SE 577
Software Architecture

Architectural Styles

Acknowledgement

- Material from several sources including:
  - D. Garlan and M. Shaw, “An Introduction to Software Architecture”, CMU-CS-94-166
  - R.N. Taylor, N. Medvidovic, and E.M. Dashofy. *Software Architecture: Foundations, Theory and Practice*, Wiley.
  - Farshidi, Jansen, and van der Wolf. Capturing software architecture knowledge for pattern-driven design. Journal of Systems and Software, 2020
  - Erder, Pureur, and Woods. Continuous Architecture in Practice
  - Richards and Ford. Fundamentals of Software Architecture
  - Carvantes and Kazman. Designing Software Arhitectures

Software Architecture - Ingredients

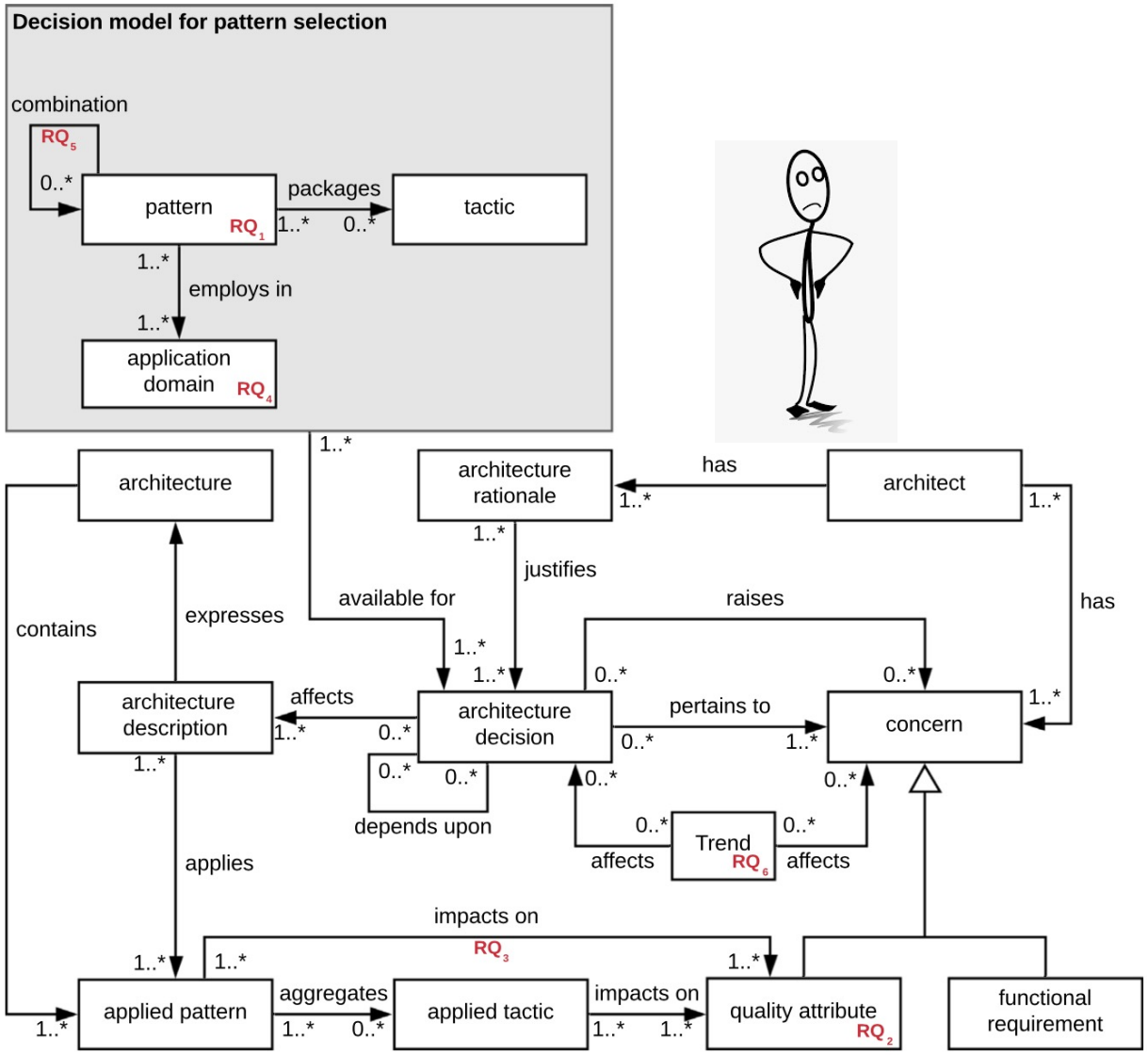
Most definitions of software architecture generally talk about

- What are the important "architectural" components
- What are the connectors between the architectural components
- What patterns, constraints, or decisions govern the restrictions on connecting the architectural components.

Think about it, not all ways to connect architecture components together are equally good.

