Microservices

Where Did It All Go Wrong

Ian Cooper @ICooper (X, Hachyderm.io)

Who are you?

I am a polyglot coding architect with over 20 years of experience delivering solutions in government, healthcare, and finance and ecommerce. During that time I have worked for the DTI, Reuters, Sungard, Misys, Beazley, Huddle and Just Eat Takeaway delivering everything from bespoke enterprise solutions, 'shrink-wrapped' products for thousands of customers, to SaaS applications for hundreds of thousands of customers.

I am an experienced systems architect with a strong knowledge of OO, TDD/BDD, DDD, EDA, CQRS/ES, REST, Messaging, Design Patterns, Architectural Styles, ATAM, and Agile Engineering Practices

I am frequent contributor to OSS, and I am the owner of: https://github.com/BrighterCommand. I speak regularly at user groups and conferences around the world on architecture and software craftsmanship. I run public workshops teaching messaging, event-driven and reactive architectures.

I have a strong background in C#. I spent years in the C++ trenches. I dabble in Go, Java, JavaScript and Python.

www.linkedin.com/in/ian-cooper-2b059b

BRIGHTER

Agenda

- Some Definitions
- The Problem
- Advice
- Q&A

Some Definitions

When I use a word,' Humpty Dumpty said in rather a scornful tone, 'it means just what I choose it to mean — neither more nor less.

— Lewis Carroll, Through the Looking Glass

Fundamental Theorem of Software Engineering

$$C(P) > (\frac{1}{2}P) + C(\frac{1}{2}P)$$

Due to complexity: It is always easier (and cheaper) to create two small pieces rather than one big piece if the two small pieces do the same job as the single piece. This is because combining two problems in one piece forces us to deal with the unneeded interactions between the two pieces

Yourdon, Edward; Constantine, Larry Structured Design, 1975

Fundamental Theorem of Software Engineering

Specifically, we suggest that the cost of implementing a computer system will be minimized when the parts of the problem are:

- solvable separately
- manageably small

Of course, everyone has a different definition of "manageably small”

Yourdon, Edward; Constantine, Larry Structured Design, 1975

module: A module is a lexically contiguous sequence of program statements, bounded by boundary elements, having an aggregate identifier. Another way of saying this is that a module is a bounded, contiguous group of statements having a single name by which it can be referred to as a unit.

Yourdon, Edward; Constantine, Larry Structured Design, 1975

software component: software components are things that are independently replaceable and upgradeable

- Martin Fowler, Software Component

- components come in two forms: libraries and services.

- components relate to modules, but are replaceable

service: A service is a component that exists in its own process, clients talk to it over some inter-process communication mechanism: RPC, RESTful calls over HTTP, messaging, etc.

- Martin Fowler, Software Component

For services to be components:

- The upgrade/replacement of one service should not require co-ordination with another

autonomous component: it is an architectural style where the activation of the *services* within our system is under the control of the component itself as a response to timers, messaging etc. *

For services to be be autonomous components:

- They need to meet the criteria for a component

- They need to control their own activation; excludes RPC

If you are autonomous from your business partners, you won't share transactions with them. Transactions aren't used across organizations since this may lock up YOUR database if the OTHER organization messes up. Without transactions, you must use multiple messages over time.

- Pat Helland, Autonomous Computing

* This doesn't include RPC, like Thrift or gRPC

microservices: a set of practices for successfully building a *software component* architectural style (at scale):

- Componentization via Services*
- Organized around Business Capabilities
- Smart endpoints and dumb pipes
- Design for failure
- Infrastructure Automation
- Decentralized Data Management
- Decentralized Governance
- Products not Projects

Lewis, James Fowler ,Martin - Microservices

* It is a component and a service i.e. should be replaceable; does not *require* autonomy

Benefits

- Strong Module Boundaries
- Independent Deployment
- Technology Diversity
- Fault Tolerance
- Cognitive Load (Code that fits in my head)

Costs

- Distribution
- Verification
- Operational Complexity
- Skill Issues
- Data: BI & ML

- After Fowler, Martin Microservice Trade-Offs

Microservice Premium

don't even consider microservices unless you have a system that's too complex to manage as a monolith

