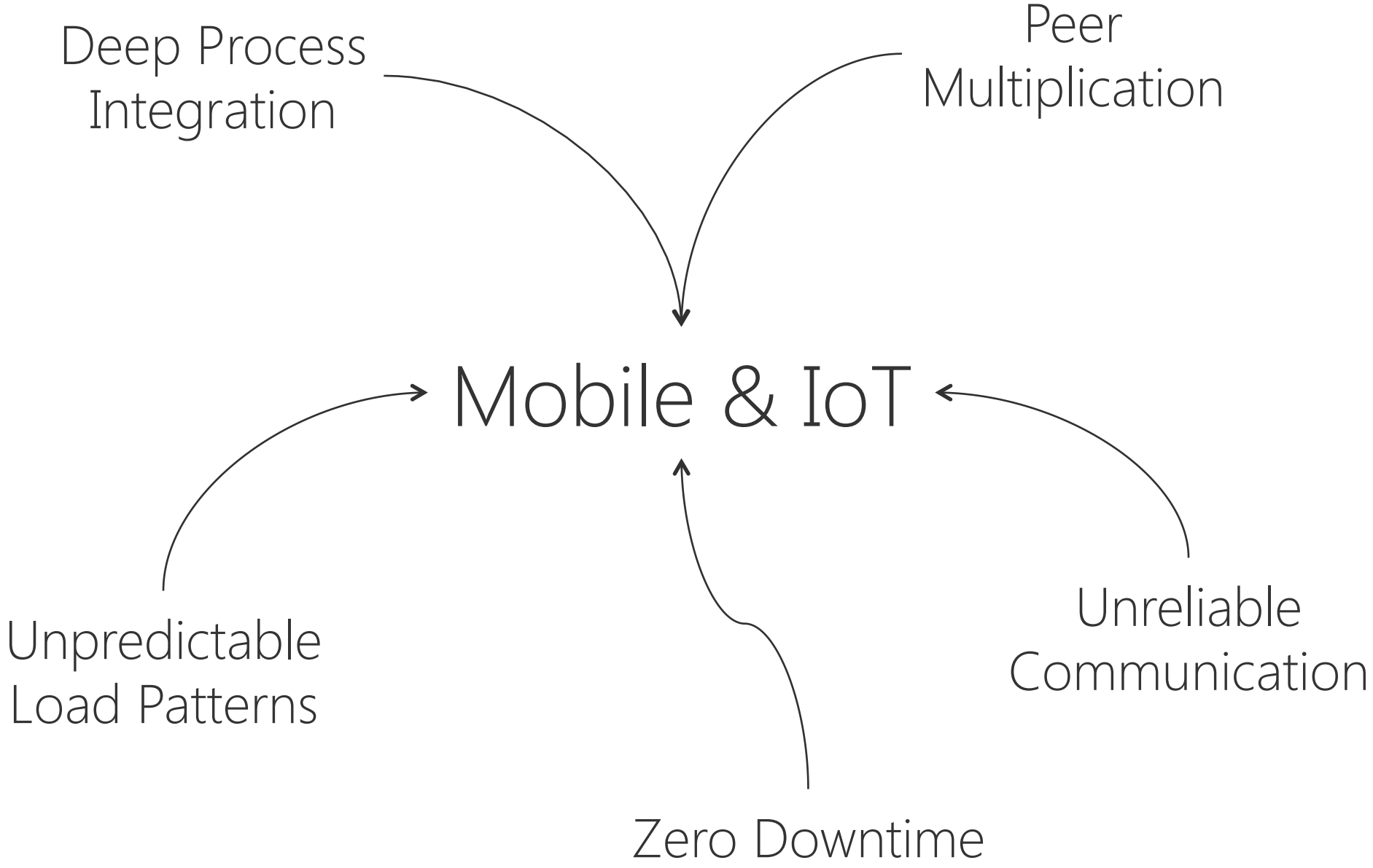
Technology-related Change Drivers

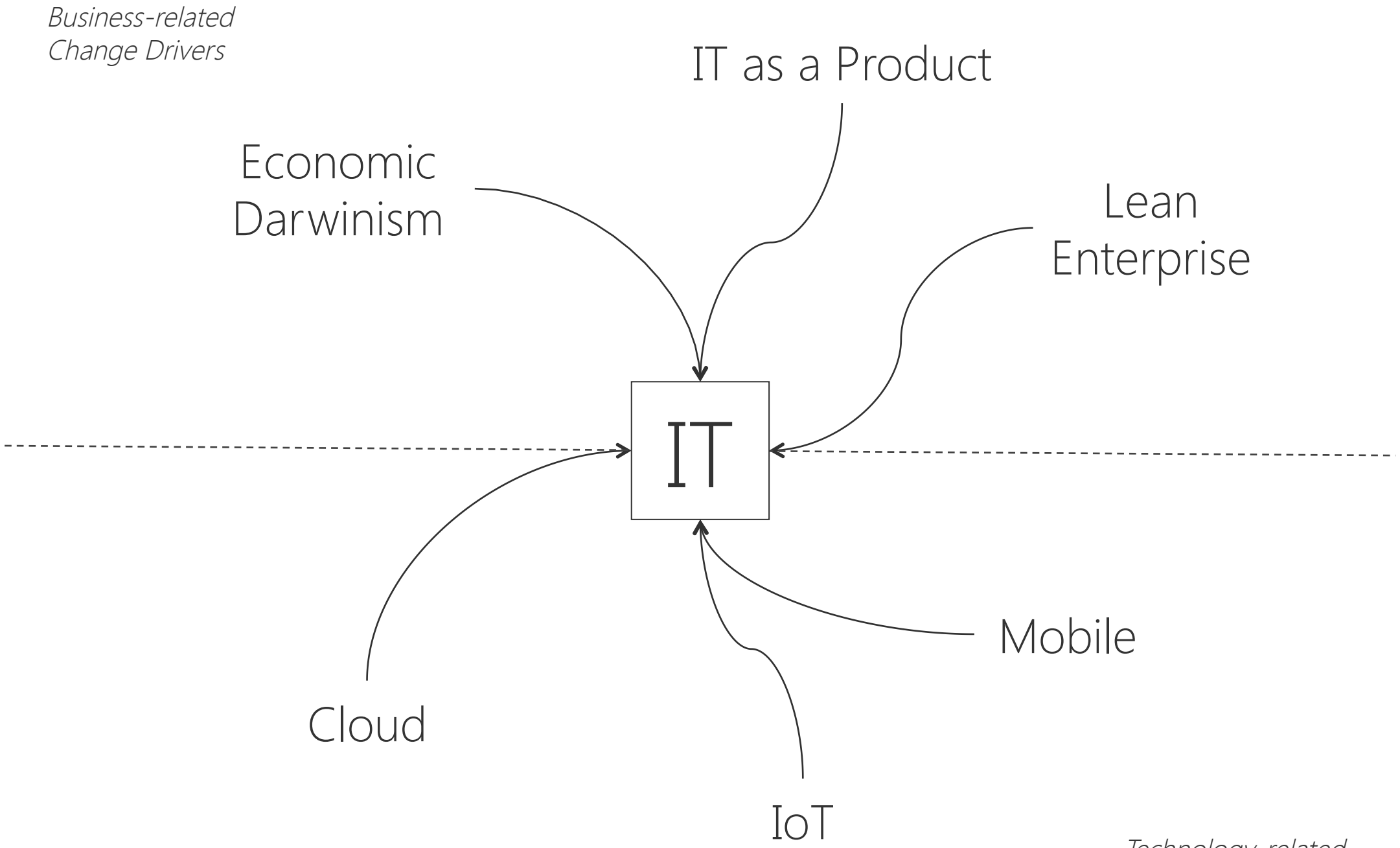

Self-Service
Elasticity
Pay-per-Use
Cloud
Provisioning
Speed
Unreliable
COTS Hardware
Business Case