Architecture as an Algorithm

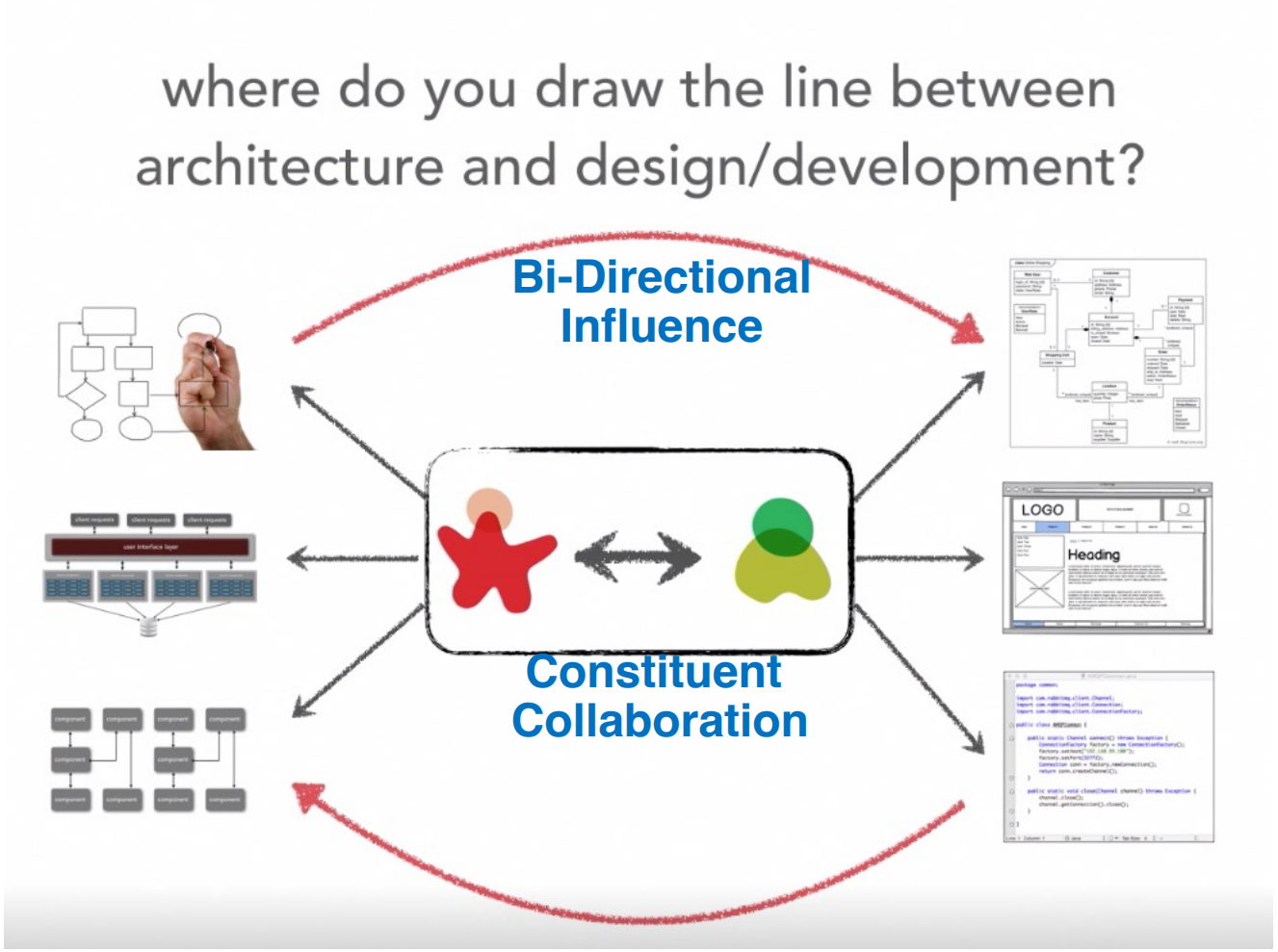
https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:42020:ed-1:v1:en



Term	Definition
Software architecture	Software architecture is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships between them.
Pattern	universal and reusable solutions to commonly occurring problems in software architecture.
Tactic	design decisions that improve individual quality attribute concerns
Quality	The quality of a system is the degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value.
Architect	person, team, or organization responsible for systems architecture
Rationale	captures the knowledge and reasoning that justify the resulting design, and its primary goal is to support designers by providing means to record and communicate the argumentation and reasoning behind the design process.
Decision	A decision is consisting of a restructuring effect on the components and connectors that make up the software architecture, design rules imposed on the architecture and resulting system as a consequence, design constraints imposed on the architecture, and a rationale explaining the reasoning behind the decision.
Functional requirement	condition or capability that must be met or possessed by a system, system component, product, or service to satisfy an agreement, standard, specification, or other formally imposed documents
Concern	is any interest in the system. The term is derived from the phrase “separation of concerns” as in Software Engineering. One or more stakeholders may hold a concern. Concerns involve system considerations such as performance, reliability, security, availability, and scalability.

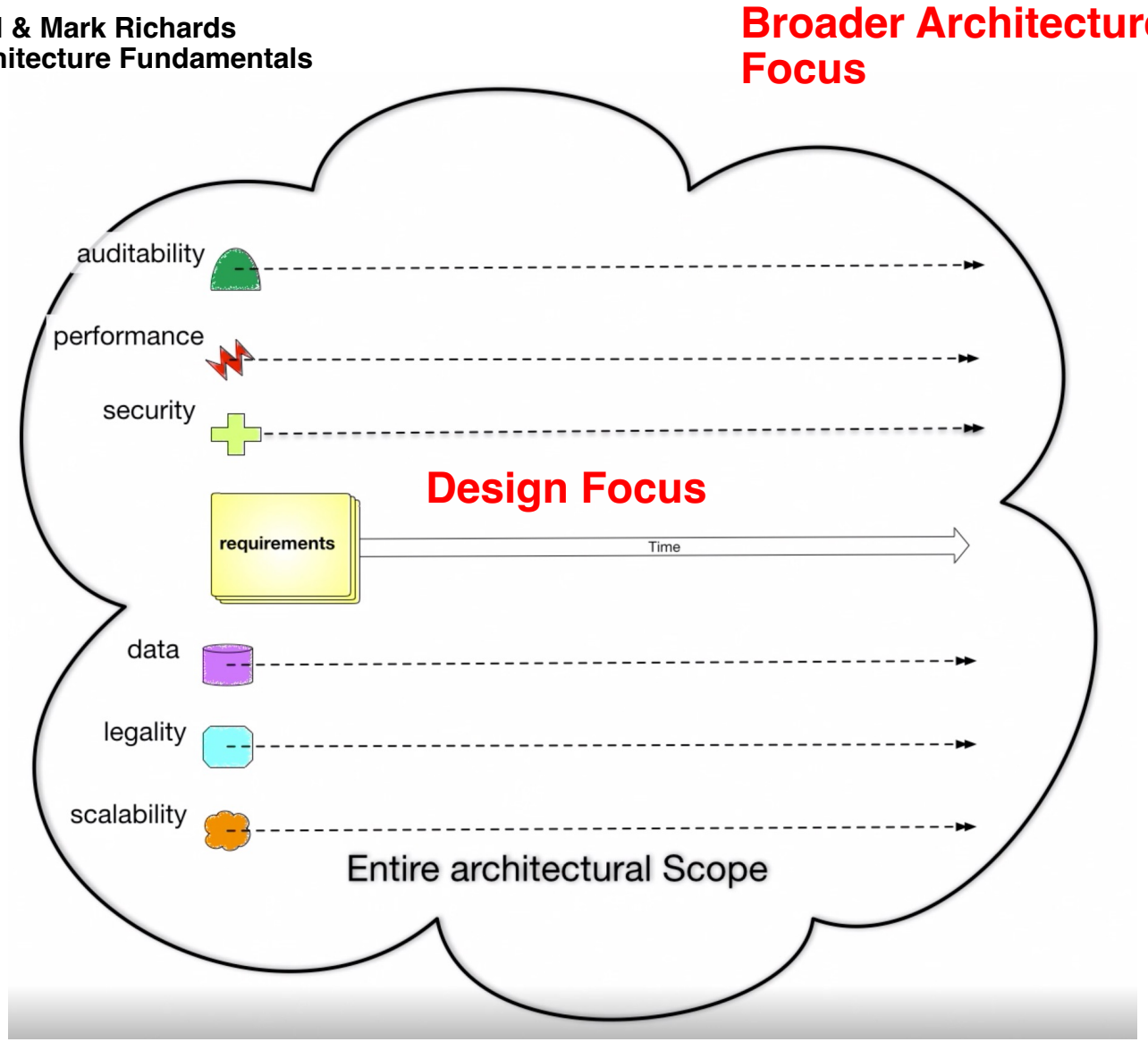
We can also look at this non- hierarchical… The relation between the architecture and design spaces

Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals

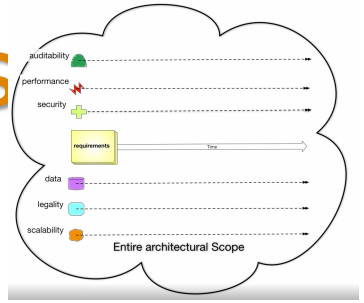


Architecture has a broader scope than design

Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals



System Quality Attributes



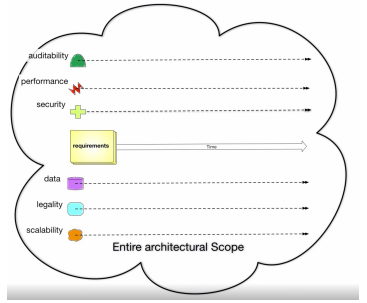
Ref: https://en.wikipedia.org/wiki/List_of_system_quality_attributes

- accessibility
- accountability
- accuracy
- adaptability
- administrability
- affordability
- agility
- auditability
- autonomy
- availability
- compatibility
- composability
- configurability
- correctness
- credibility
- customizability
- debugability
- degradability
- determinability
- demonstrability
- dependability
- deployability
- discoverability [Erl]
- distributability
- durability
- effectiveness
- efficiency
- evolvability
- extensibility
- failure transparency
- fault-tolerance
- fidelity
- flexibility
- inspectability
- installability
- integrity
- interchangeability
- interoperability [Erl]
- learnability
- maintainability
- manageability
- mobility
- modifiability
- modularity
- operability
- orthogonality
- portability
- precision
- predictability
- process capabilities
- producibility
- provability
- recoverability
- relevance
- reliability
- repeatability
- reproducibility
- resilience
- responsiveness
- reusability [Erl]
- robustness
- safety
- scalability
- seamlessness
- self-sustainability
- serviceability securability
- simplicity
- stability
- standards compliance
- survivability
- sustainability
- tailorability
- testability
- timeliness
- traceability
- transparency
- ubiquity
- understandability
- upgradability
- vulnerability
- usability

These attributes and the tradeoffs associated with them influence architecture and design decisions