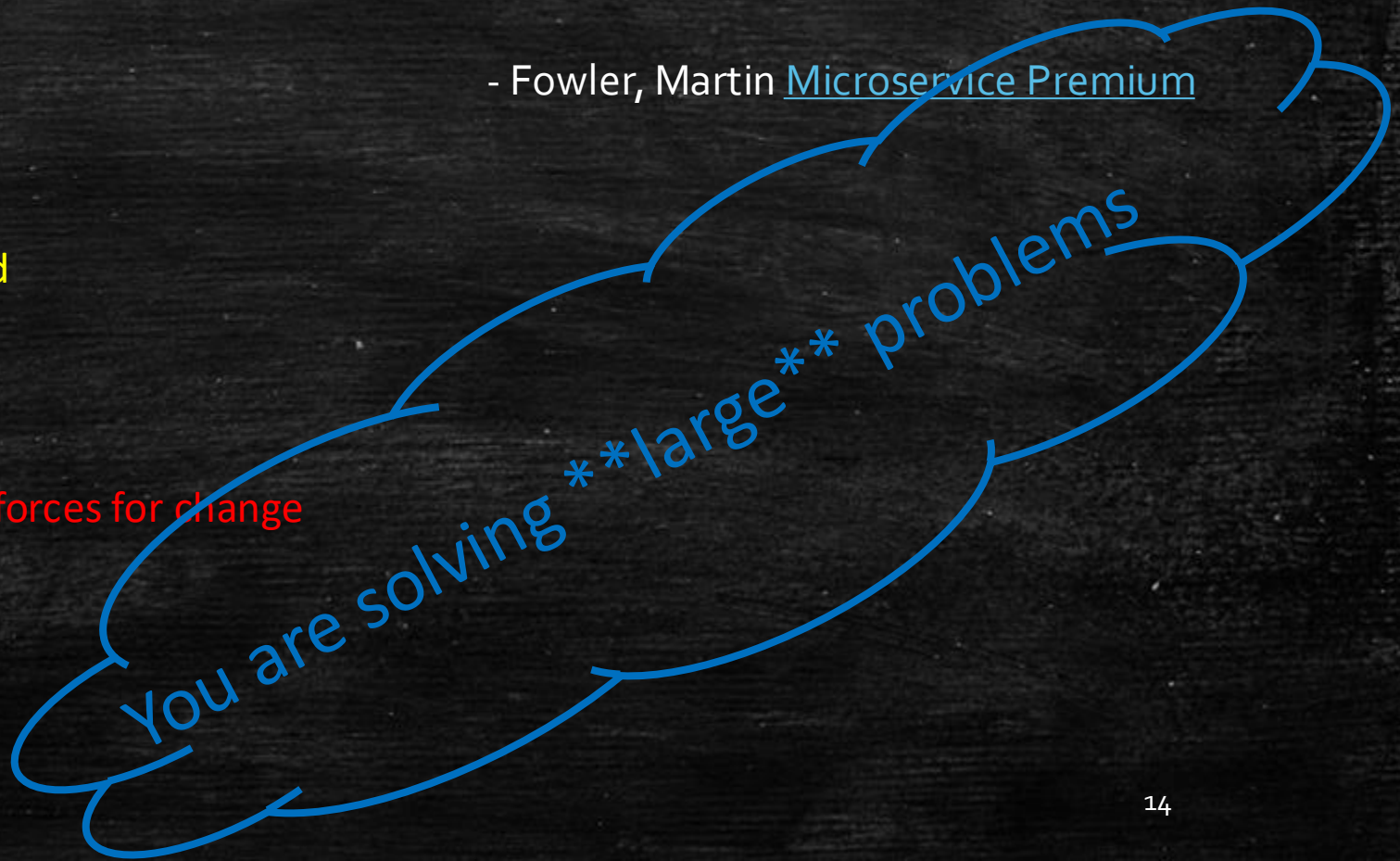
- Fowler, Martin Microservice Premium

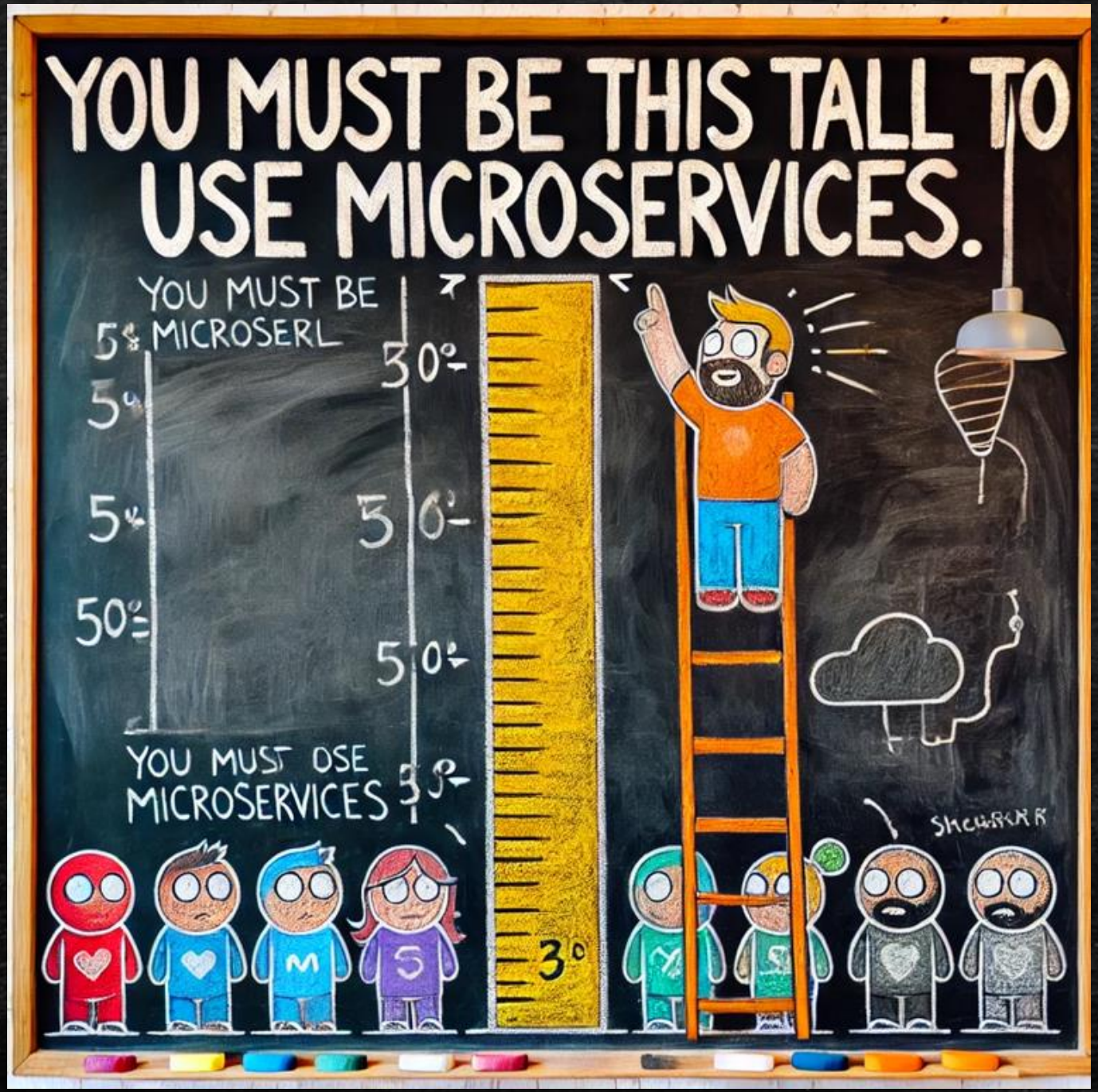
- Code doesn’t fit in your head

- A Five-Pizza Team

- Sub-domains have different forces for change

- A wide range of interactions



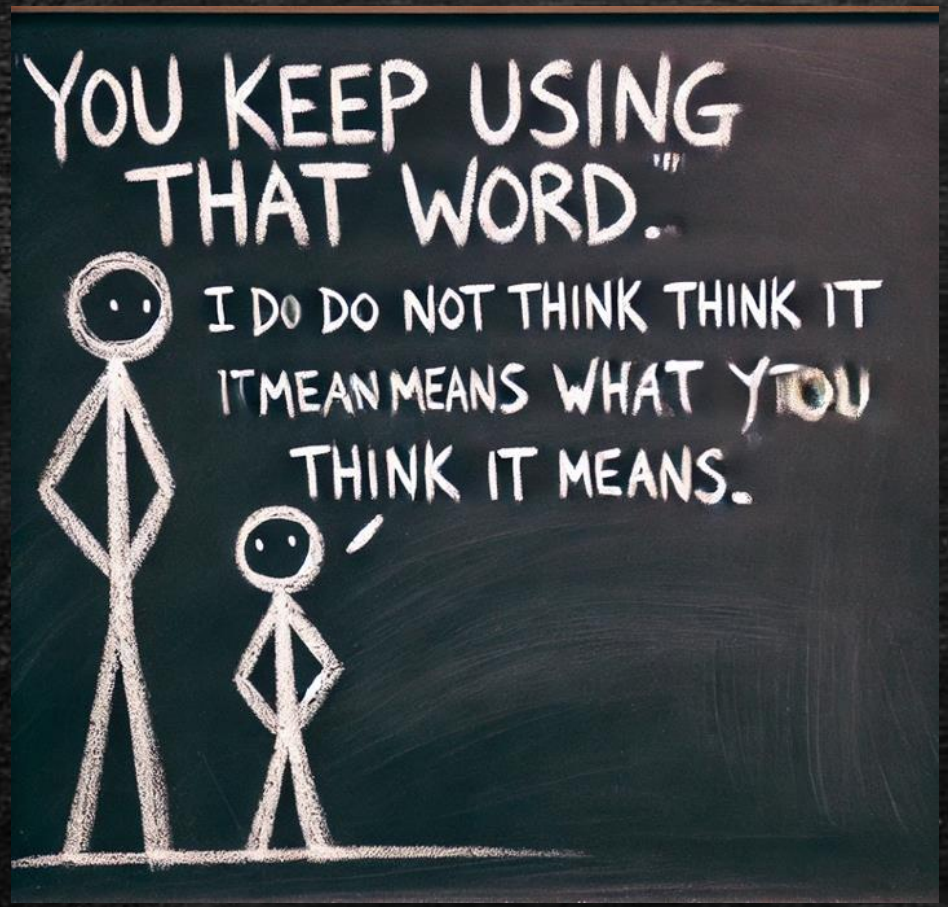

YOU MUST BE THIS TALL TO USE MICROSERVICES.
YOU MUST BE MICROSERL
YOU MUST OSE MICROSERVICES
M
S

The Problem

Parents wonder why the streams are bitter, when they themselves poison the fountain.

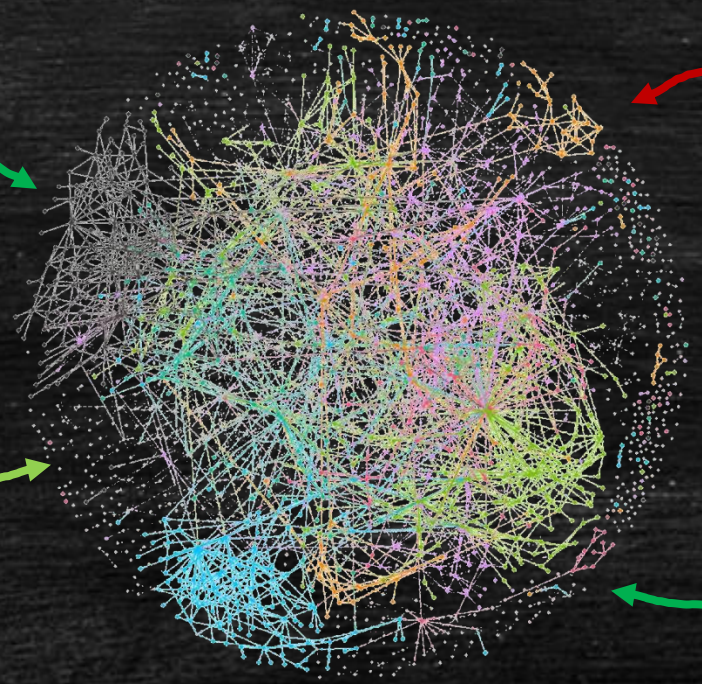
— John Locke,

Microservices



- "They are very, very small. I mean 100 lines of code is probably a big service these days." Fred George, Barcelona Ruby Conference
- "If its more than one programmer to develop & design and maintain it, it's not a microservice" Fred George, GOTO 2016
- "If something starts to be big enough that you view it as an asset that's worth preserving, then it's too big and you should target it for destruction" – Michael Nygard, GOTO 2016

Nanoservices



Many Integration Points

High Latency

High Operational Overhead

Synchronous Orientation

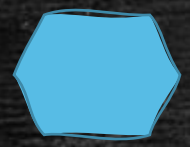
Low Cohesion, High Coupling

Long Release Cycles

Obstacle to Reasoning

Nanoservice – Where the overhead of a being service not balanced by the value.

Microservices

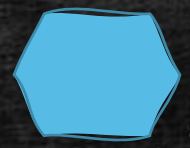


Microservice

=

Process

Each service is autonomous and self-contained and runs a unique process. PaloAltoNetworks.com



Microservice

=

Pod

Each **microservice** is packaged as a **Docker** *container* to enable deployment to a **Kubernetes** cluster for application orchestration. ThinkMicroservices.com

A cloud native application consists of discrete, reusable components that are known as microservices that are .. packaged in containers IBM.com

Microservice

=

FaaS

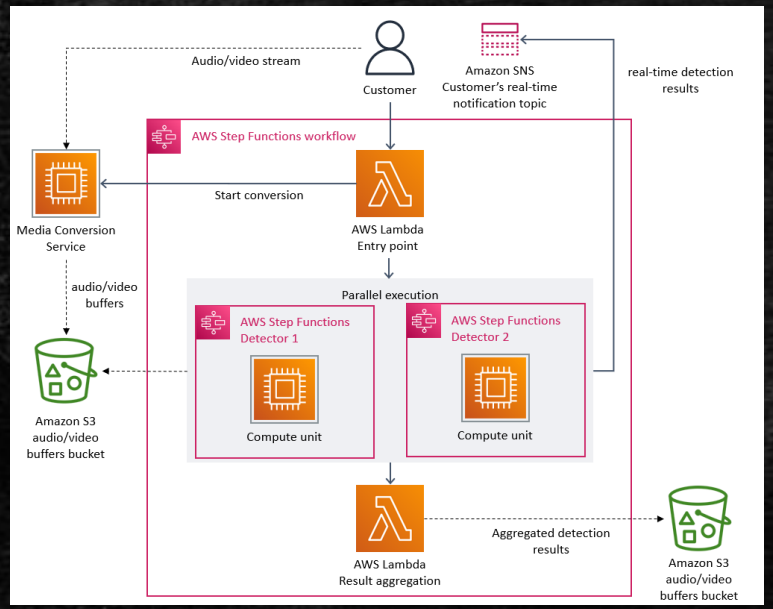
functions .. launched by AWS Lambda, is all that you need to build a microservice … to the level of granularity desired AWS.Amazon.com

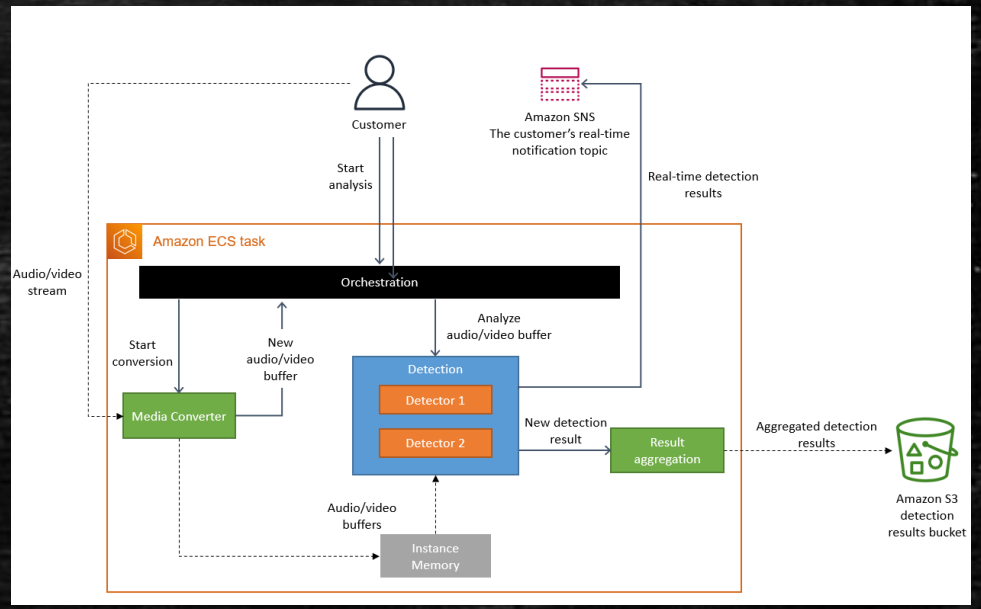
Source: Prime Video Tech (Archived)

Prime Video

From distributed microservices to a monolith application

We realized that distributed approach wasn't bringing a lot of benefits in our specific use case, so we packed all of the components into a single process.