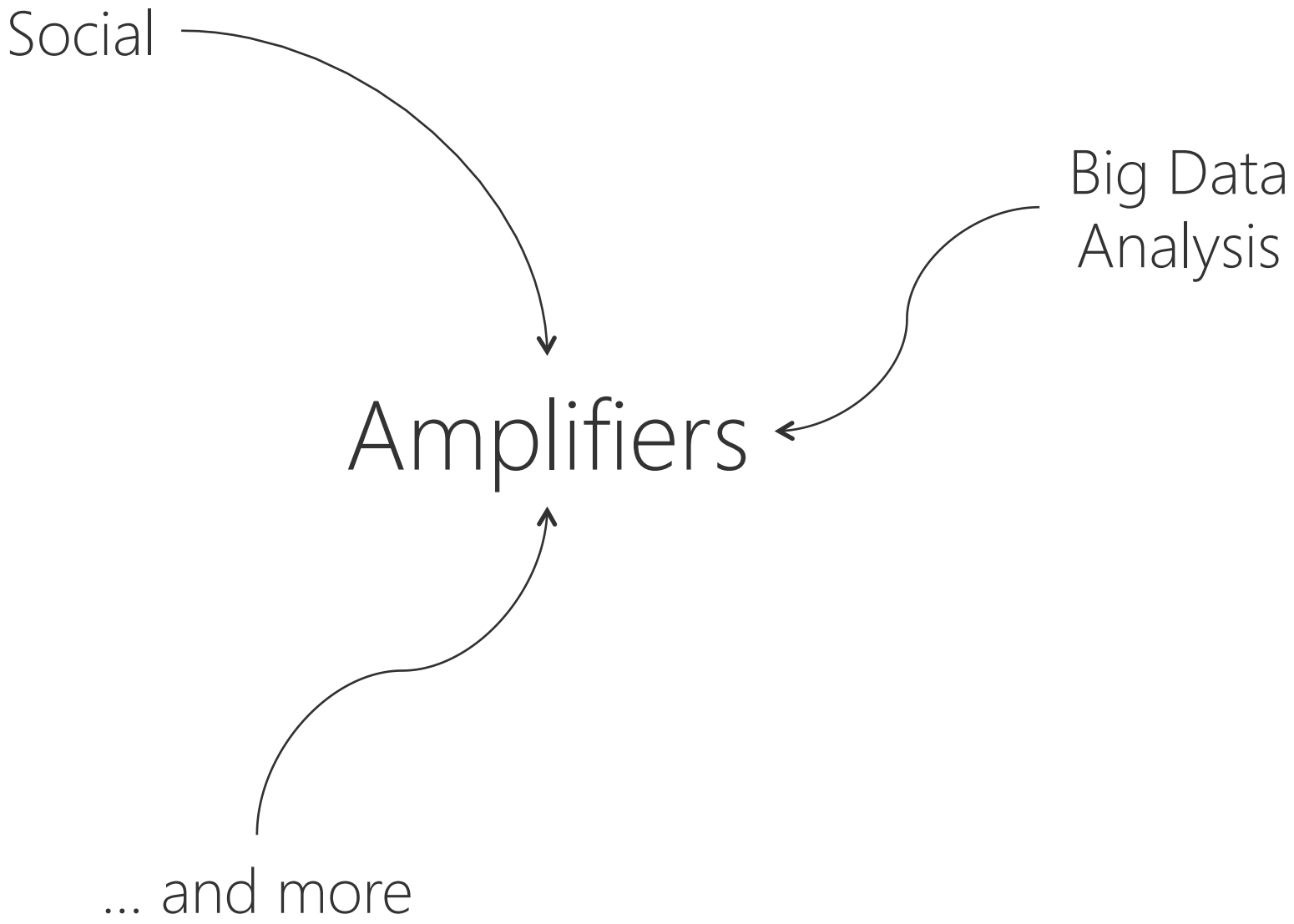
Business-related Change Drivers

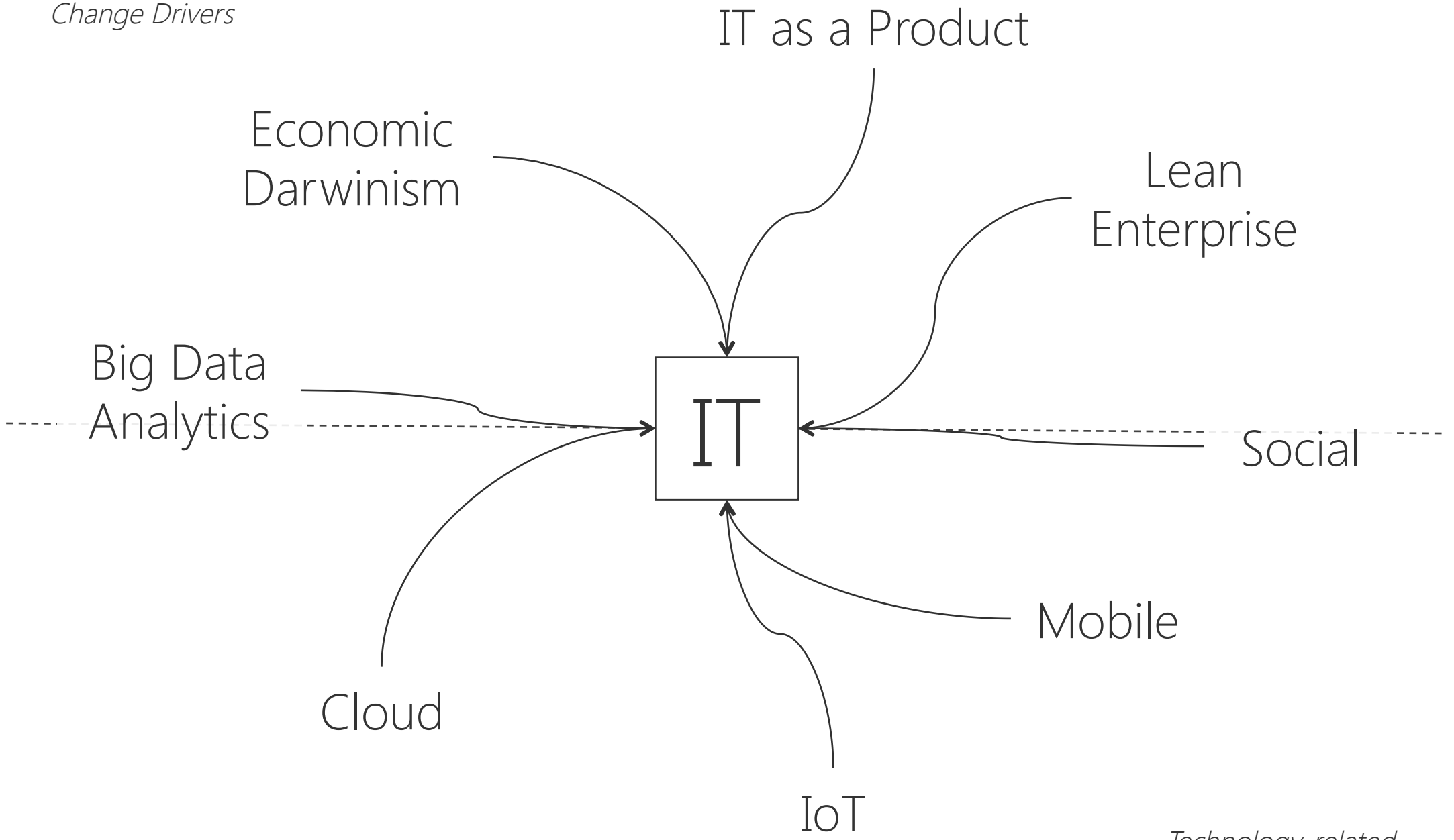


Technology-related Change Drivers


Deep Process Integration
Peer Multiplication
Mobile & IoT
Unpredictable Load Patterns
Zero Downtime
Unreliable Communication


Business-related
Change Drivers
IT as a Product
Economic
Darwinism
Lean
Enterprise
IT
Mobile
Cloud
IoT
Technology-related
Change Drivers


Social
Big Data
Analysis
Amplifiers
... and more


Business-related
Change Drivers
IT as a Product
Economic
Darwinism
Lean
Enterprise
Big Data
Analytics
IT
Social
Cloud
Mobile
IoT
Technology-related
Change Drivers

Why does traditional IT usually fail to respond to those challenges?

Traditional IT bases its optimization efforts on the wrong goals and principles

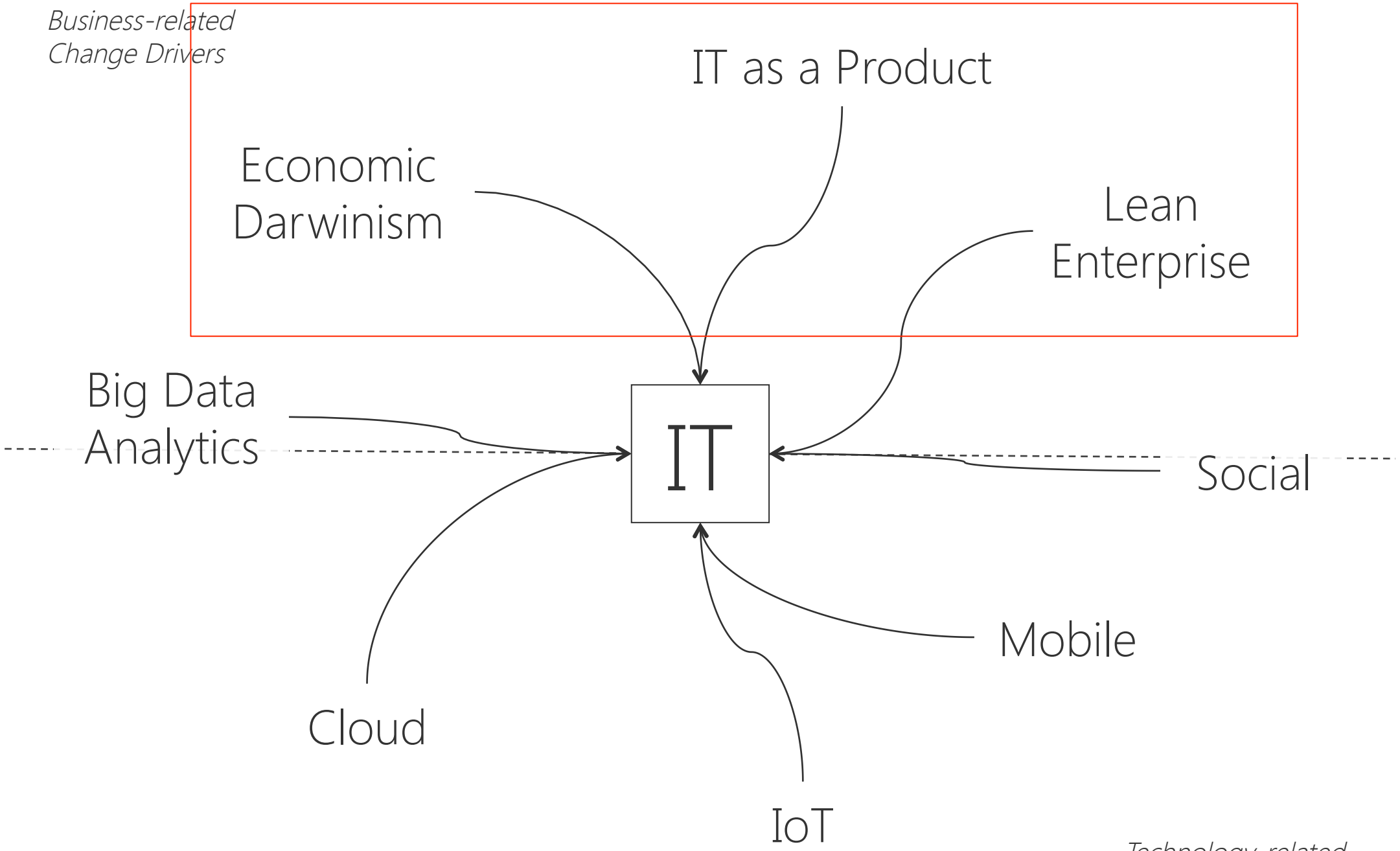
Traditional IT goals/principles

- Fault avoidance at any cost
  a.k.a. "the root of all evil"