System Quality Attributes

Ref: https://en.wikipedia.org/wiki/List_of_system_quality_attributes

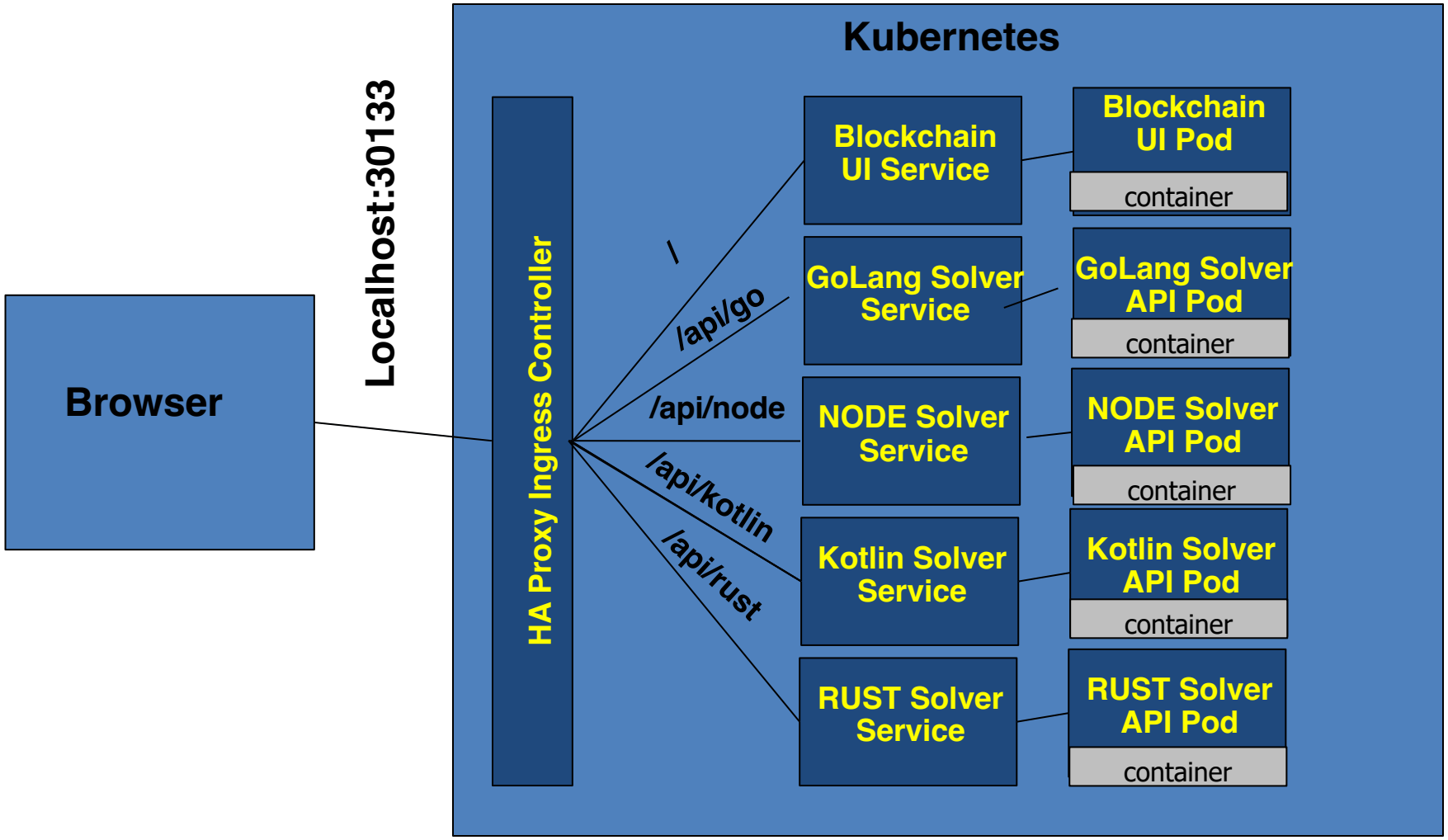


Although there are many quality attributes, certain ones tend to appear again and again that impact the design and architecture of systems

- **Modularity**
- **Maintainability**
- **Testability**
- **Availability**
- **Deployability**
- **Reliability**
- **Scalability**
- **Evolvability**
- **Affordability**