Even Amazon can't make sense of serverless or microservices

That really sums up so much of the microservices craze that was tearing through the tech industry for a while: IN THEORY. Now the real-world results of all this theory are finally in, and it's clear that in practice, microservices pose perhaps the biggest siren song for needlessly complicating your system. And serverless only makes it worse.

— Hansson, David Heinemeier blog

Fracture Plane: Team Availability

A fracture plane is a natural seam in the software system that allows the system to be split easily into two or more parts….

Skelton, Matthew; Pais, Manuel. Team Topologies

Fracture plane: team availability is an anti-pattern that occurs when the driving decision is that we want to create a fracture plane to support a new team working on the problem over assigning to an existing team, for the whom the work may not represent the highest priority, to achieve faster time-to-market.

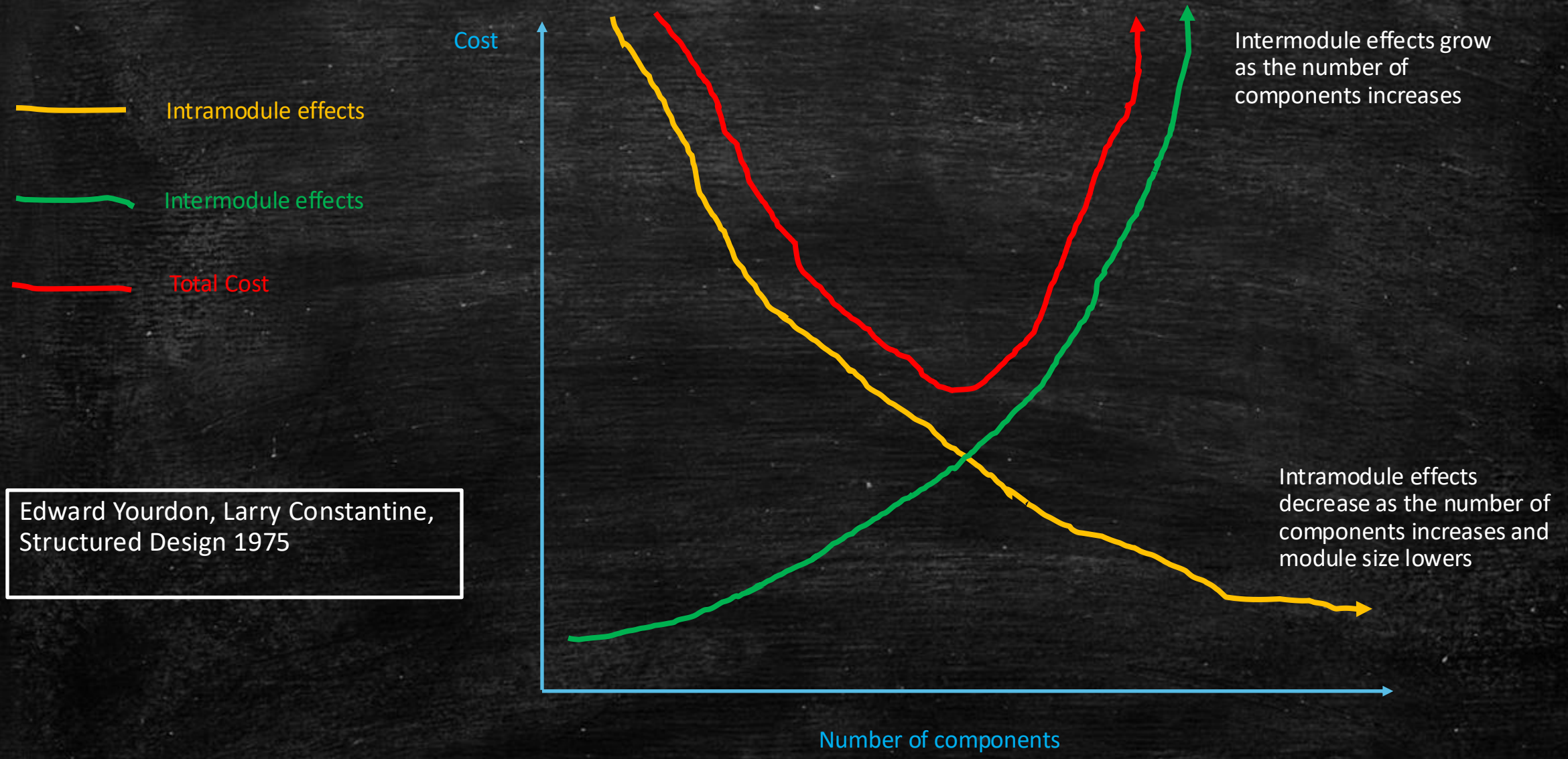
Fracture plane: team availability leads to poor cohesion at best, high coupling at worst. It may lead to nanoservices.

Cost of Ownership

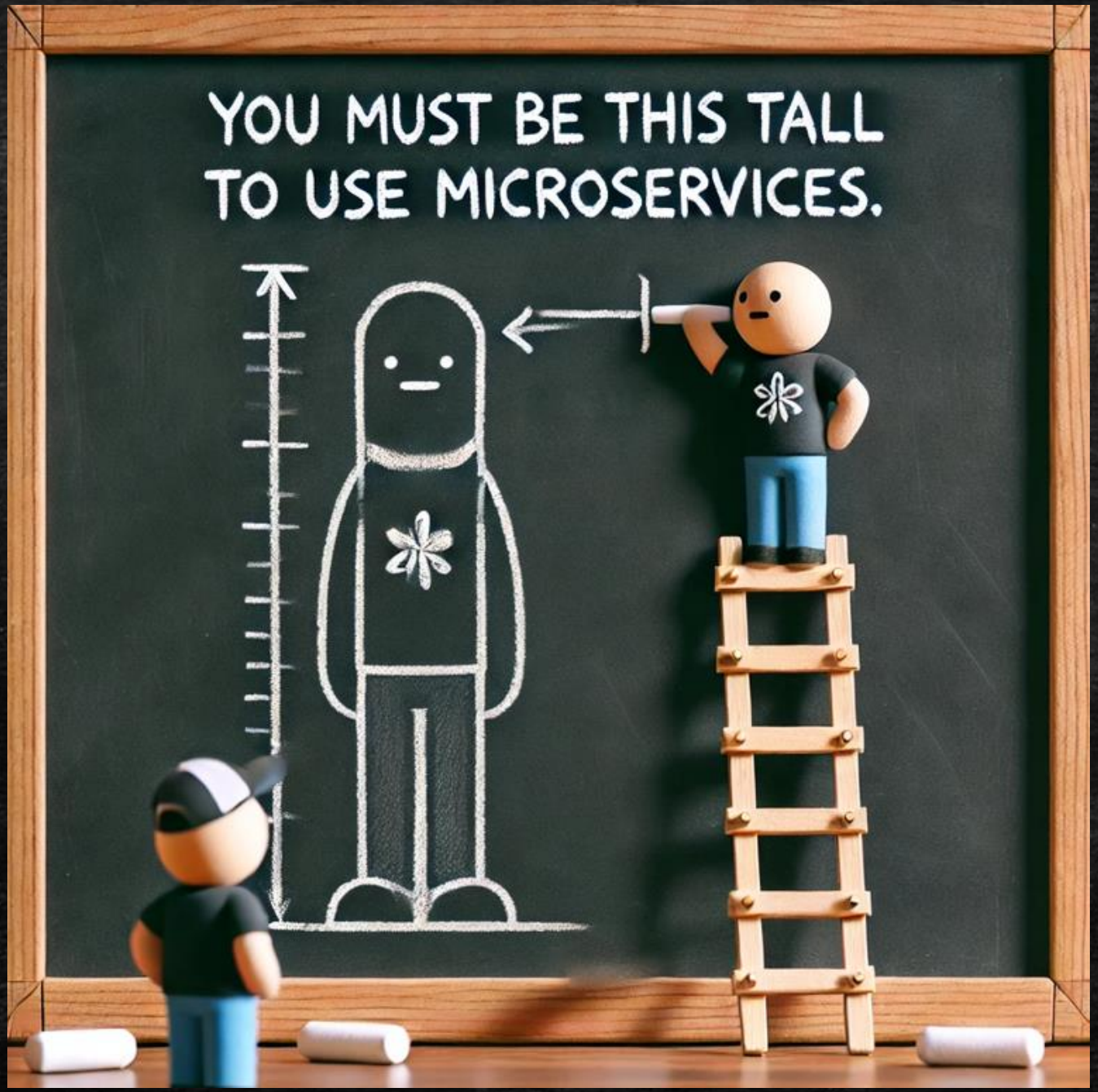
The cost of ownership of software exceeds the cost of authorship (proof: assume a two-pizza team takes 3 months to write software but 36 months owning it, the cost of ownership is an order of magnitude greater).

Reducing the cost of authorship improves the time to market. In a context where speed of feature release wins we risk prioritizing fracture plane: team availability.

But over time a growing number of components will bankrupt us as ownership costs rise-and rise.