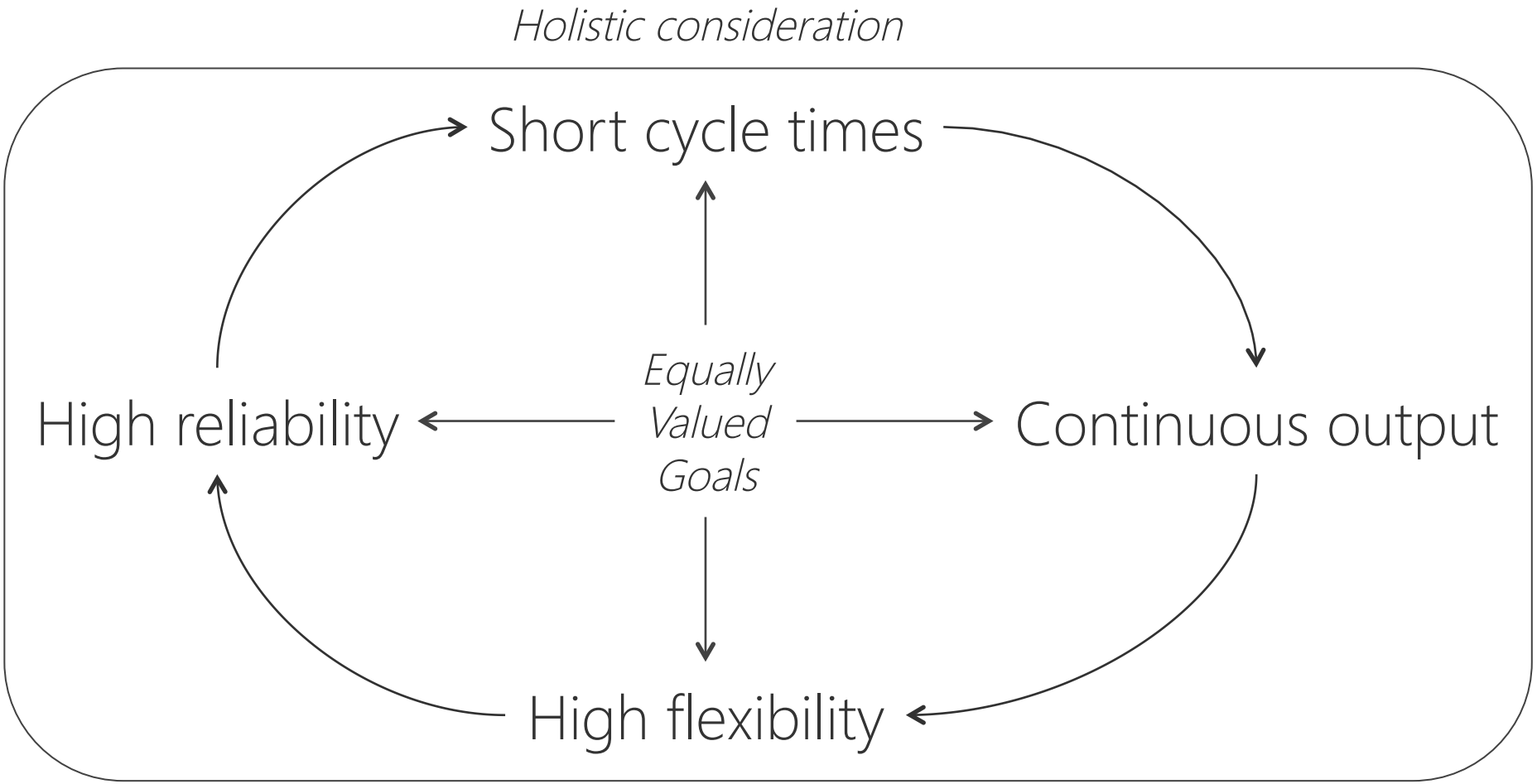
- Tayloristic organization
- Local optimization
- Process frenzy
- Central control
- Long-running projects
- Standardization
- Cost minimization

→ Not suitable to respond to new challenges

Then, what are the new goals?


Business-related
Change Drivers
IT as a Product
Economic
Darwinism
Lean
Enterprise
Big Data
Analytics
IT
Social
Cloud
Mobile
IoT
Technology-related
Change Drivers

Goals of a Next Generation (of) IT



And what are the new principles?

Principles of a Next Generation (of) IT

The Core Principles

Maximizing innovation instead of minimizing costs

Controlled experiments instead of fault avoidance at any cost

Decentralized, self dependent teams instead of central control and goal sheets

Flexible adaption instead of static planning

Accepting complexity on all levels

Based on Jeff Sussna's 21st Century IT Manifesto (http://blog.ingineering.it/post/39385342347/21st-century-it-manifesto)
Refined in collaboration with Eberhard Wolff

Principles of a Next Generation (of) IT

The Technical Principles

Diversity & lightweight tools instead of monoculture & integrated solutions

Resilience instead of stability

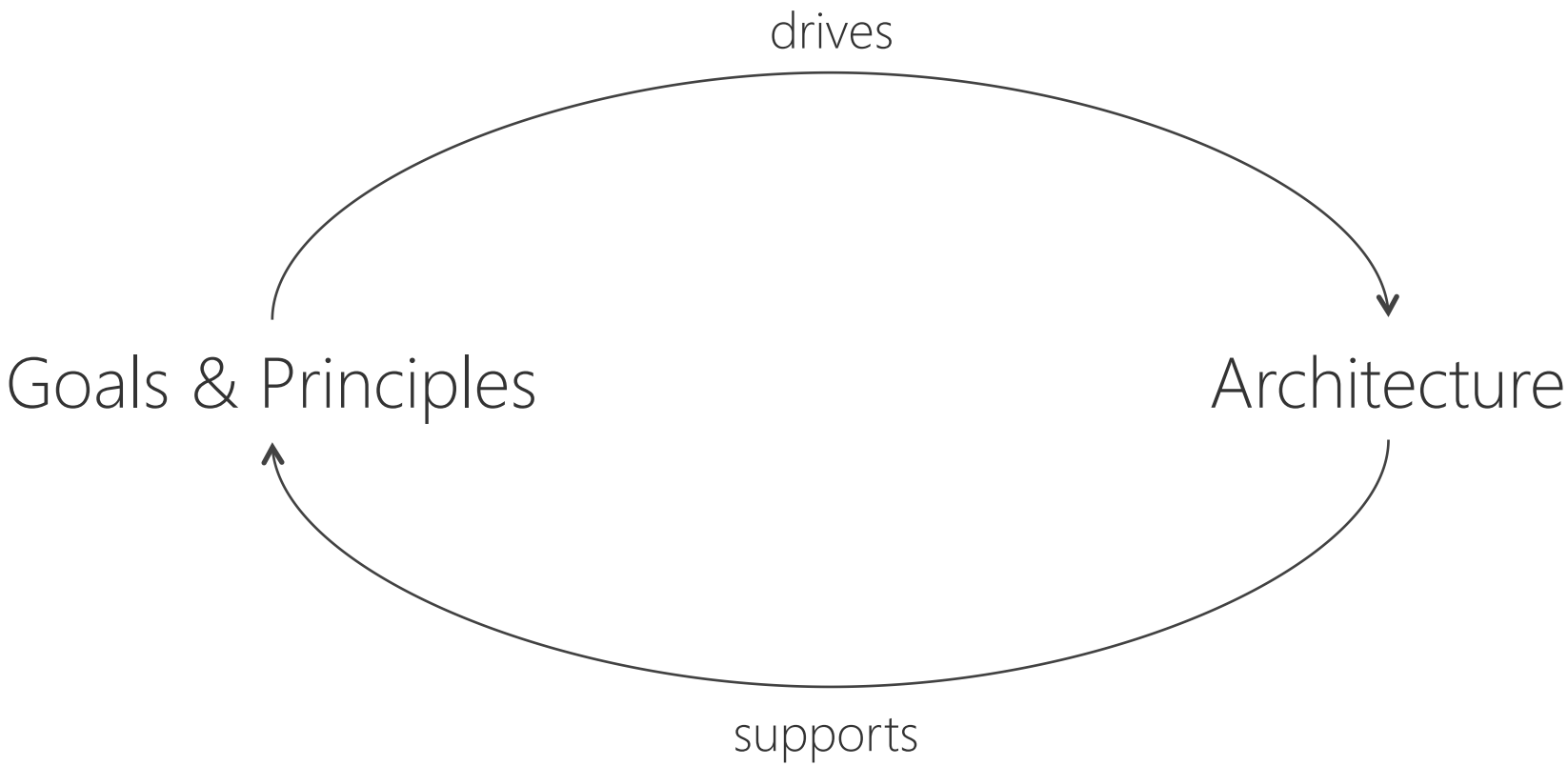
(µ)Services instead of monoliths

Elasticity instead of upfront capacity planning

Consistent automation of routine tasks

Based on Jeff Sussna's 21st Century IT Manifesto (http://blog.ingineering.it/post/39385342347/21st-century-it-manifesto)
Refined in collaboration with Eberhard Wolff

Nice (again), but how does this relate to architecture?


drives
Goals & Principles
Architecture
supports

What does that mean for architecture?