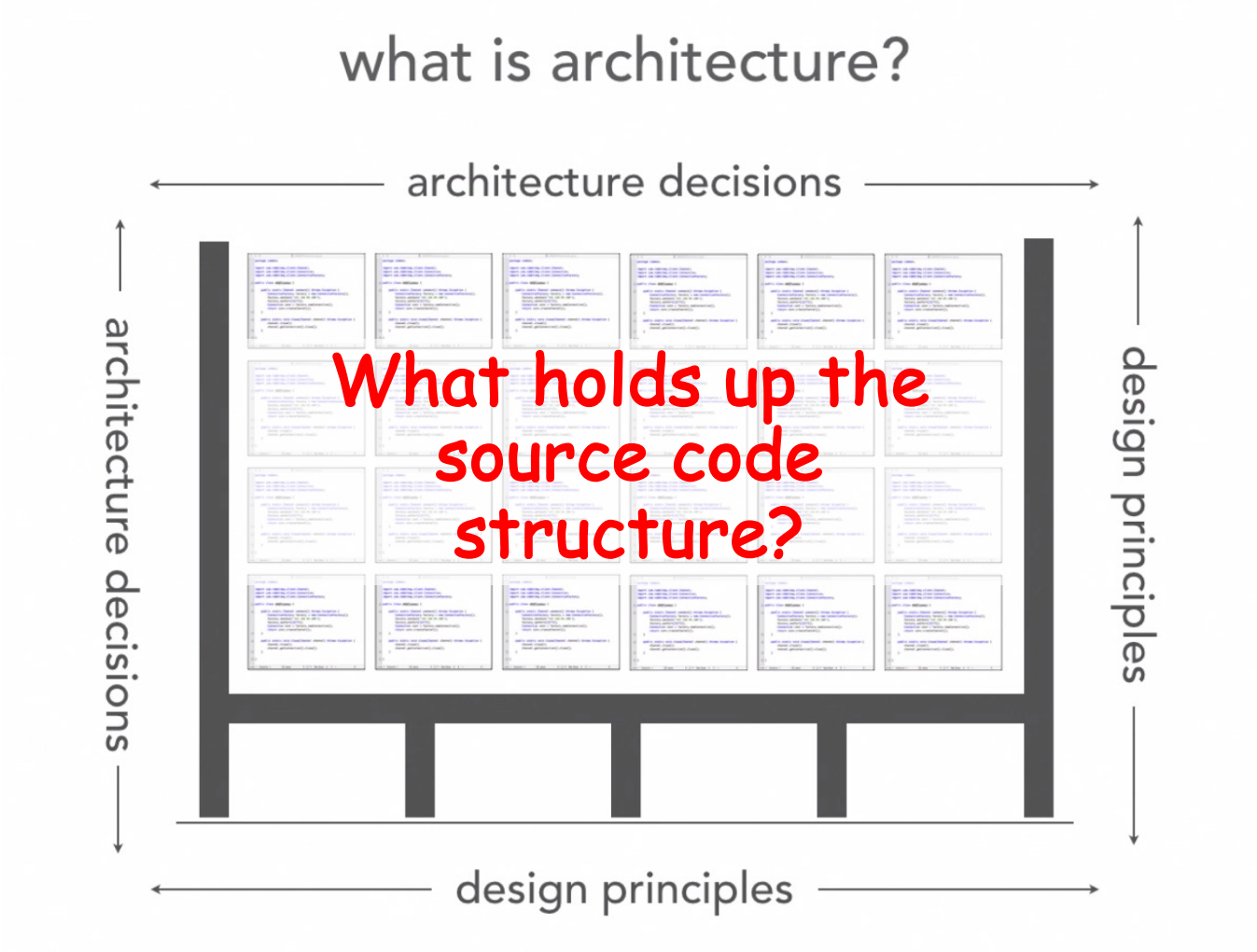
++ FEASIBILITY ++

Blockchain Demo – Quality Attributes and Design



The structure of the source code should not be determined arbitrarily

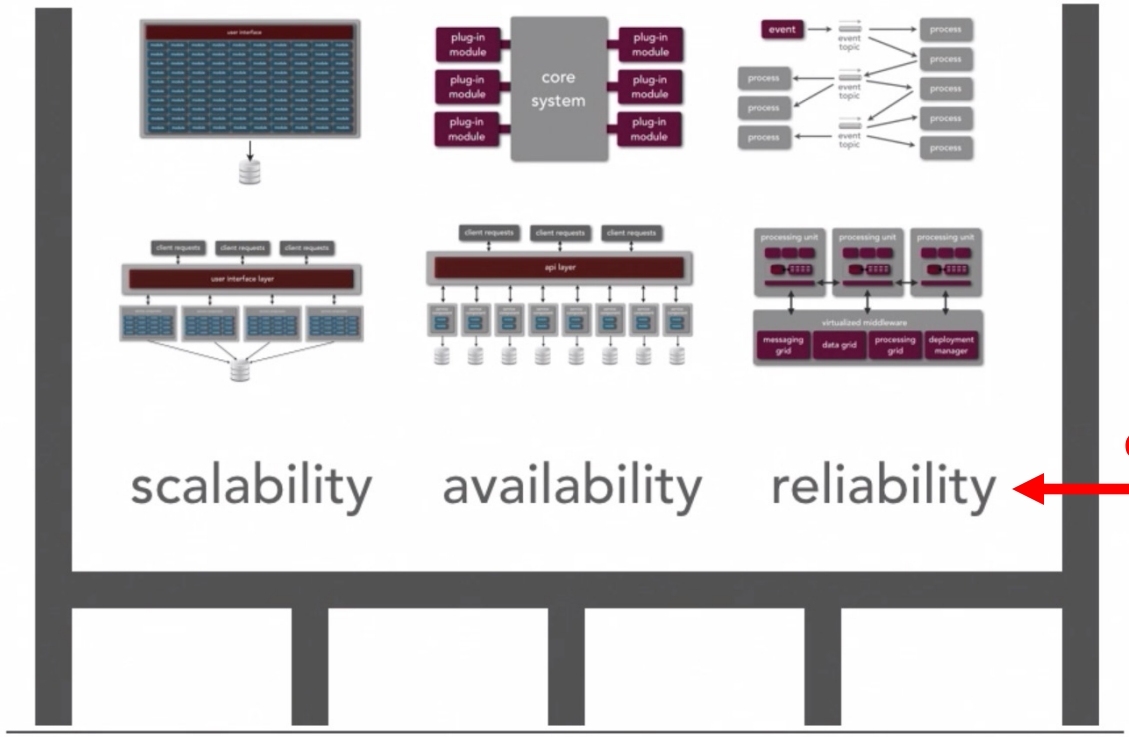
Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals



Starting with understanding the structure of the system

Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals

We can use architecture styles and patterns to structure the system



Needed quality attributes will also help dictate the system structure

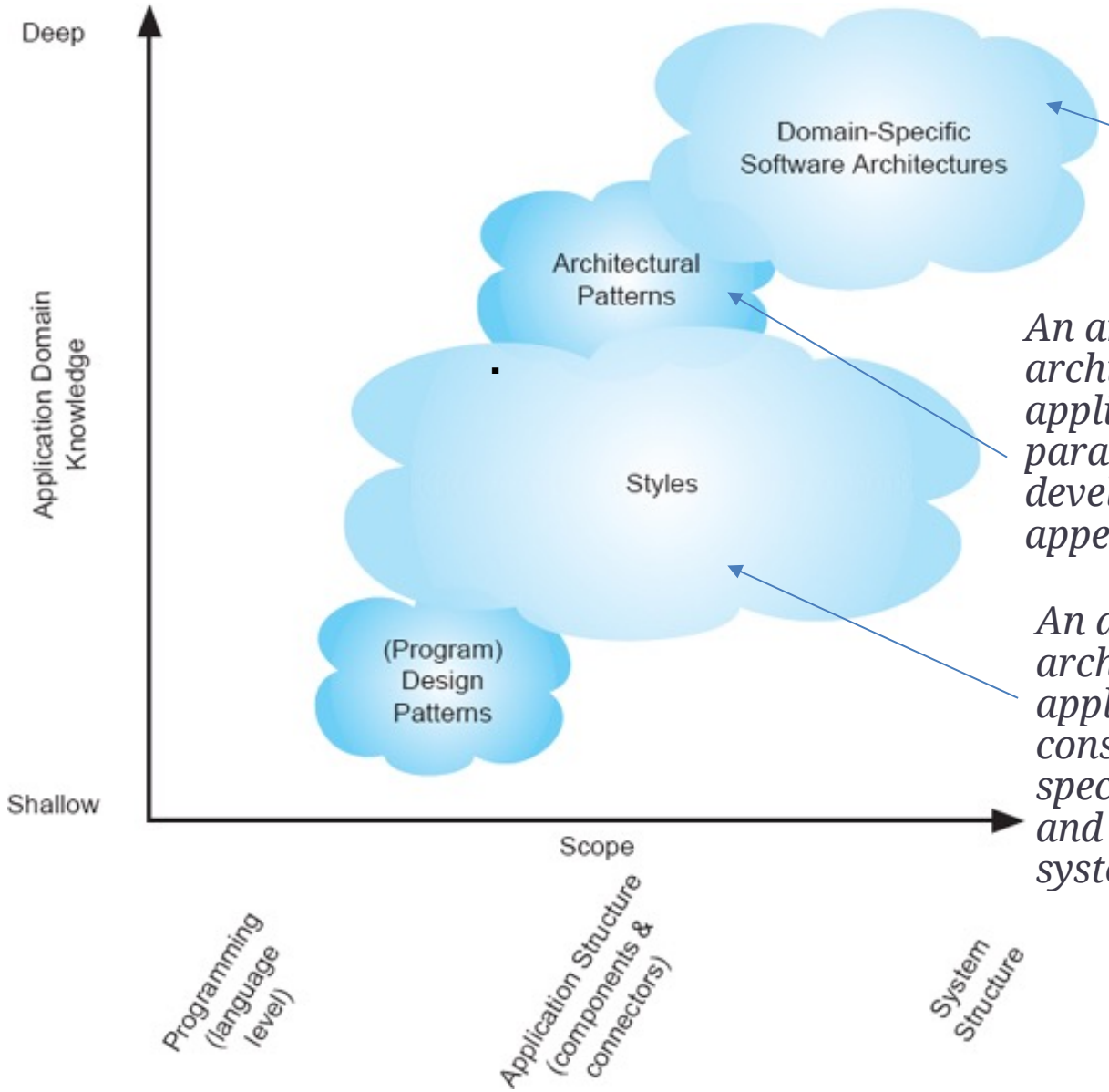
Back to Architecture Basics

- When I introduced the vocabulary of architecture components
  - Terms used: Components, Connectors, Structures, and so on.
- I like to think about architecture in terms of laying out the foundation for design
  - What **design decisions** do we need to make, and/or what **structures do we need to document** in order to realize all of the constraints imposed on the system
  - Constraints typically come from the non-functional requirements – time to market, budget, technology standards, skillsets, etc
- Good architecture makes important decisions early and defers less-important decisions to later
  - Do we really need to pick the database technology up front?
  - How much should the fact that the system is web-based influence the overall design, can we abstract this for now and specify it later?