Edward Yourdon, Larry Constantine, Structured Design 1975

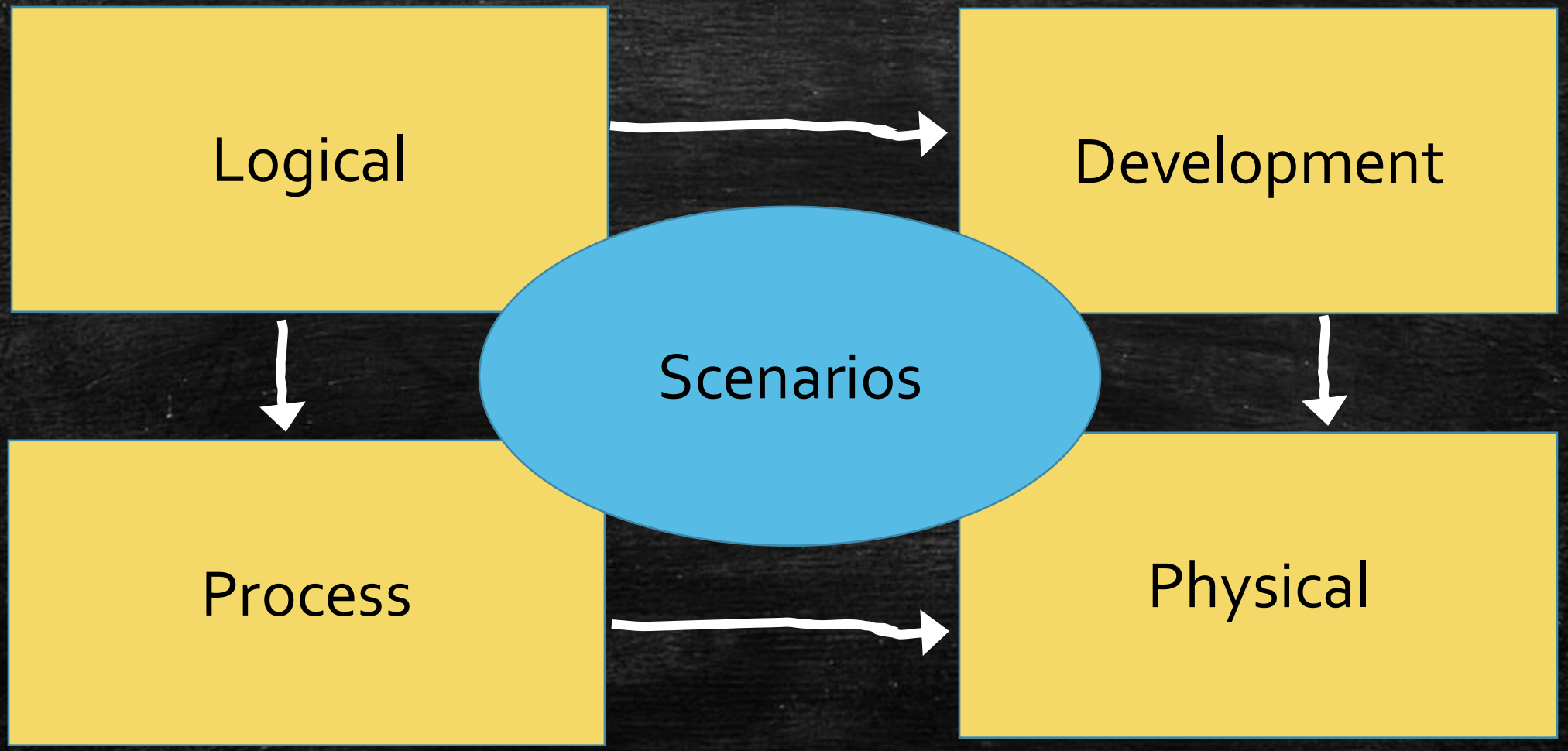

YOU MUST BE THIS TALL
TO USE MICROSERVICES.

Advice

Why is it that a child sometimes does the opposite of what he is told? Why would a person sometimes dislike receiving a favor? Why is propaganda frequently ineffective in persuading people? And why would the grass in the adjacent pasture ever appear greener?.

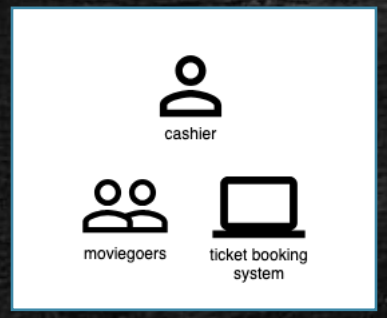
— Jack Brehm, 1966

The “4 +1 “ View Model - Philippe Kruchten

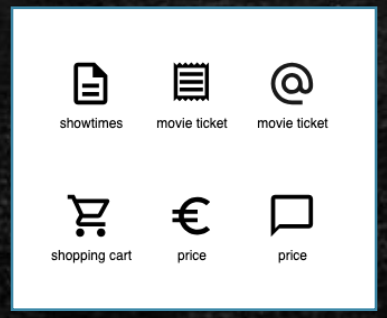


- See Chris Simon, Experiences scaling a modular monolith to microservices using the 4+1 views

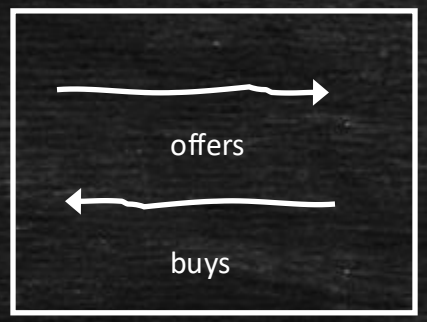
Scenarios



Actors: systems or people the story is about



Work Objects: actors work with or exchange information about work objects: resources in the domain



Activities: An activity describes the relationship of the actor to the work object – what do they do to it?

[Domain Storytelling]

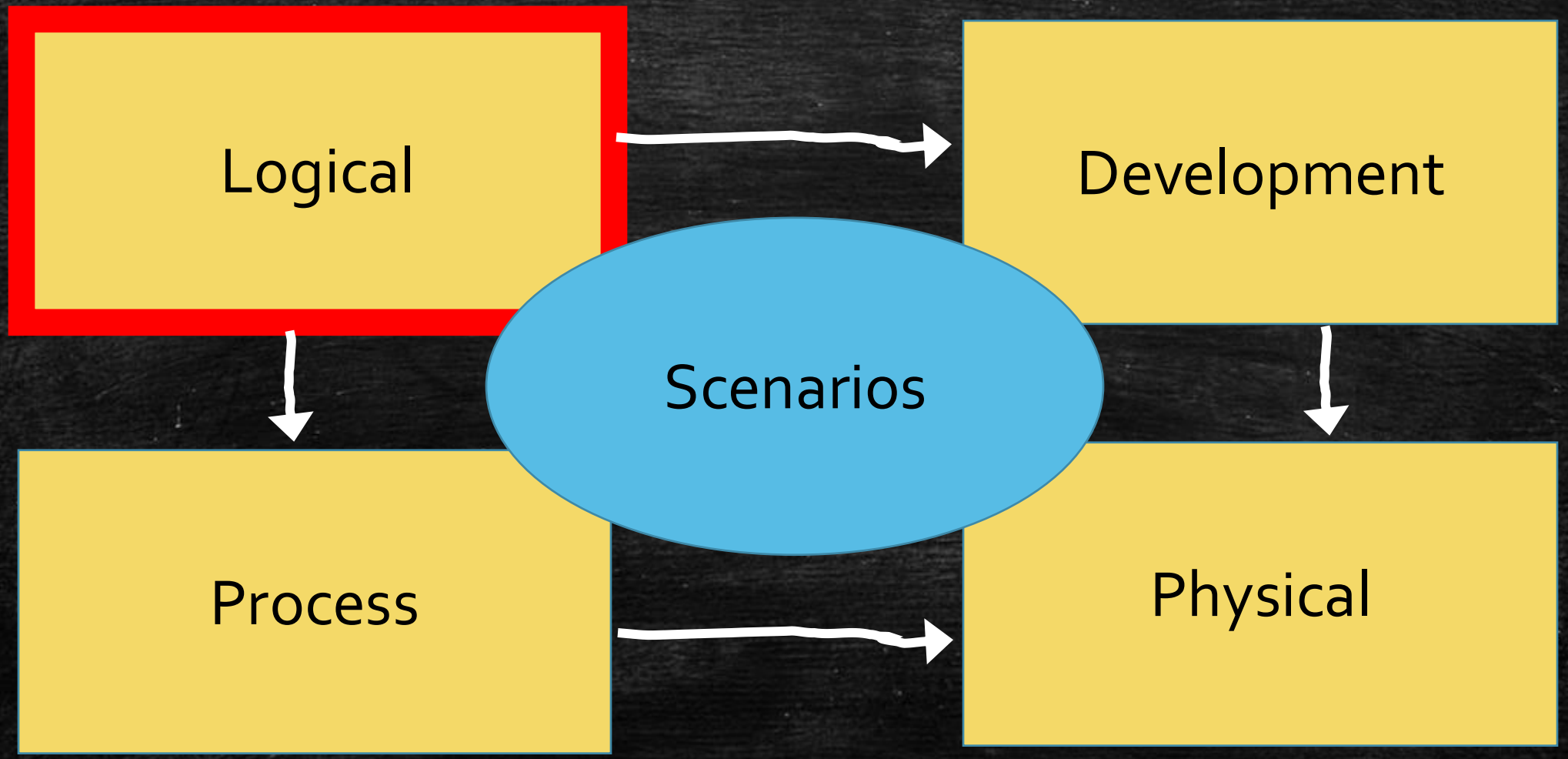
Scenarios

Table 1: Structure of Matrix to Evaluate Candidate Patterns

	Pattern 1		**Pattern 2**		. . .	**Pattern n**	
	Pros	Cons	Pros	Cons		Pros	Cons
Architectural driver 1							
Architectural driver 2							
. . .							
Architectural driver n							

[Attribute Driven Design]

The “4 +1 “ View Model - Philippe Kruchten



- See Chris Simon, Experiences scaling a modular monolith to microservices using the 4+1 views

Logical

The logical architecture primarily supports the functional requirements—what the system should provide in terms of services to its users. The system is decomposed into a set of key abstractions, taken (mostly) from the problem domain

Kruchten, Philippe Architectural Blueprints the 4 +1 View Model of Software

This is where we explore our requirements, typically a domain model

Logical

A fracture plane is a natural seam in the software system that allows the system to be split easily into two or more parts…. It is usually best to try to align software boundaries with the different business domain areas…

Skelton, Matthew; Pais, Manuel. Team Topologies

How do we find these sub-domains within our system?