Architectural drivers

- Need for quick change and extension
- Replace over reuse
- Need for quick releases

- Unpredictable load patterns
- Distributed, highly interconnected systems
- Extreme high service availability

- Diverse front-ends and devices

- Cost efficiency

Architectural requirements

- Easy to understand
- Easy to extend
- Easy to change
- Easy to replace
- Easy to deploy

- Easy to scale
- Easy to recover

- Easy to connect

- Easy to afford

Architectural requirements

- Easy to understand → Understandability
- Easy to extend → Extensibility
- Easy to change → Changeability
- Easy to replace → Replaceability
- Easy to deploy → Deployability

- Easy to scale → Scalability
- Easy to recover → Resilience

- Easy to connect → Uniform interface

- Easy to afford → Cost-efficiency
  (for development & operations)

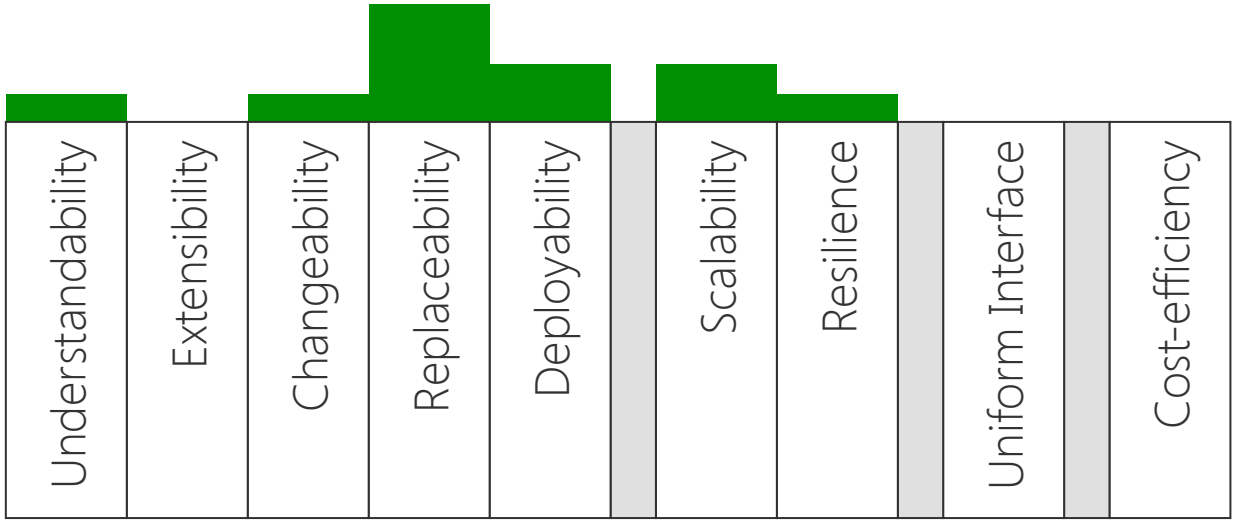
What are the appropriate solutions?

Let’s check a few hype topics …

µServices

- Built for replacement (not reuse)
- Self-dependent, loosely coupled services
- Should be aligned with business capability
- Size should not exceed what one brain can grasp

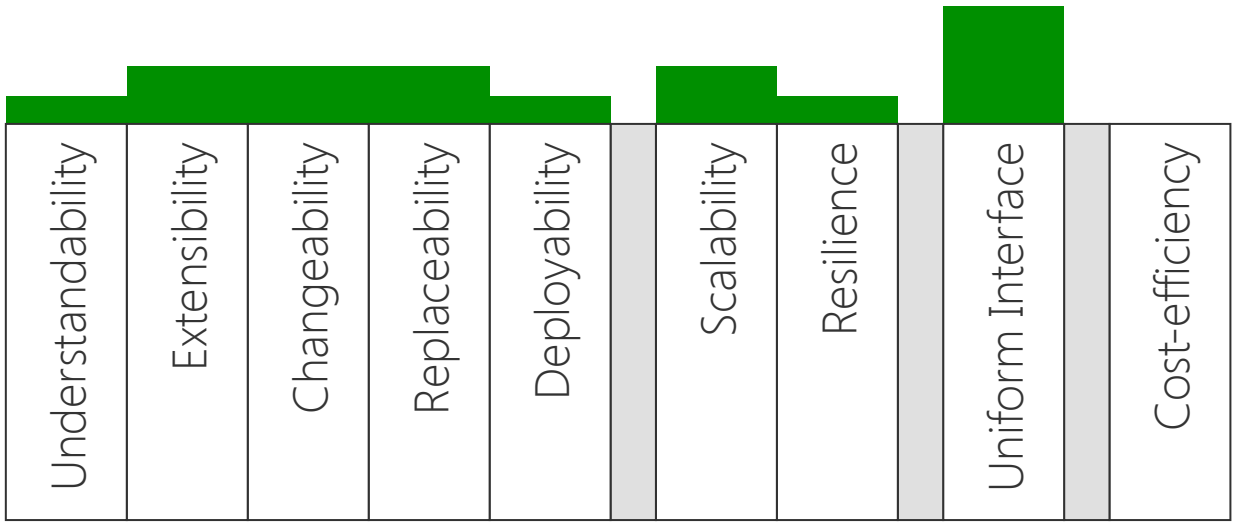
µServices


Understandability
Extensibility
Changeability
Replaceability
Deployability
Scalability
Resilience
Uniform Interface
Cost-efficiency

REST

- Uniform access interface to resources
- Closely related to the HTTP protocol
- HATEOAS (Hypermedia as the engine of application state)

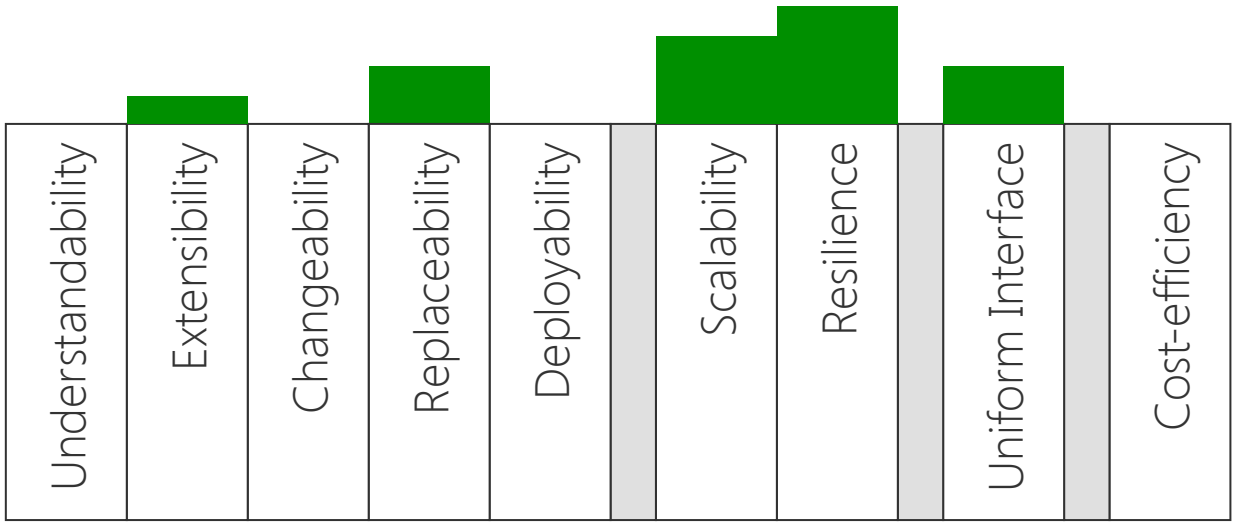
REST


Understandability
Extensibility
Changeability
Replaceability
Deployability
Scalability
Resilience
Uniform Interface
Cost-efficiency

Event-driven

- Asynchronous communication paradigm
- Technical decoupling of communication peers (isolation)
- Location transparency in conjunction with MOM
- Call-stack paradigm replaced by (complex) message networks

Event-driven

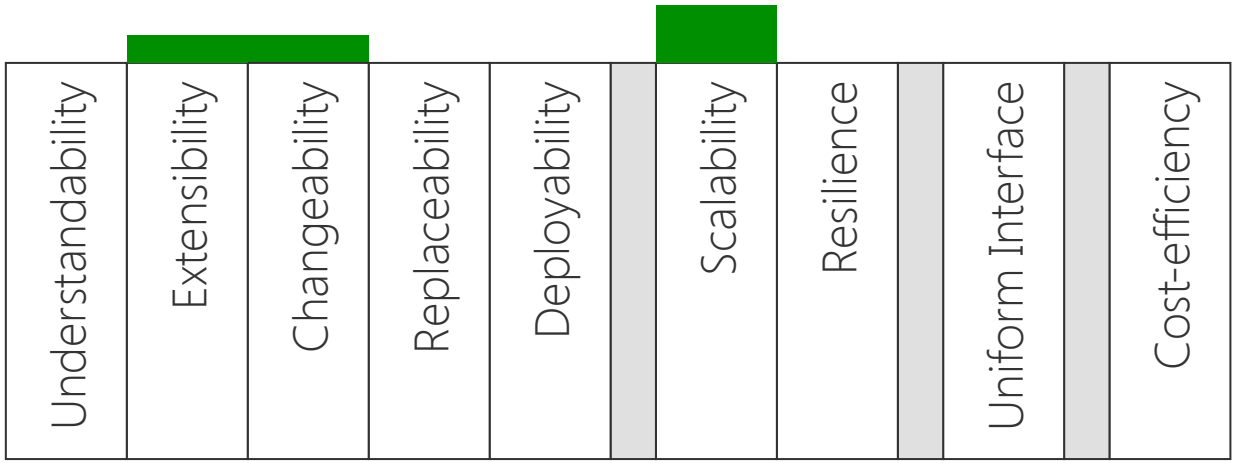


CQRS

- Command Query Responsibility Segregation
- Separate read and write interfaces including underlying models
- Separation can be extended up to the data store(s)
- Allows for optimized data representations and access logic