Architecture and Design Patterns

- Architecture Style
  - Defines the **vocabulary** of the components and connectors in a software architecture and the **constraints** on how they can be combined.
  - Example: Pipe and Filter
- Architecture Patterns
  - Focus is on the organization of the overall system
  - An instance of an architecture style
- Design Patterns
  - Focus is on the organization of a block of code

Software Architecture Landscape



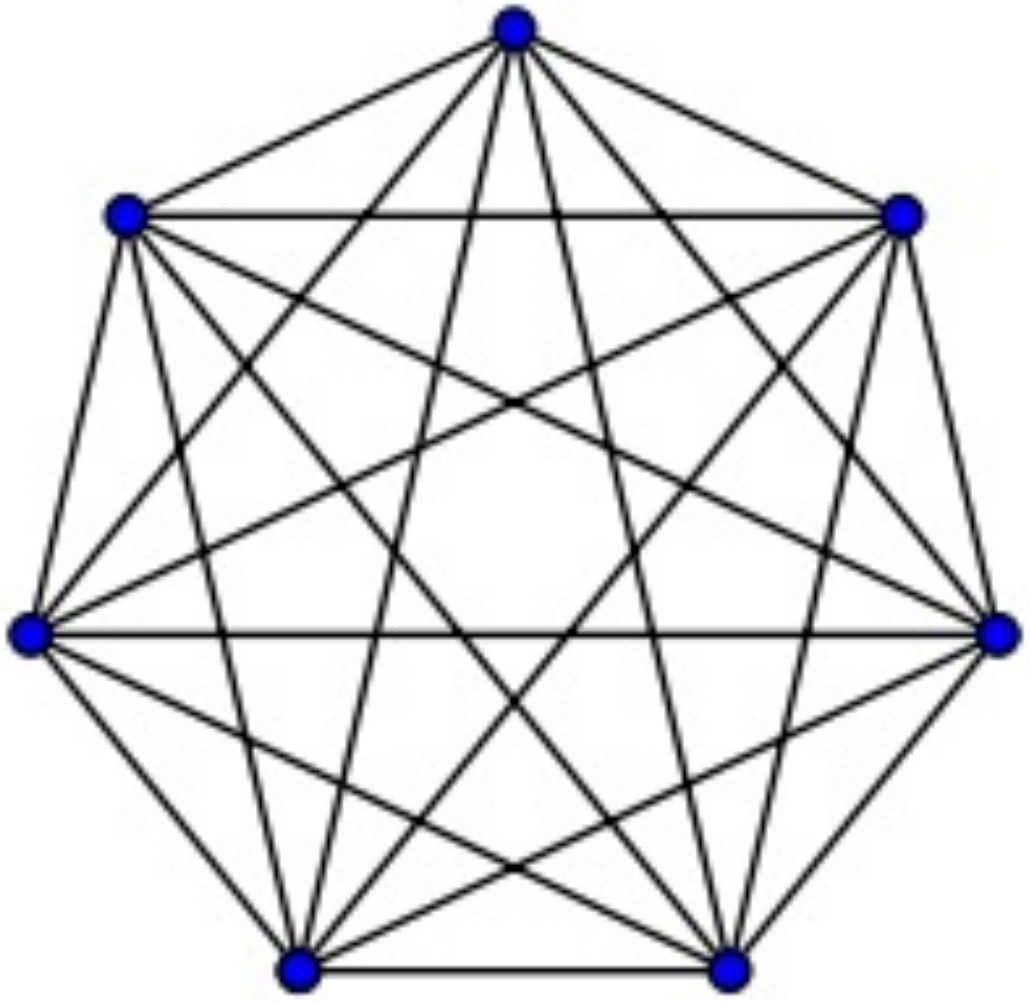
Also known as a reference architecture, is the set of principal design decisions that are simultaneously applicable to multiple related systems, typically within an application domain, with explicitly defined points of variation

An architectural pattern is a named collection of architectural design decisions that are applicable to a recurring design problem, parameterized to account for different software development contexts in which that problem appears.

An architectural style is a named collection of architectural design decisions that (1) are applicable in a given development context, (2) constrain architectural design decisions that are specific to a particular system within that context, and (3) elicit beneficial qualities in each resulting system.

R.N. Taylor, N. Medvidovic, and E.M. Dashofy. *Software Architecture: Foundations, Theory and Practice*, Wiley.

Remember every system has an architecture – Our first architecture pattern

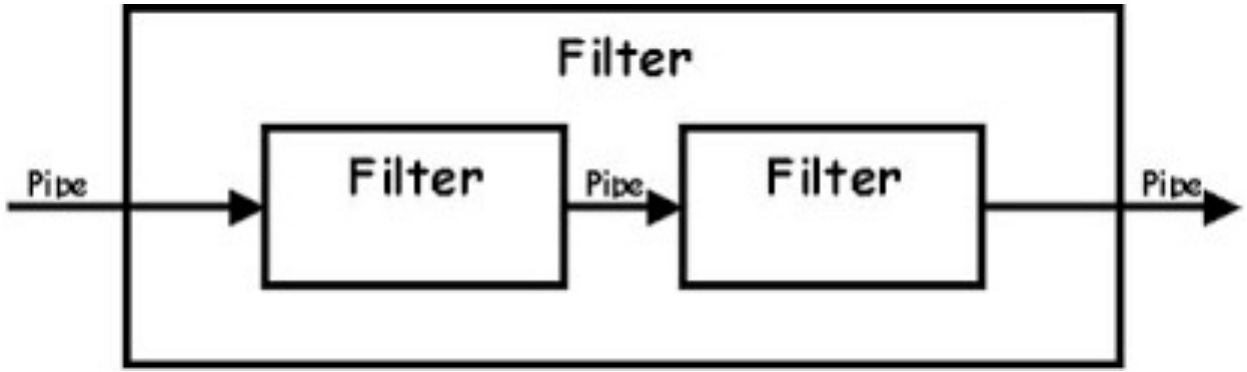


Lets call this the "ball of mud" architecture pattern – it meets the definition

Constraints of this pattern – every component is allowed to directly talk to every other component – in other words – there are no constraints