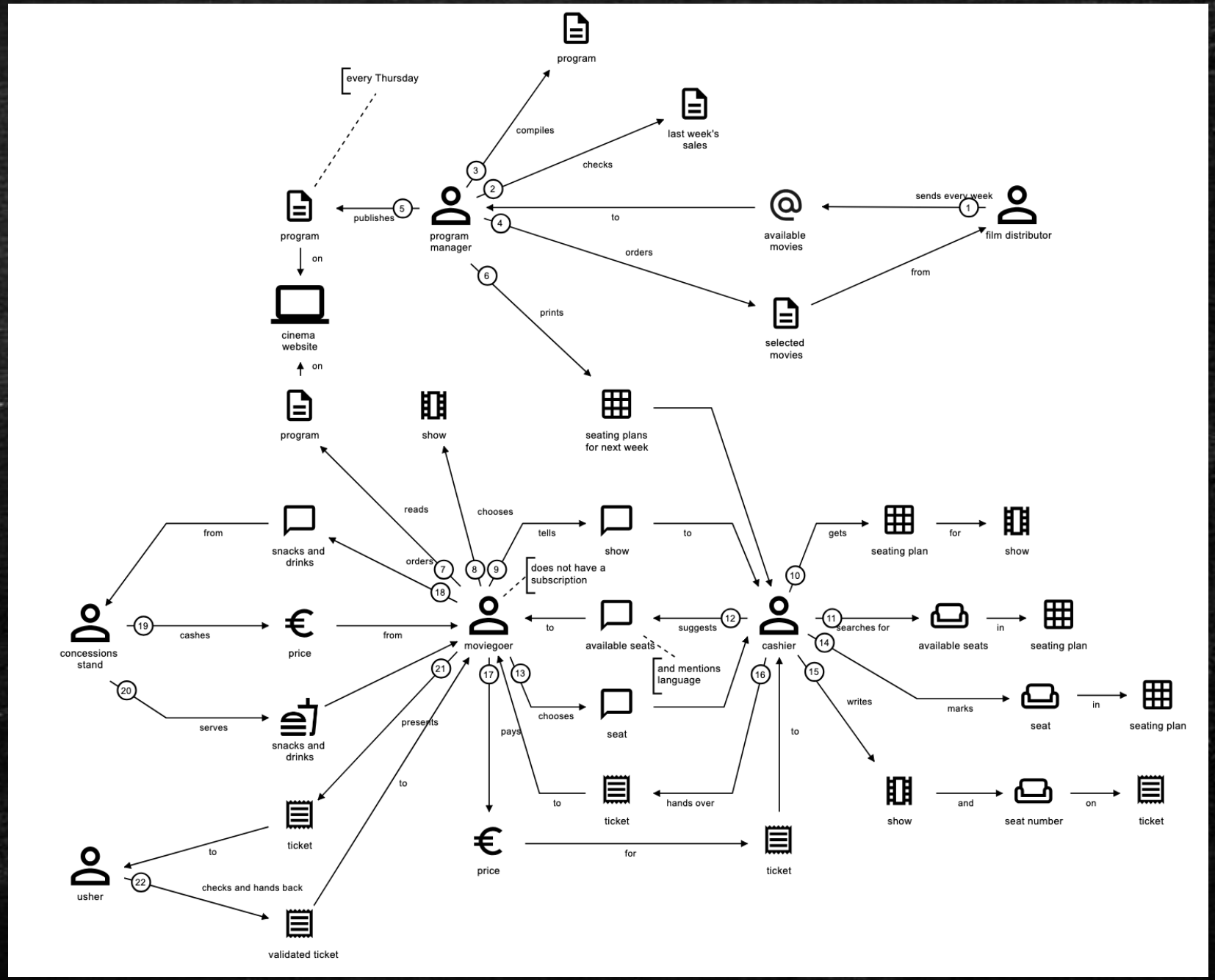
- Natural seams within the domain understood by SMEs

- Domain Storytelling

- Value Stream Mapping to align with processes

Domain Storytelling

Examples



Domain Storytelling and Boundaries

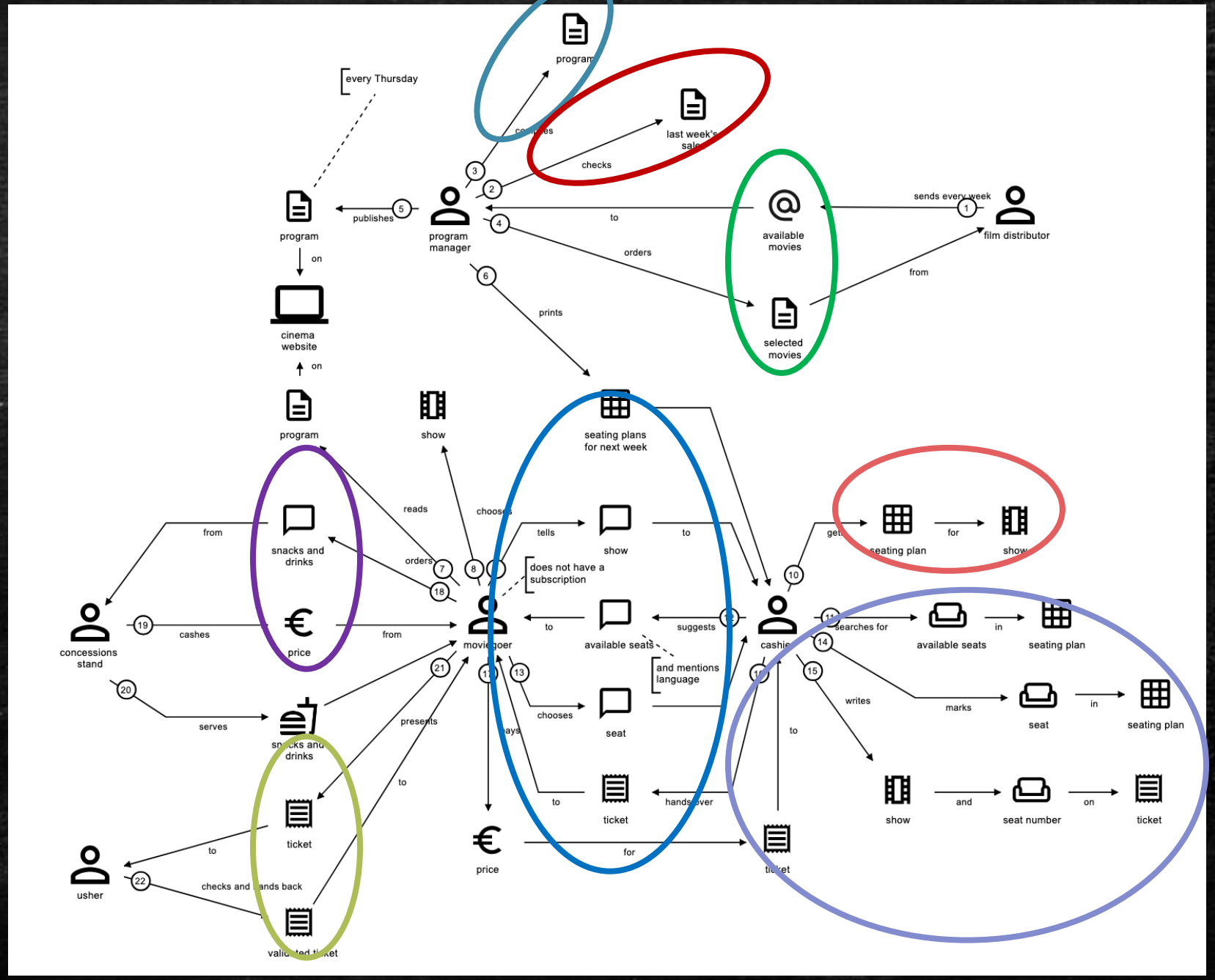
Subdomains consist of activities that belong together from an actor's perspective.

- Actor produces result on their own
- One-way information flow
- Different triggers
- Activities supporting something not in the picture
- Differences in language
- Different use of the same thing

Hofer, Stefan; Schwentner, Henning. Domain Storytelling

Domain Storytelling

Examples



Logical

The Domain will change as the business changes:

- The changes to our domain will be a constant force; a successful business is not static

- Tasks and activities change more frequently, are retired, replaced or introduced

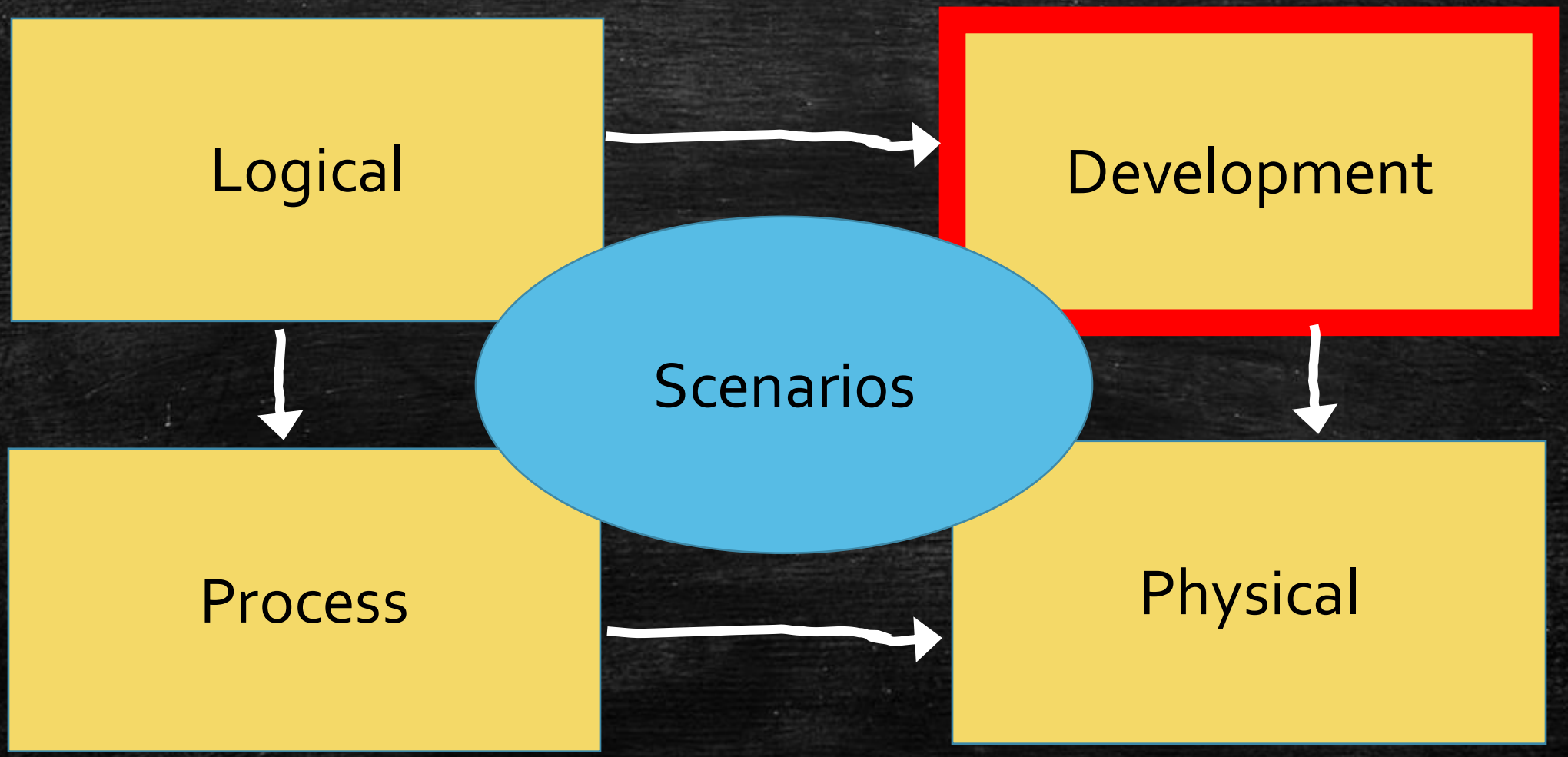
- Processes change more slowly, are less frequently retired, replaced; may be more frequently introduced as new opportunities emerge.