CQRS

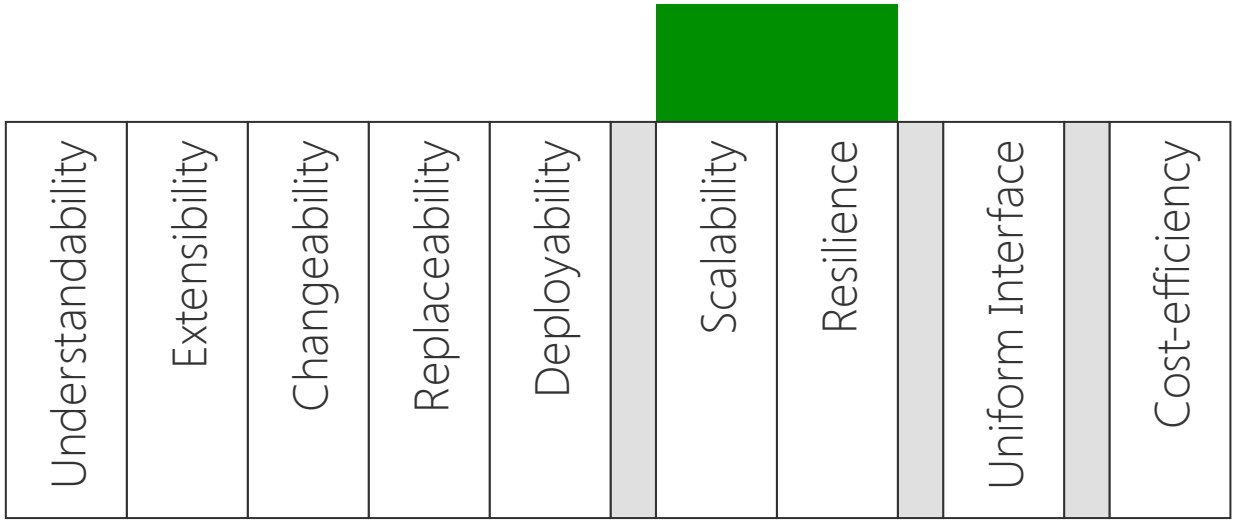

READ
WRITE


Understandability
Extensibility
Changeability
Replaceability
Deployability
Scalability
Resilience
Uniform Interface
Cost-efficiency

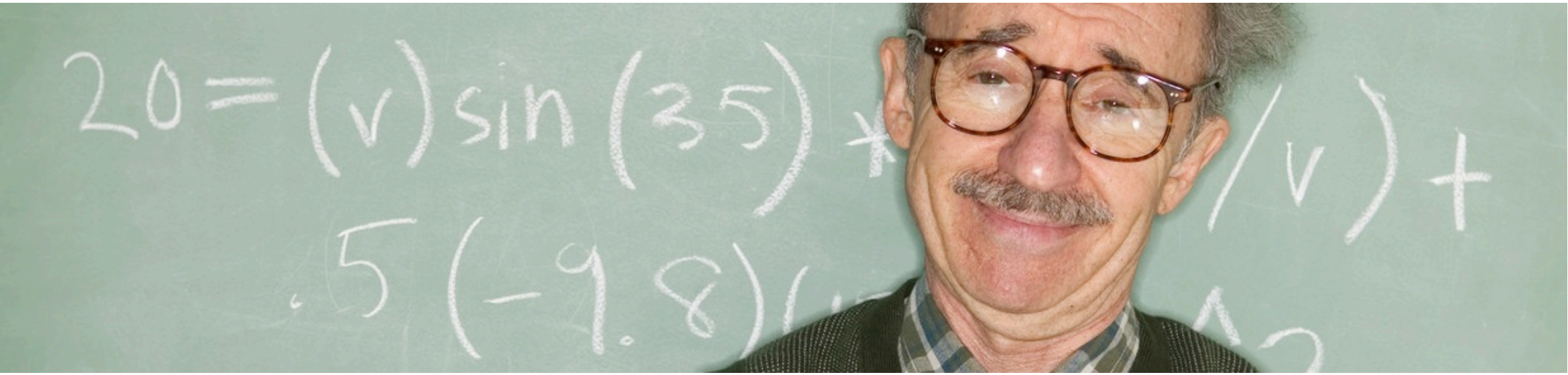
Reactive

- Event-driven – asynchronous and non-blocking
- Scalable – scaling out and embracing the network
- Resilient – isolation, loose coupling and hierarchical structure
- Responsive – latency control and graceful degradation of service

Reactive

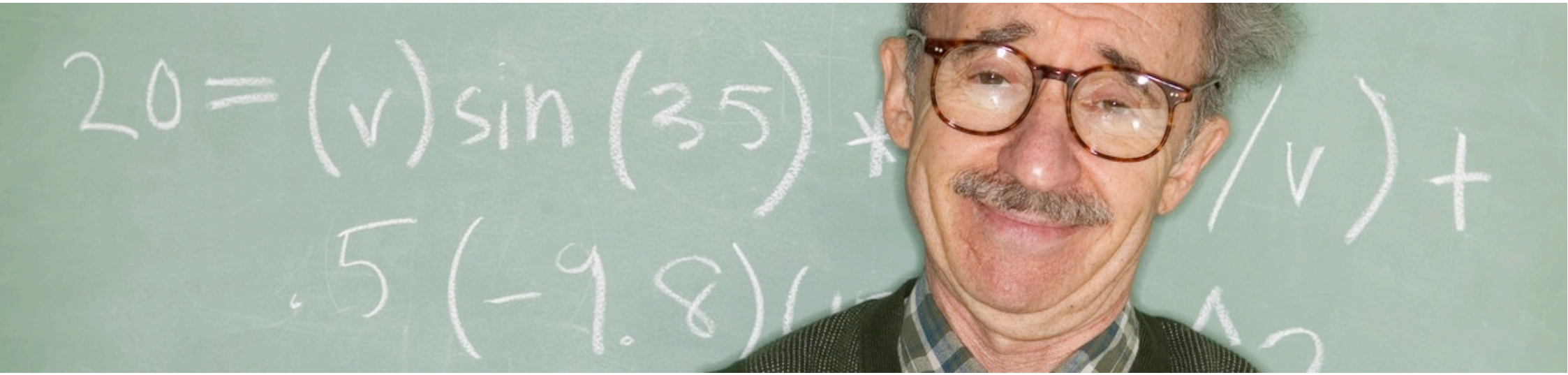

Understandability
Extensibility
Changeability
Replaceability
Deployability
Scalability
Resilience
Uniform Interface
Cost-efficiency

Functional Programming



- Alternative programming paradigm
- Functional languages (Erlang, Haskell, Clojure, …)
- Hybrid languages (Scala, …)
- Languages with functional extensions (Python, JavaScript, Java, …)

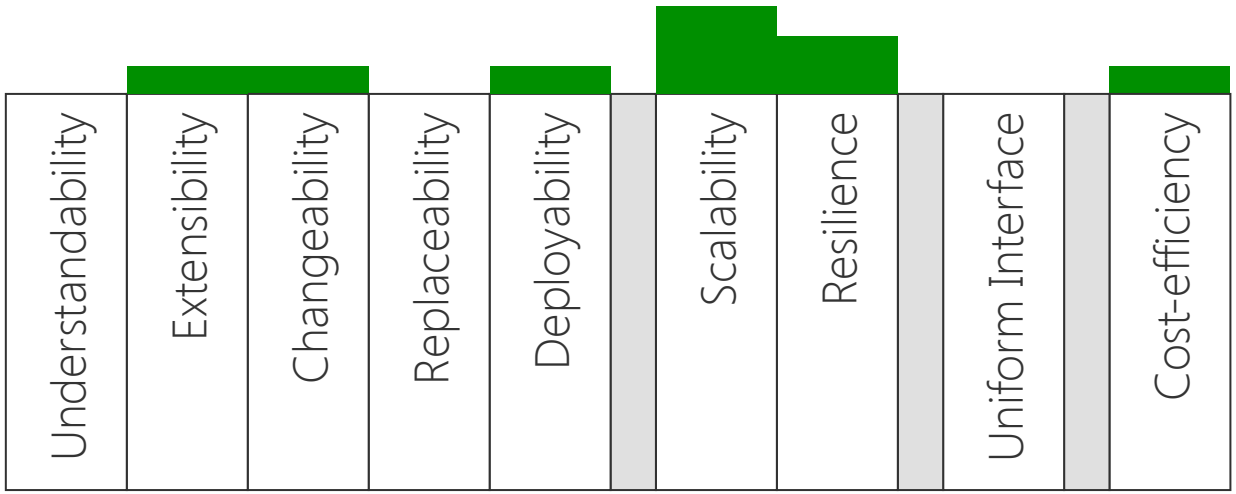
Functional Programming



NoSQL

- Augments the data store solution space
- Different sweet spots than RDBMS
- Key-Value Store – Wide Column Store – Document Store
- Graph Database