- Our logical view requires constant gardening

- We need a continuous conversation to share the logical model as it changes.


YOU MUST BE THIS TALL
TO USE MICROSERVICES.
MS
5.
8"

The “4 +1 “ View Model - Philippe Kruchten



- See Chris Simon, Experiences scaling a modular monolith to microservices using the 4+1 views

Development

The development architecture focuses on the actual software module organization on the software development environment. The software is packaged in small chunks—program libraries, or subsystems — that can be developed by one or a small number of developers. The subsystems are organized in a hierarchy of layers, each layer providing a narrow and well-defined interface to the layers above it.

Kruchten, Philippe Architectural Blueprints the 4 +1 View Model of Software

This is where we define the service boundaries that map to the logical model

Development

A fracture plane is a natural seam in the software system that allows the system to be split easily into two or more parts…. It is usually best to try to align software boundaries with the different business domain areas…
Fracture Plane: **Business Domain Bounded Context**

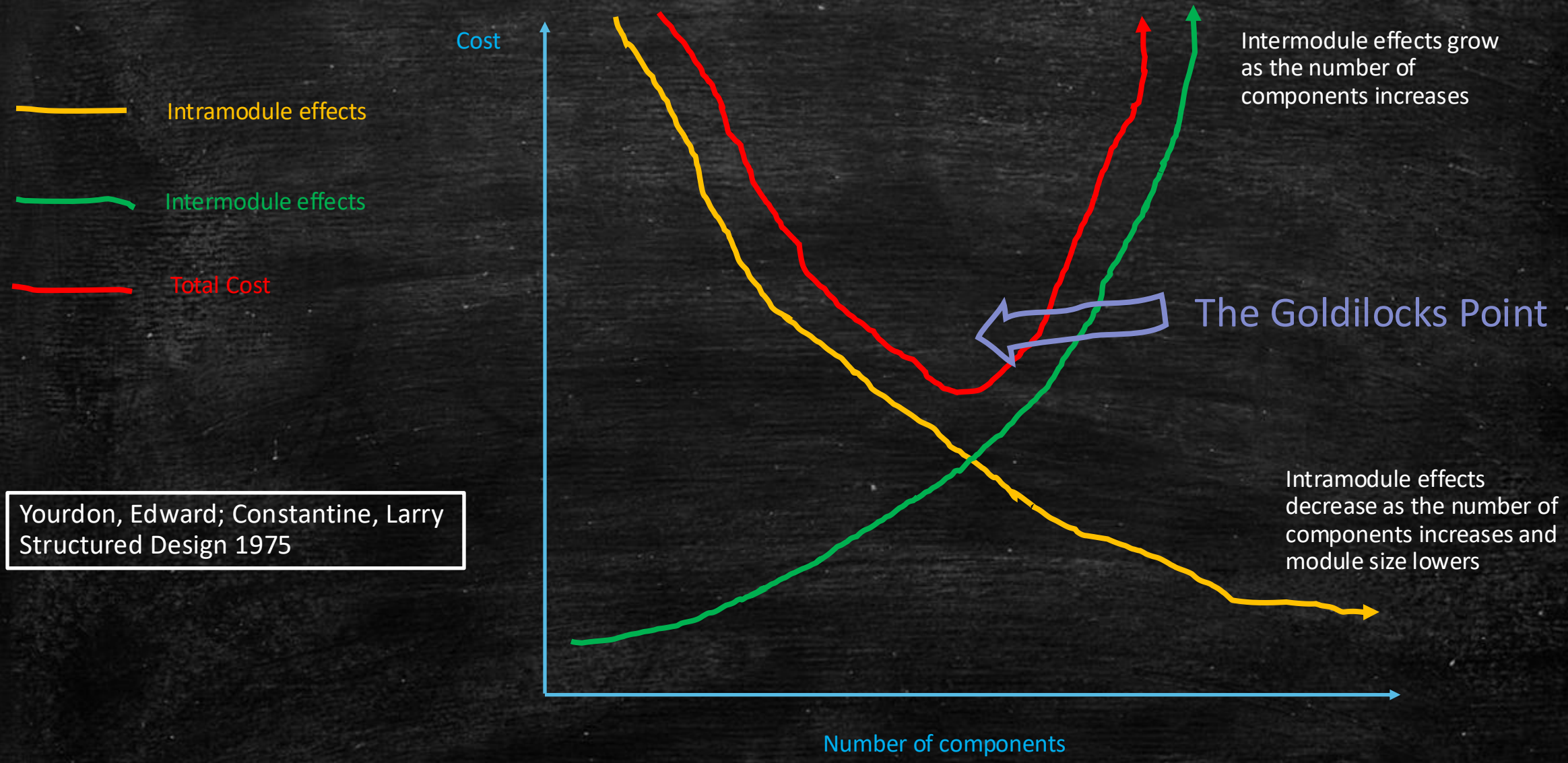
Skelton, Matthew; Pais, Manuel. Team Topologies

Multiple [domain] models are in play on any large project. Explicitly define the context within which a [domain] model applies. Explicitly set boundaries in terms of team organization, usage within specific parts of the application, and physical manifestations such as code bases and database schemas. Keep the [domain] model strictly consistent within these bounds, but don't be distracted or confused by issues outside. A BOUNDED CONTEXT delimits the applicability of a particular [domain] model.

Evans, Eric Domain-Driven Design

Many designers have made attempts to chop a system into manageably small pieces; unfortunately, they have often found that implementation time increased rather than decreased. The key frequently lies in the second part of our stipulation above: **The parts of the original problem must be solvable separately.**

Yourdon, Edward; Constantine, Larry Structured Design, 1975



Yourdon, Edward; Constantine, Larry
Structured Design 1975

Granularity Integrators and Disintegrators

Granularity Disintegrators

Service scope too large

Distribution of code changes

Continuous change for new problems

Granularity Integrators

ACID transactions

Shared Code

Shared Data

After Ford, Neal; Richards, Mark; Sadalage, Pramod; Dehghani, Zhamak. Software Architecture: The Hard Parts

Development

Simple model: one bounded context per subdomain (two-pizza team).

- Code for the subdomain is fits neatly into your head

- Forces for change or replacement are cohesive: keep what changes together, together

- Allows ACID requirements to be satisfied within the service boundary

- This is the ideal microservice

    - The microservice approach to division is different, splitting up into services organized around business capability.

Lewis, James Fowler, Martin - Microservices

Development

Simple model: one bounded context per domain (two-pizza team).

- Code for the domain is much smaller than your head
- Should this be a module and combined with other domains in one bounded context?
  - Is this domain truly independent of others? Does it have separate reasons to change?
  - Are the behaviours about the same work item?
  - Are we talking to ourself?
  - Will we distribute a transaction without necessity?
  - Will we weaken cohesion as a separate context?
  - Are we likely to be a nanoservice?

Development

More complex model:

- Is the code for the domain larger than your head?
  - Can you find seam(s) within the domain and split into multiple bounded contexts
    - Within a business process look for activities within the process (in domain storytelling deal with different work items)
  - Beware complexity of intermodule dependencies
- If there are multiple contexts, now define relationship between contexts
- If independently solvable each split within the domain becomes a microservice
  - But might create dependency issue around transactions or shared data
- If not independently solvable you need a macroservice
  - Might use modularity if the macroservice has non-independent bounded contexts

Development

Macroservice:

- not a monolith

- Has no more than 20 devs/3 teams working on the service (5 pizza rule?)

- May or may not have/need monorepo.

- Dependency management becomes a lot easier (though still non-trivial) the fewer the services/repos
- better observability, debugging

Sridharan, Cindy @copyconstruct X

Development

We identified that our domain will change. This implies that the bounded contexts we have drawn will move out-of-alignment

When your components are services with remote communications, then refactoring is much harder than with in-process libraries.

Lewis, James Fowler ,Martin - Microservices

A hyperliminal system is a system where a complicated, ergodic, ordered system executes inside a complex, non-ergodic, disordered context... It is the job of the architect to design a structure for the complicated system that will allow it to survive as the complex context around it changes and moves.

O'Reilly Barry - Residues

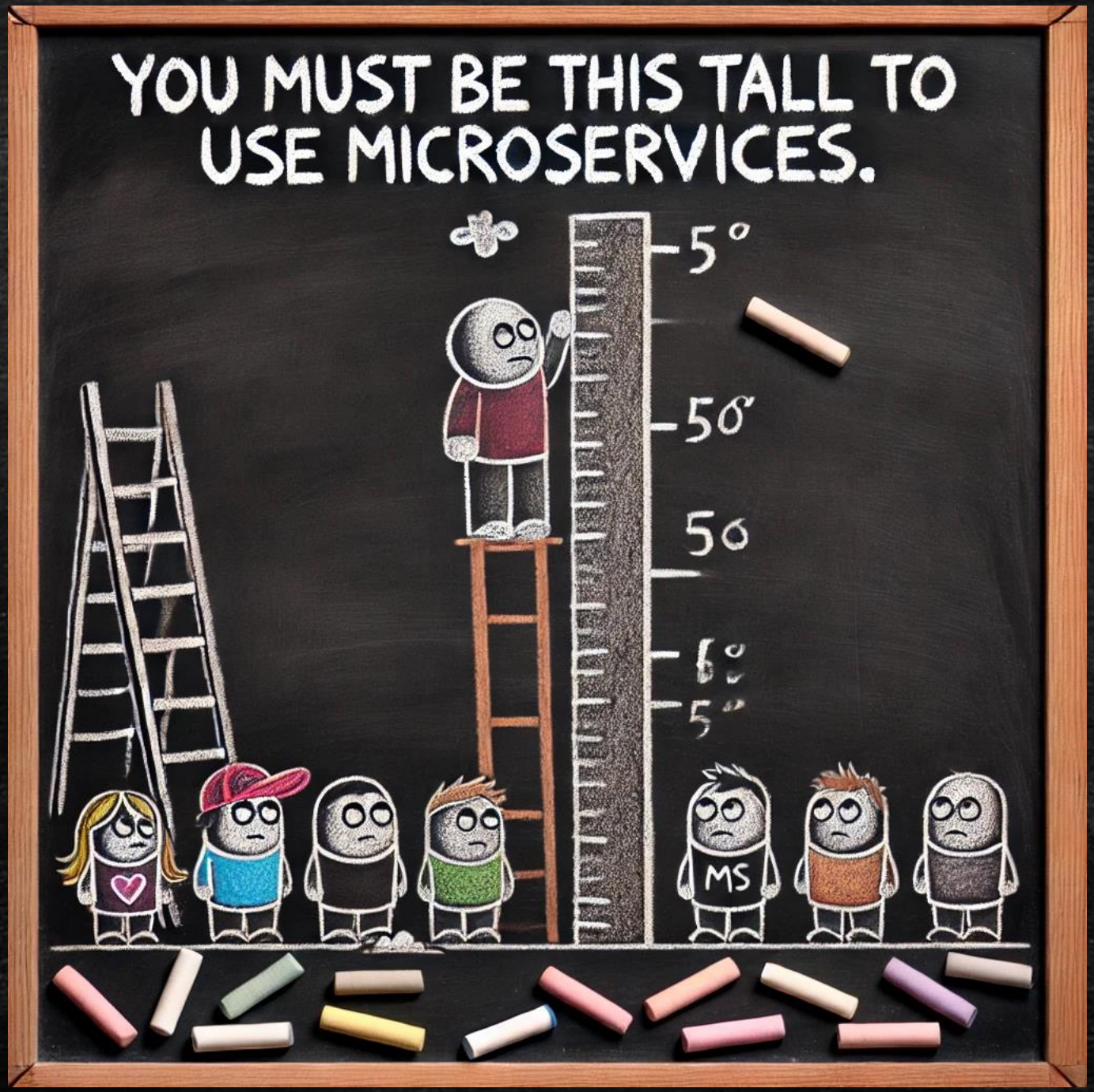
Development

Kauffman identified the property of criticality… This means that at a certain level of N (nodes) and K (links), a system is resilient to unexpected changes and at the same time not so complicated that it collapses under the weight of managing its own resources.

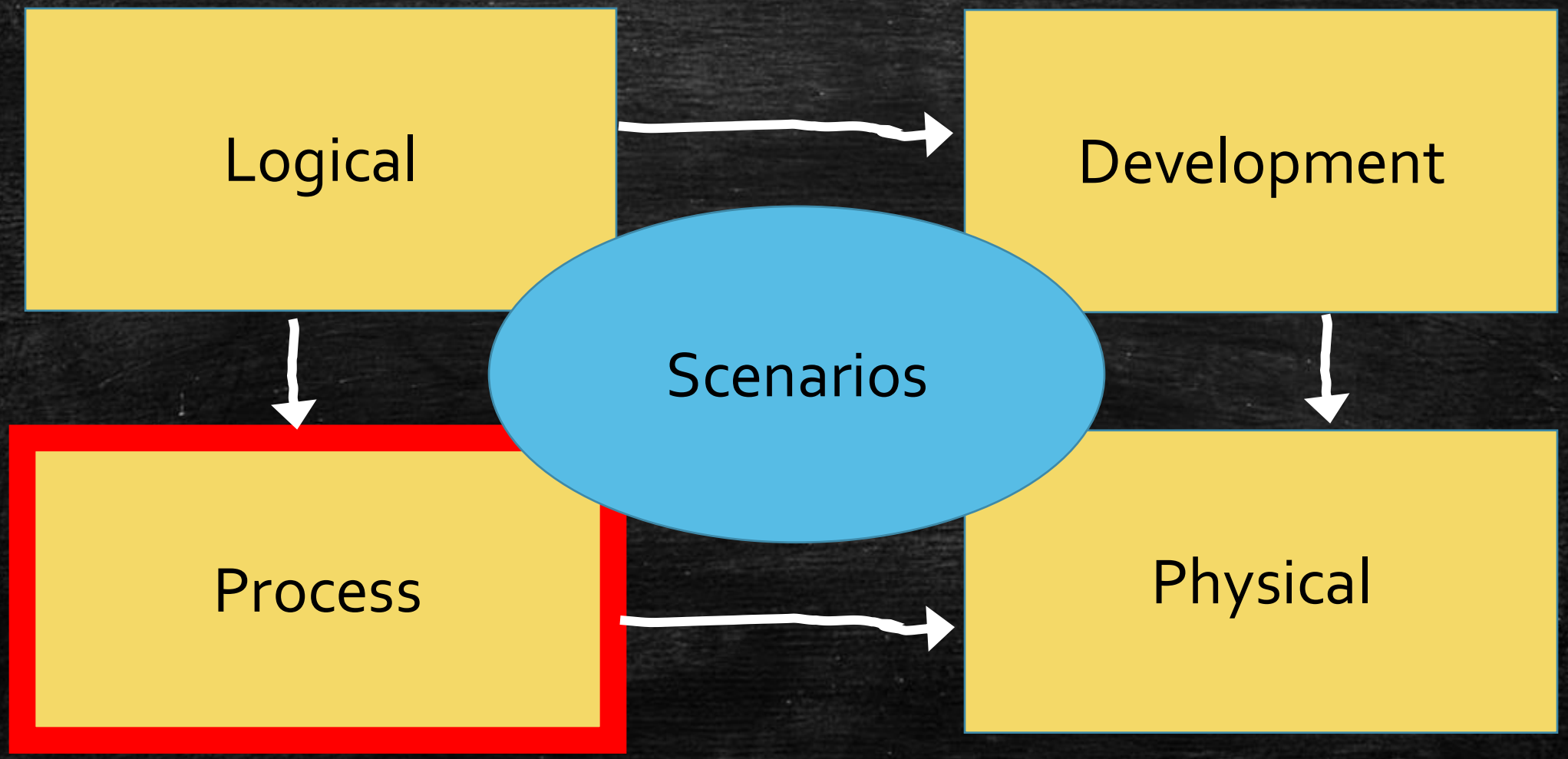
A good parallel in the software industry is the comparison of monoliths (Low N and K) and microservices (high N and K). Criticality is finding the balance between these things.

O'Reilly Barry - Residues

Development


YOU MUST BE THIS TALL TO USE MICROSERVICES.
5'
5'
5'
MS

The “4 +1 “ View Model - Philippe Kruchten



- See Chris Simon, Experiences scaling a modular monolith to microservices using the 4+1 views

Process

The process architecture takes into account some non-functional requirements, such as performance and availability. It addresses issues of concurrency and distribution, of system's integrity, of fault-tolerance, and how the main abstractions from the logical view fit within the process architecture…A process is a grouping of tasks that form an executable unit. Processes represent the level at which the process architecture can be tactically controlled (i.e., started, recovered, reconfigured, and shut down)

Kruchten, Philippe Architectural Blueprints the 4 +1 View Model of Software

This is where we define the processes that make up a microservice

Process

This is about how our processes interoperate with each other and infrastructure

- Seek to be application-centric, not infrastructure centric
- 12-Factor Apps

Interoperability

- Autonomous Component
  - Asynchronous communication
  - Independent Deployment
  - Reactive Architecture
  - ECST
- Software Component
  - Synchronous communication
  - REST: Independent deployment (uniform interface; resource versioning)
  - RPC: Distributed Ball of Mud

Process

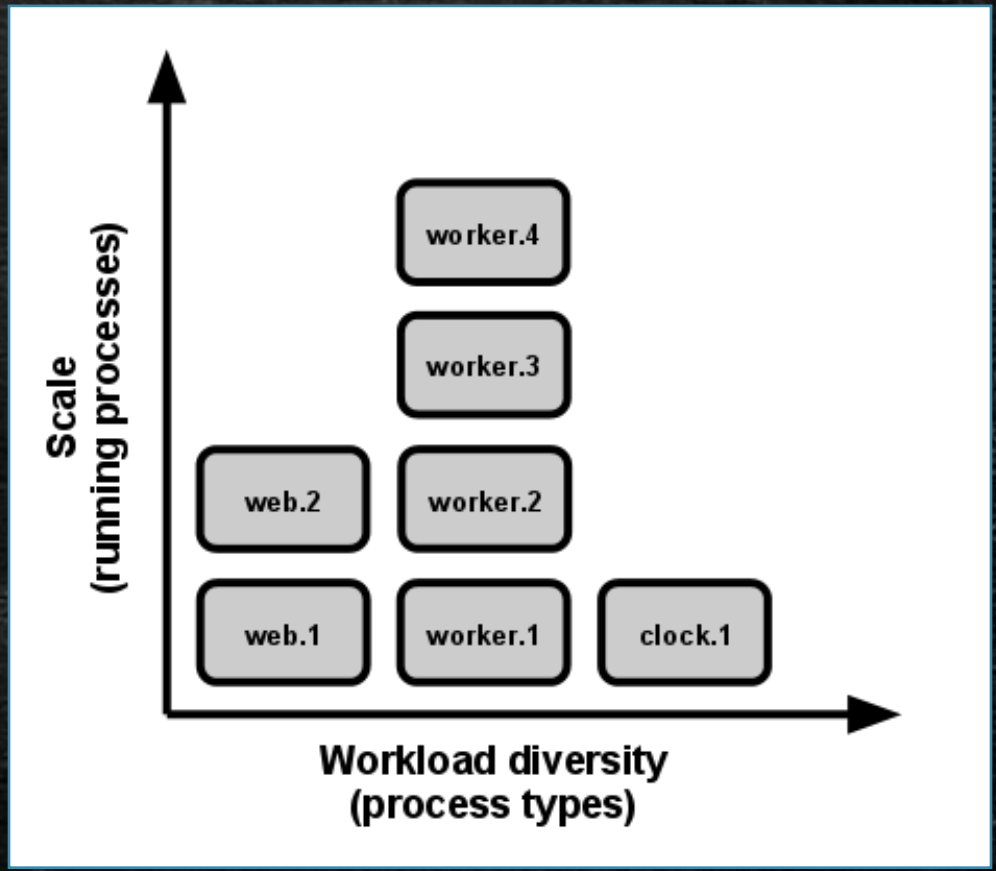
At a first approximation, we can observe that services map to runtime processes, but that is only a first approximation. A service may consist of multiple processes that will always be developed and deployed together, such as an application process and a database that's only used by that service.

Lewis, James Fowler ,Martin - Microservices

Process

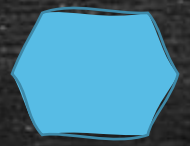
In the twelve-factor app, processes are a first class citizen. Processes in the twelve-factor app take strong cues from the unix process model for running service daemons.