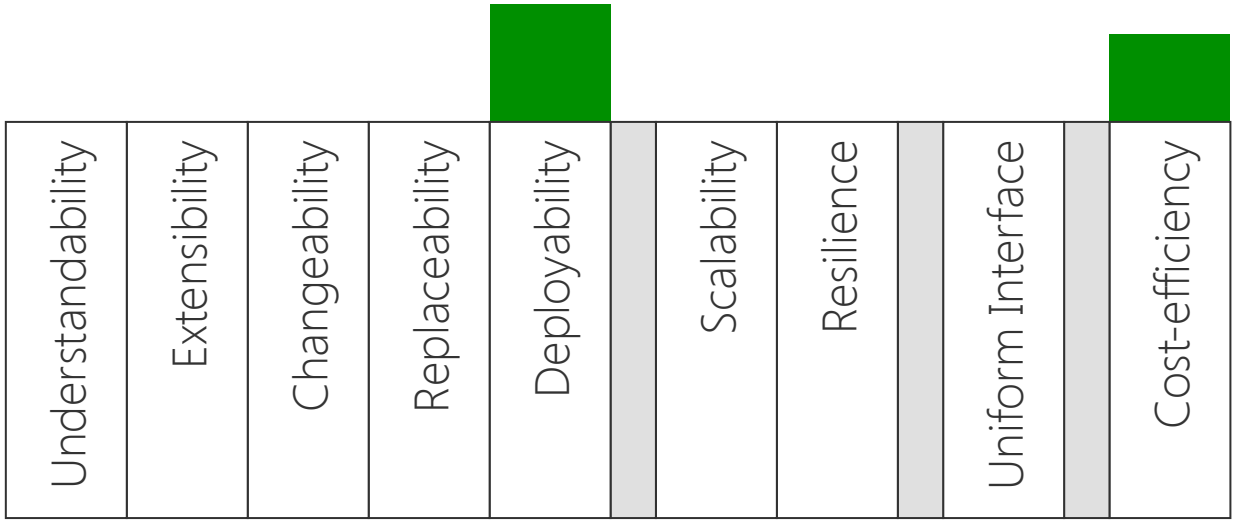
NoSQL


Understandability
Extensibility
Changeability
Replaceability
Deployability
Scalability
Resilience
Uniform Interface
Cost-efficiency

Continuous Delivery

- Automate the software delivery chain
- Build – Continuous Integration, …
- Test – Test Automation, …
- Deploy – Infrastructure as Code, …

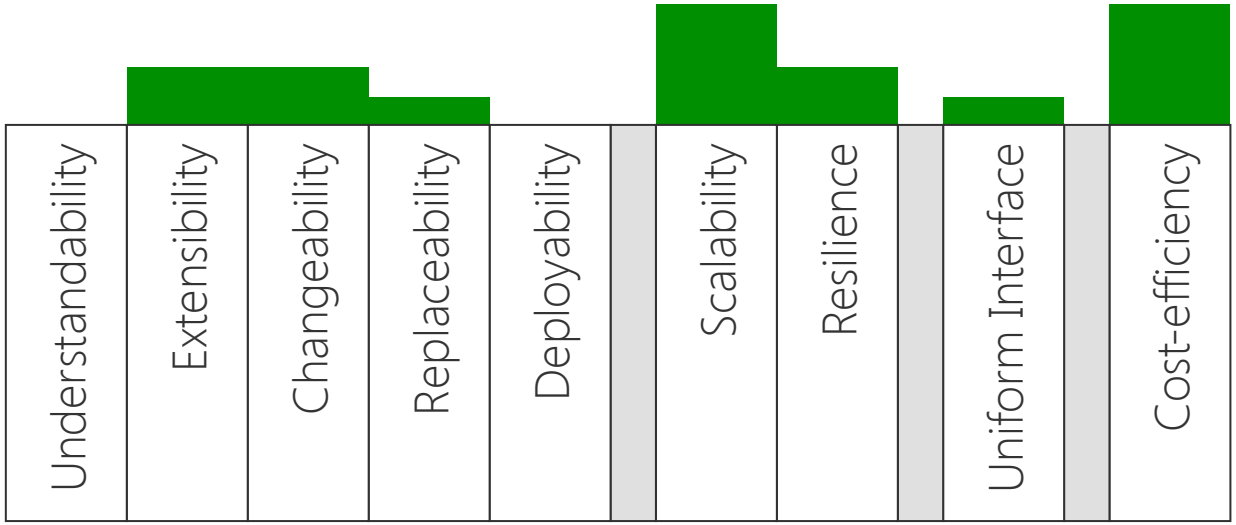
Continuous Delivery



Cloud provisioning model

- On-demand provisioning and de-provisioning
- Instant availability
- Self-service
- Pay-per-use

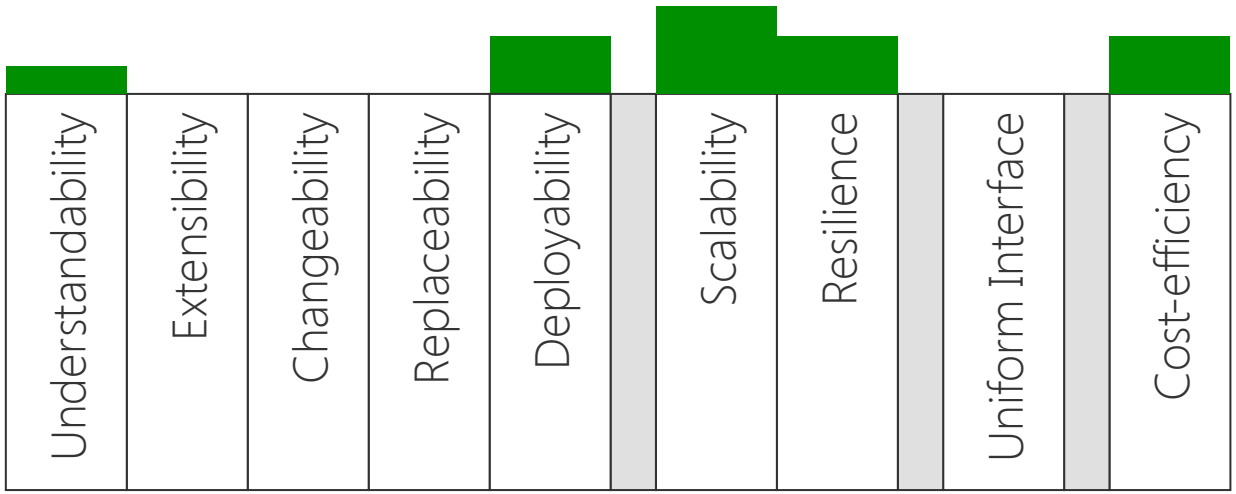
Cloud provisioning model



Docker

- Build, ship, run on container-basis
- Process-level isolation
- Declarative communication path configuration
- Cambrian explosion of ecosystem at the moment

Docker


Understandability
Extensibility
Changeability
Replaceability
Deployability
Scalability
Resilience
Uniform Interface
Cost-efficiency

... and there are many more

What can we learn from this?

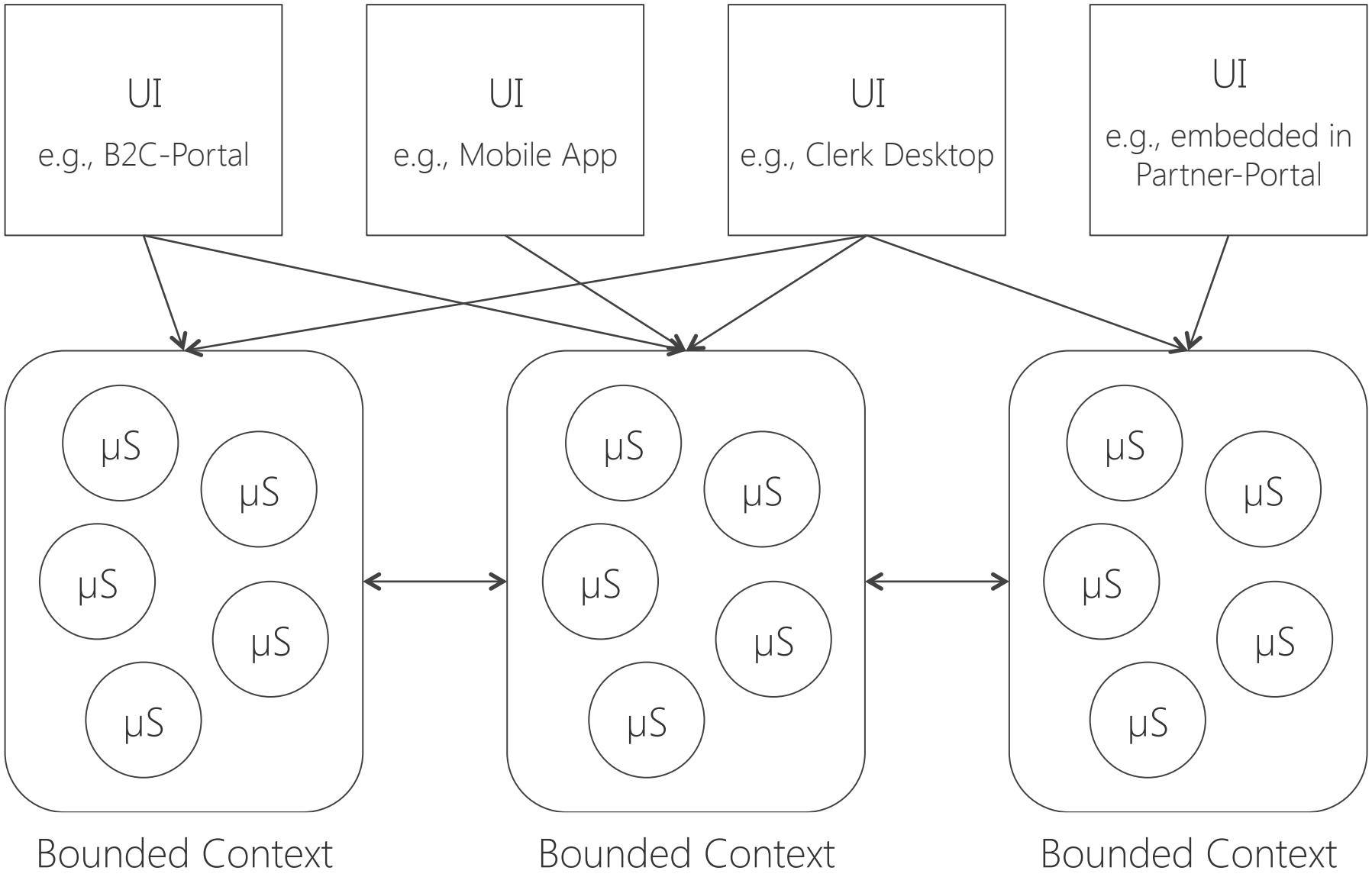
Findings

- There is not a simple solution and no “one size fits all”
- Some of the topics evaluated have a high potential
- Some of the topics evaluated do not help so much
- A combination of several approaches is needed

How would an architectural style look like?