Using this model, the developer can architect their app to handle diverse workloads by assigning each type of work to a *process type*. For example, HTTP requests may be handled by a web process, and long-running background tasks handled by a worker process... the array of process types and number of processes of each type is known as the *process formation*



Concurrency - 12-Factor App

Process

μ = 

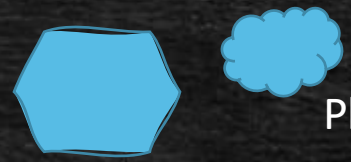
Microservice

App

M = 

Macroservice

App App

Or 

Plugin

Microkernel

App Or Microkernel Plugin

= 

Process Type Process Type Process Type

Process Formation

Granularity Integrators and Disintegrators

Granularity Disintegrators

Scalability and throughput

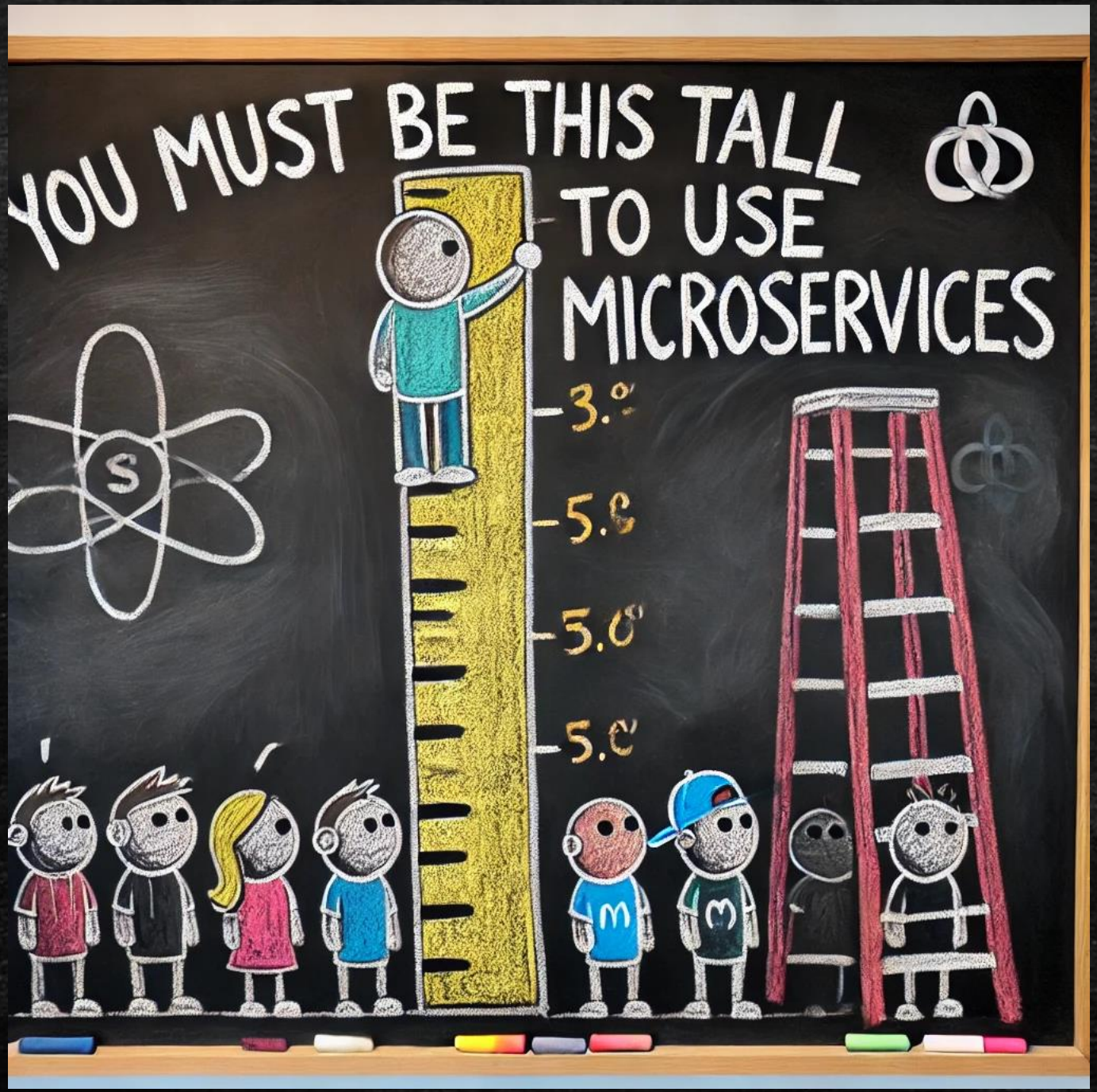
Fault tolerance

Granularity Integrators

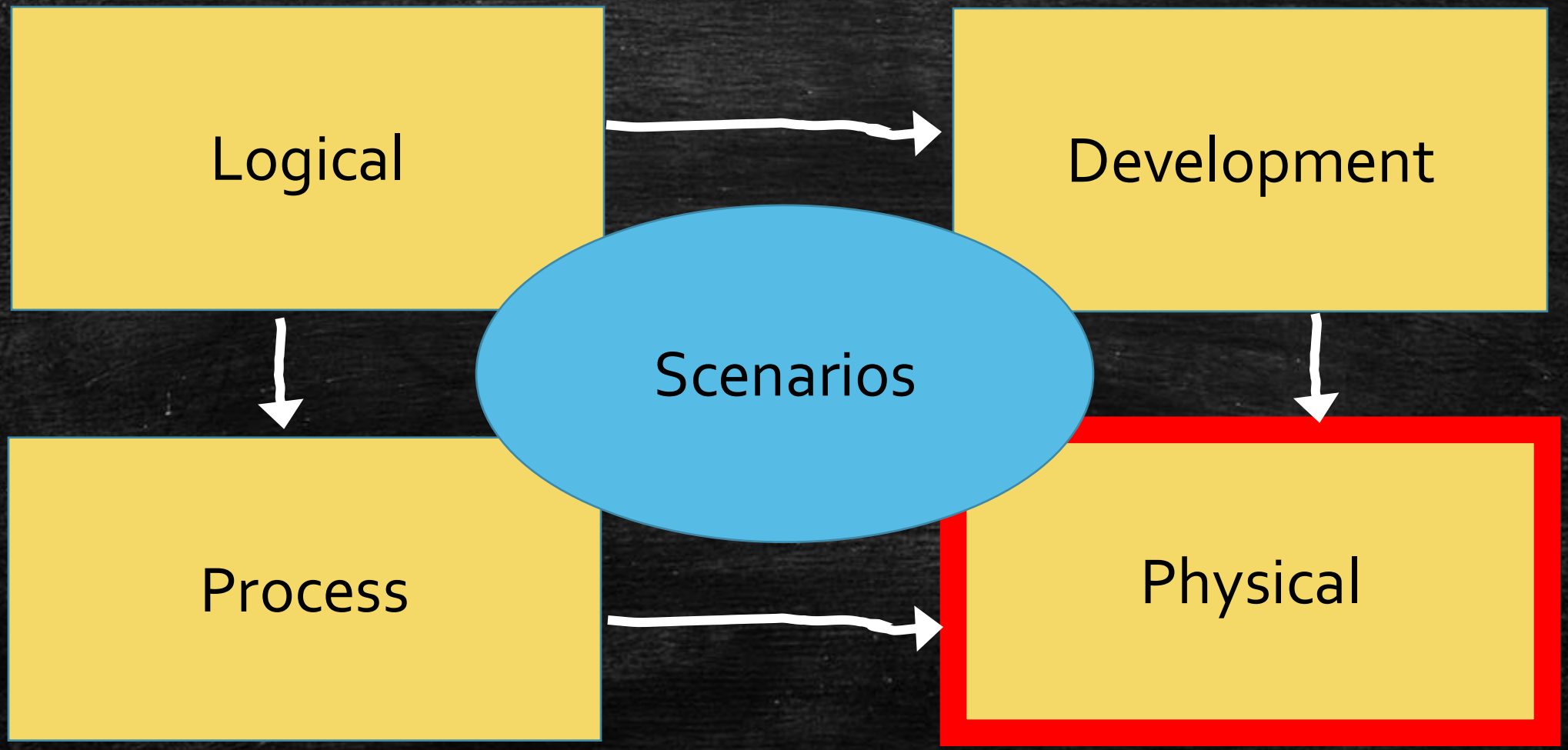
ACID transactions

Shared Code

After Ford, Neal; Richards, Mark; Sadalage, Pramod; Dehghani, Zhamak. Software Architecture: The Hard Parts


YOU MUST BE THIS TALL
TO USE
MICROSERVICES

The “4 +1 “ View Model - Philippe Kruchten



- See Chris Simon, Experiences scaling a modular monolith to microservices using the 4+1 views

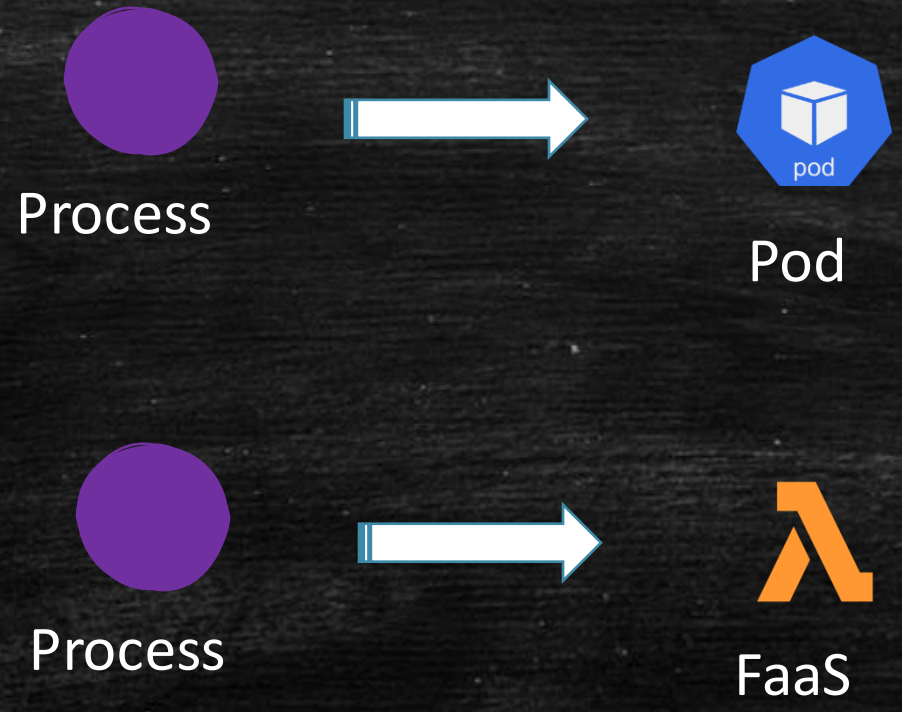
Physical

The software executes on a network of computers, or processing nodes (or just nodes for short). The various elements identified—networks, processes, tasks, and objects—need to be mapped onto the various nodes. We expect that several different physical configurations will be used: some for development and testing, others for the deployment of the system for various sites or for different customers.

Kruchten, Philippe Architectural Blueprints the 4 +1 View Model of Software

This is where we map the processes to hosts

Physical



Q&A

“Entities are not to be multiplied beyond necessity.” William of Ockham