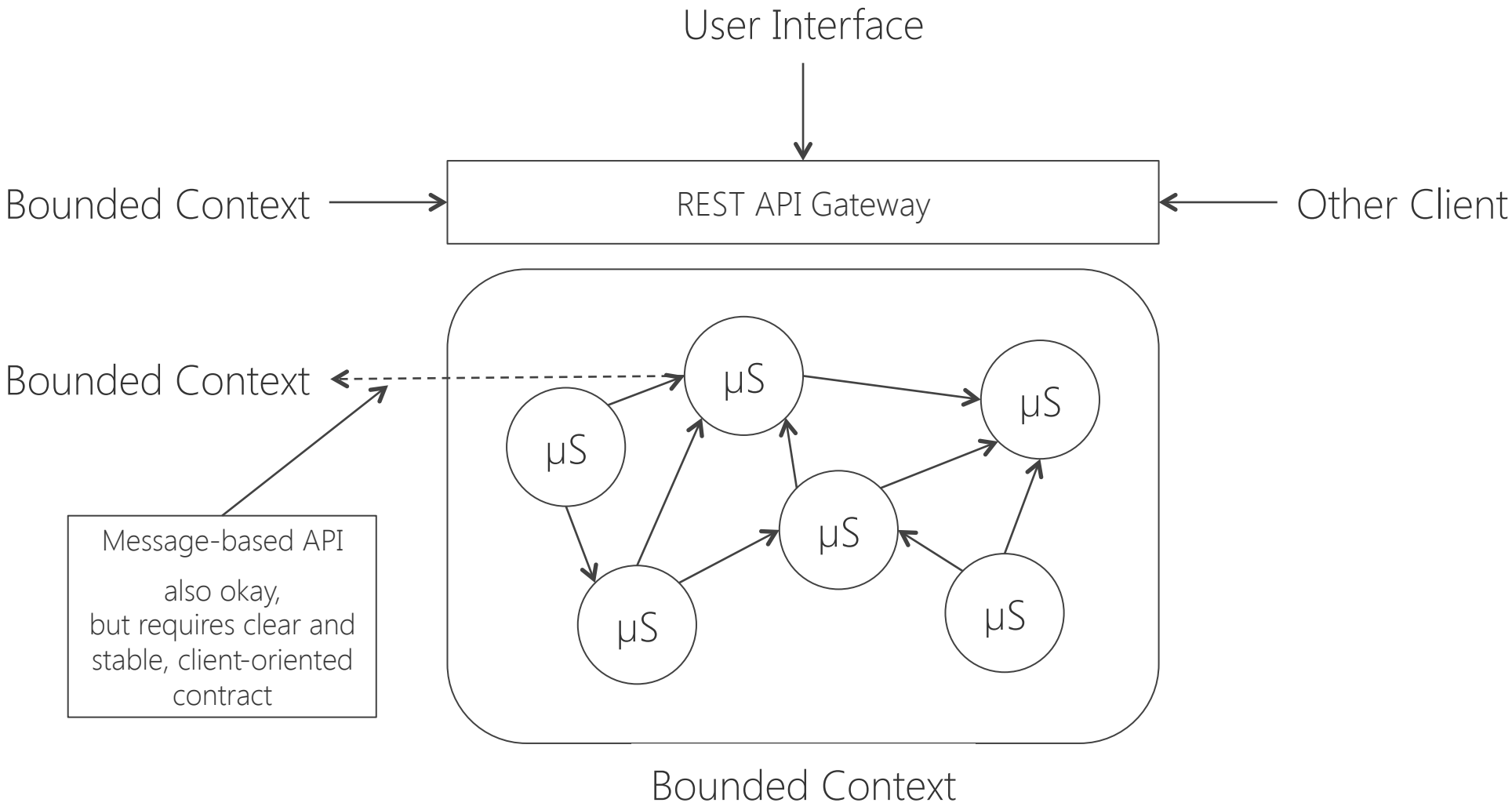
µServices

- Conway's law
- Built for replacement
- Aligned with business capabilities
- Bounded Context (Domain-Driven Design)
- Separate UI and service


UI
e.g., B2C-Portal
UI
e.g., Mobile App
UI
e.g., Clerk Desktop
UI
e.g., embedded in Partner-Portal
µS
µS
µS
µS
µS
µS
µS
µS
µS
µS
µS
µS
µS
µS
µS
Bounded Context
Bounded Context
Bounded Context

REST interfaces

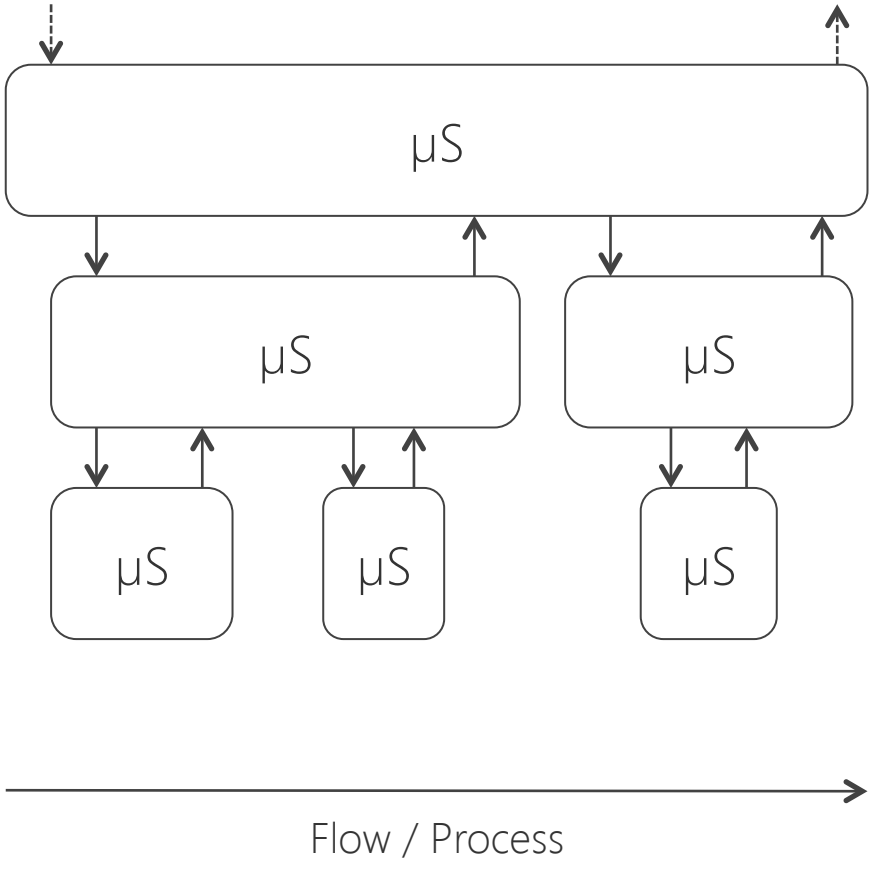
- Use as API gateway for client access
- Encapsulate dynamics and complexity of service landscape
- Provide client-driven, coarse-grained service calls behind a uniform API based on a proven protocol
- Should be provided on bounded context level
- Decouple speed of evolvement (services vs. API)


User Interface
Bounded Context
REST API Gateway
Other Client
Bounded Context
μS
μS
μS
μS
μS
μS
Message-based API
also okay,
but requires clear and
stable, client-oriented
contract
Bounded Context

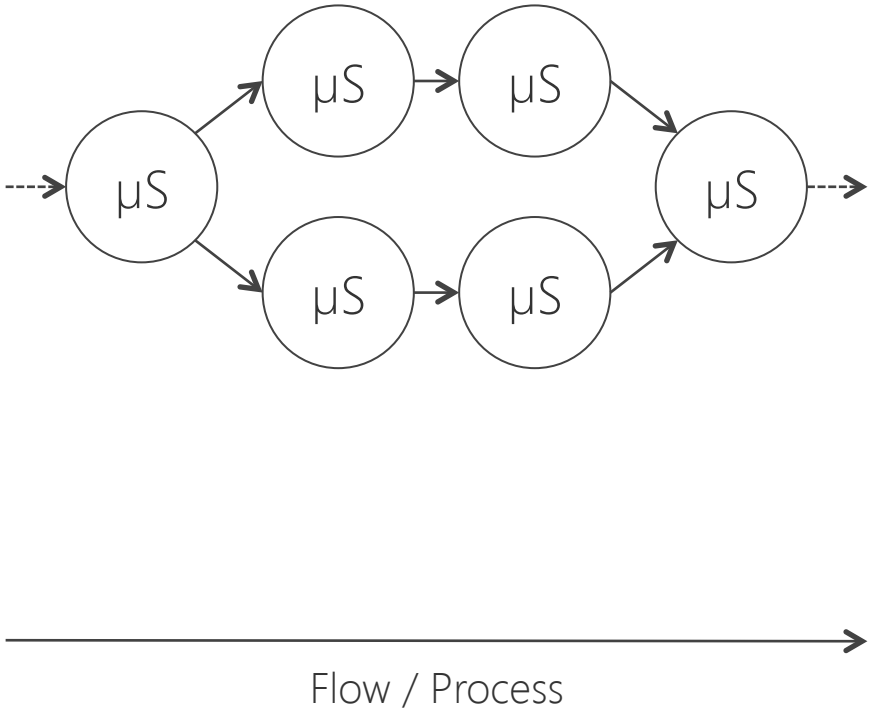
Event-driven communication

- Use for inter-service communication
- Decoupling and isolation
- Vertical slicing of functionality
- Easier evolution of flows and processes
- Configuration-visualization-monitoring support required

Request/Response : Horizontal slicing

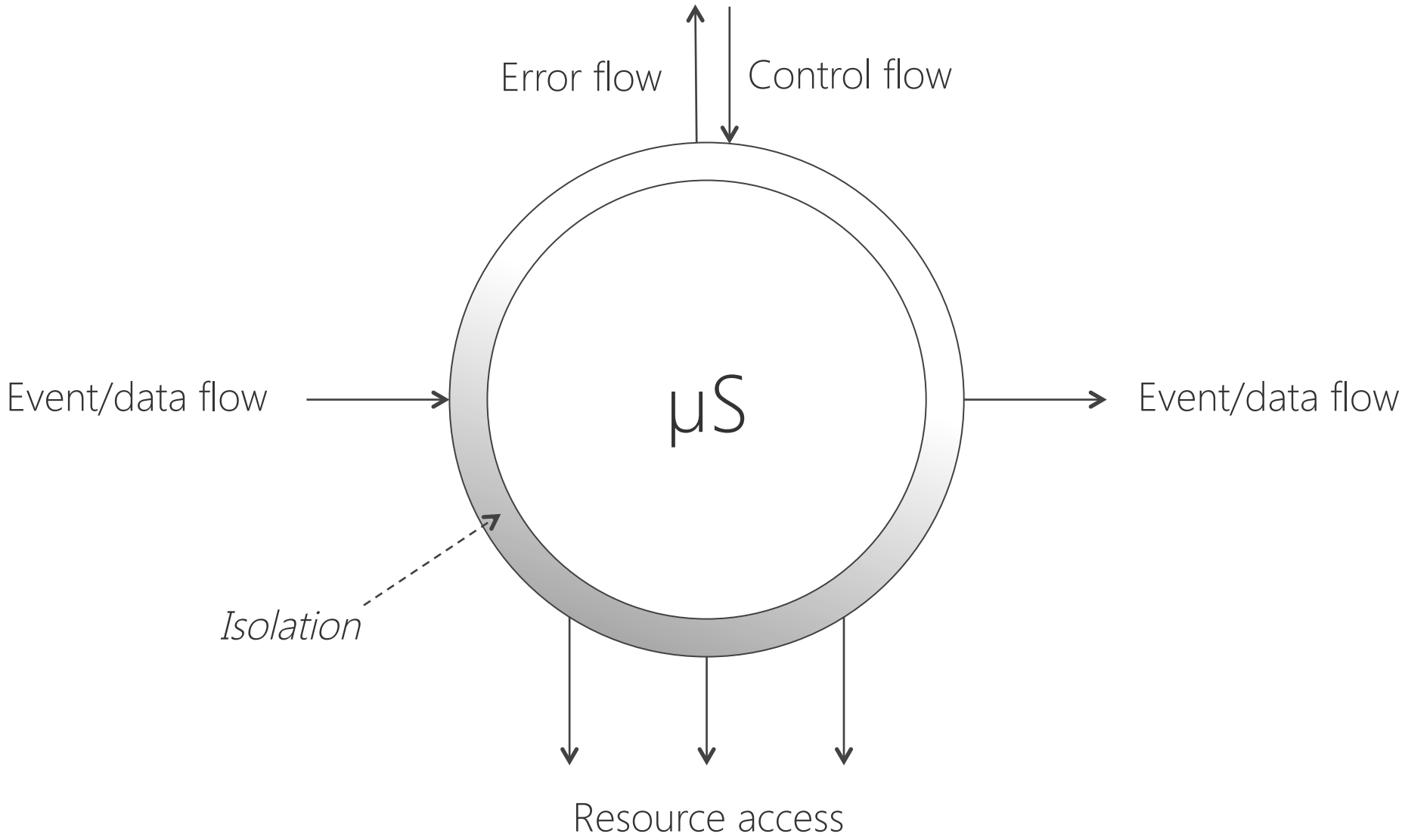


Event-driven : Vertical slicing



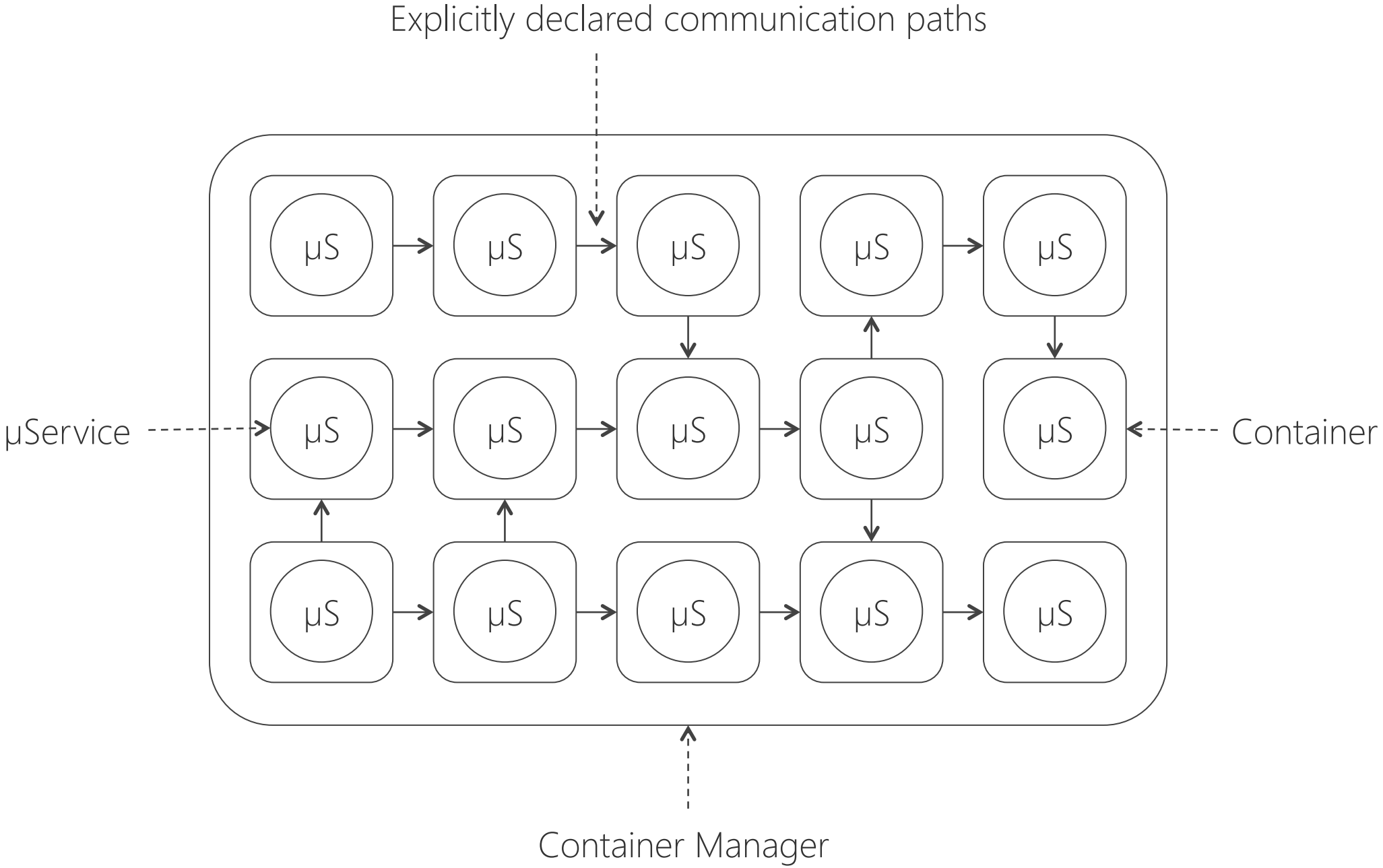
Resilient/reactive design

- Resilience and responsiveness are mandatory
- Elastic design for scalability
- Start with isolation and latency control
- Separate control and data flow
- Many new challenges for developers


Error flow
Control flow
Event/data flow
μS
Event/data flow
Isolation
Resource access

Cloud provisioning model

- Basis for elasticity at runtime
- Basis for speed and flexibility at development time
- Private, hybrid or public
- Should be combined with container approaches (e.g., Docker)
- “Natural” infrastructure for µService architecture


Explicitly declared communication paths
µS
µService
Container
Container Manager

Automate

- Automate everything
- Build, test & deployment (Continuous Delivery)
- Resource provisioning (Cloud API)
- Restart, failover, error handling (Resilience)
- Starting and tearing down instances (Scalability)

Wrap-up

- IT is the nervous system of a company
- Delivery speed is the new benchmark
- Architecture must support the drivers
- The new architectures are *different*
- New challenges for developers (& ops)

It’s the most disruptive and exciting change we have seen in IT for many years

Join the IT revolution!

@ufried

Uwe Friedrichsen | uwe.friedrichsen@codecentric.de | http://slideshare.net/ufried | http://ufried.tumblr.com