Thus, introducing a new component, allows for an additional n-1 connections where n is the total number of components

Another Architecture Pattern – this one a little more realistic



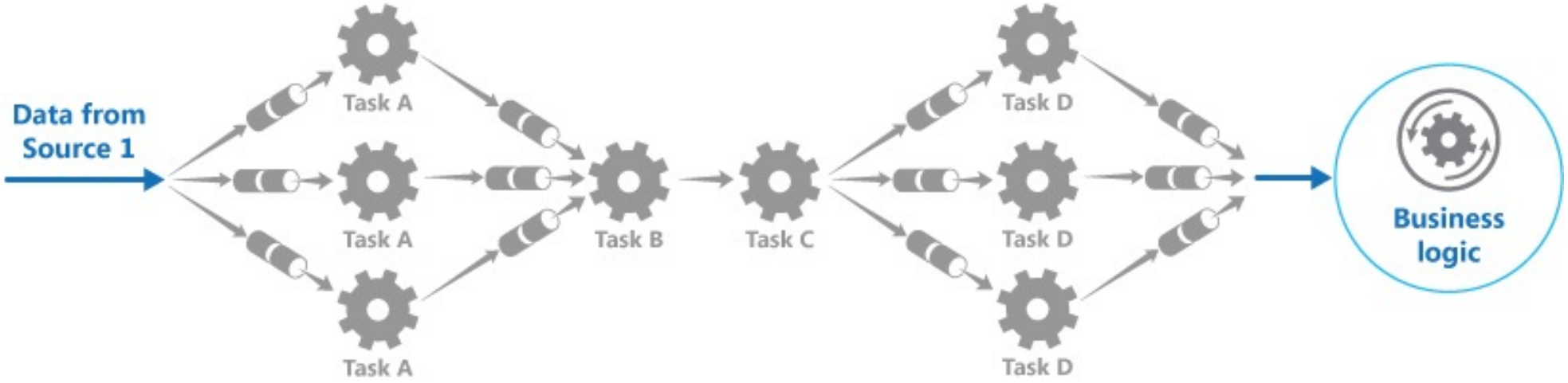
The Pipe-and-Filter is a classic architecture pattern – its actually the basis for some of the goodness in functional programming and other design patterns such as the builder pattern

Components are the filters – all inputs are provided by pipes, all transformations are done by filters.

Filters are stateless, all transformations are based on pipe inputs

The filters can ingest inputs and produce outputs atomically or via streaming

Another Architecture Pattern – this one a little more realistic



The Pipe-and-Filter is also powerful because there can be special filters that split data and other filters that coordinate merging of data

This enables large scale and concurrency

Pipe and Filter is very useful in the real world

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

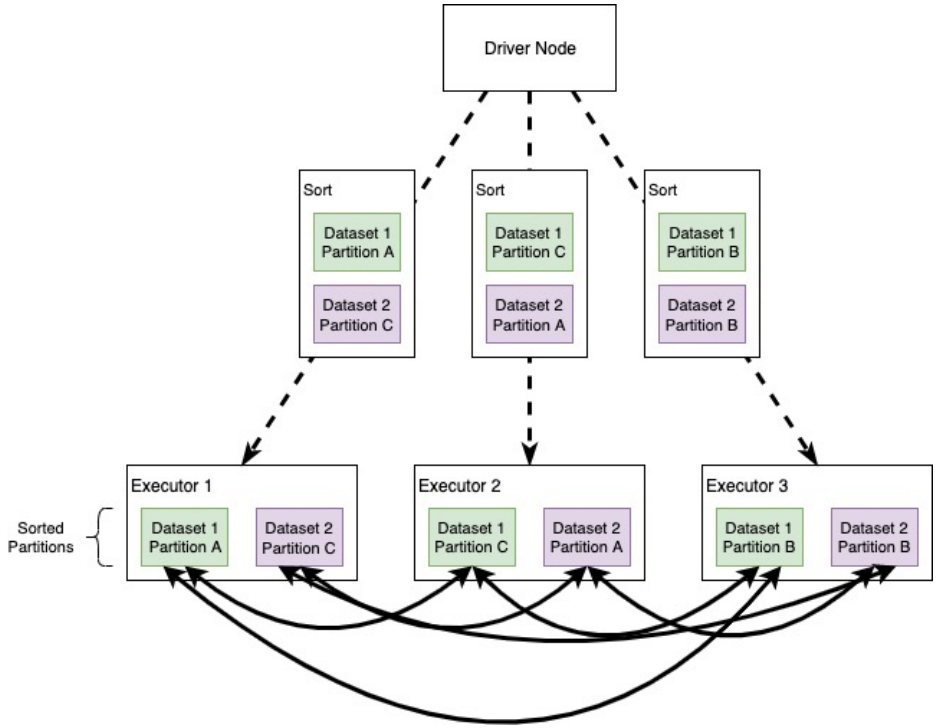
given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

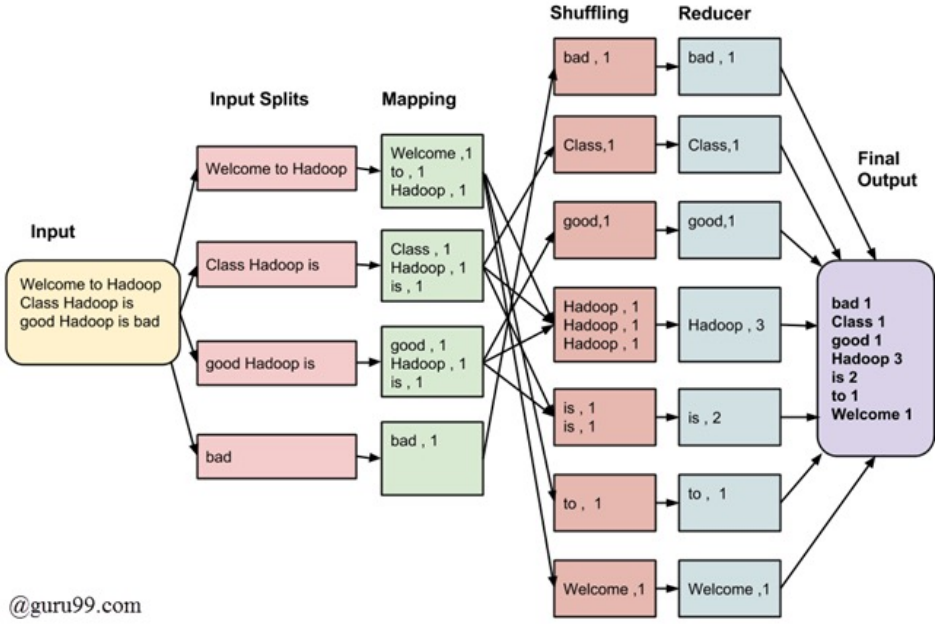
As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined

This is the seminal paper by google on MapReduce – basically an at scale implementation of pipe-and-filter

Pipe-and-Filter used as at scale data pipelines





Pipe-and-Filter – Functional Programming

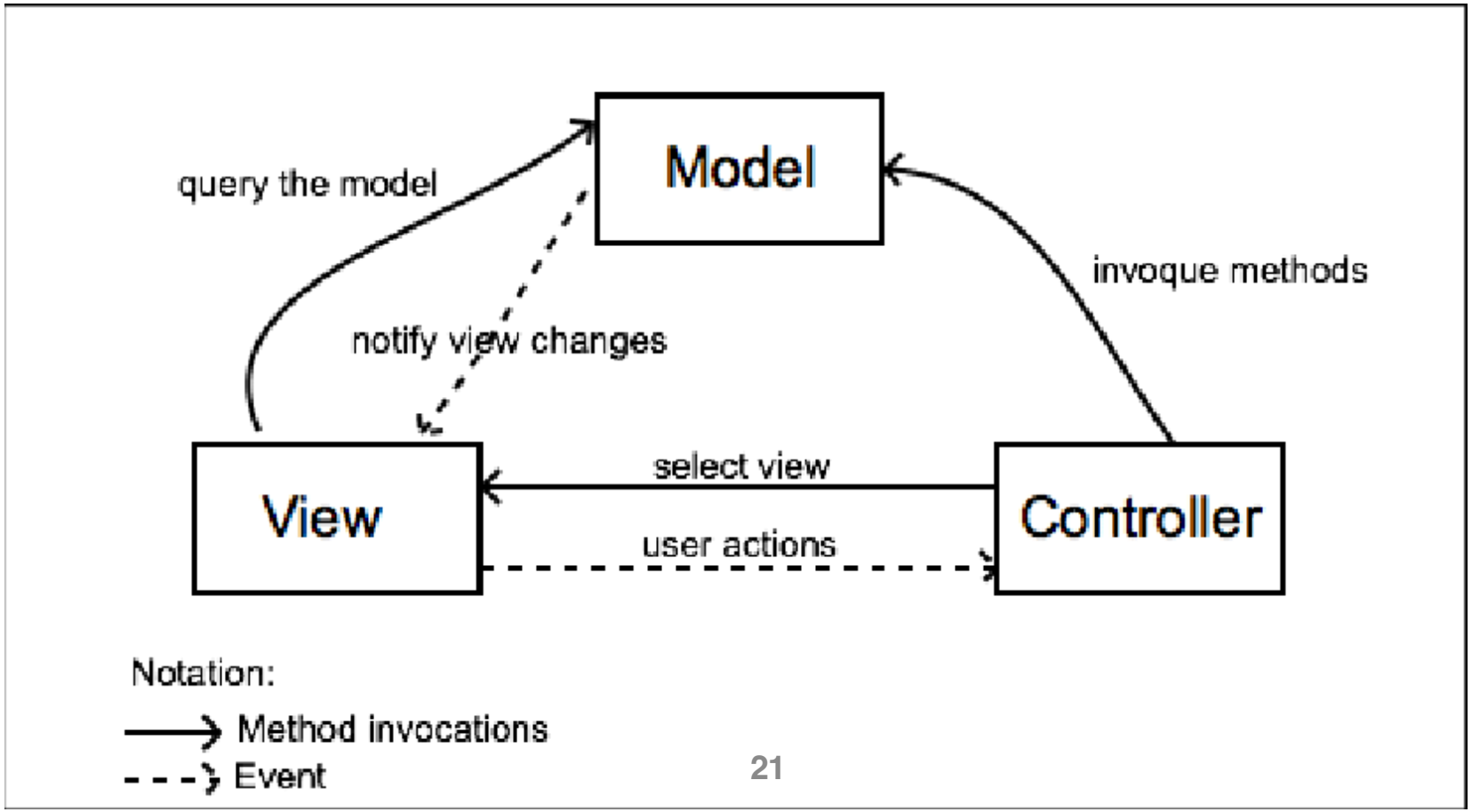
Demo at: https://www.learnrxjs.io/learn-rxjs/recipes/type-ahead

```
const getContinents = keys =>
  [
    'africa',
    'antarctica',
    'asia',
    'australia',
    'europe',
    'north america',
    'south america'
  ].filter(e => e.indexOf(keys.toLowerCase()) > -1);

const fakeContinentsRequest = keys =>
  of(getContinents(keys)).pipe(
    tap(_ => console.log(`API CALL at ${new Date()}`))
  );

fromEvent(document.getElementById('type-ahead'), 'keyup')
  .pipe(
    debounceTime(200),
    map((e: any) => e.target.value),
    distinctUntilChanged(),
    switchMap(fakeContinentsRequest),
    tap(c => (document.getElementById('output').innerText = c.join('\n')))
  )
  .subscribe();
```

An example of an architecture pattern – MVC (Model/View/Controller)

MVC is a more realistic example of an architecture style

- View is passive, reacts to changes – updates data based on model events, updates presentation based on controller reacting to user behavior
- Application state fully managed by model



Sidebar – Architecture Style versus Architecture Pattern?

If you study classic references for software architecture you will see that architecture styles and patterns are considered different things…

While we can form a general understanding of their differences based on abstraction (e.g., pipe/filter vs MVC)…

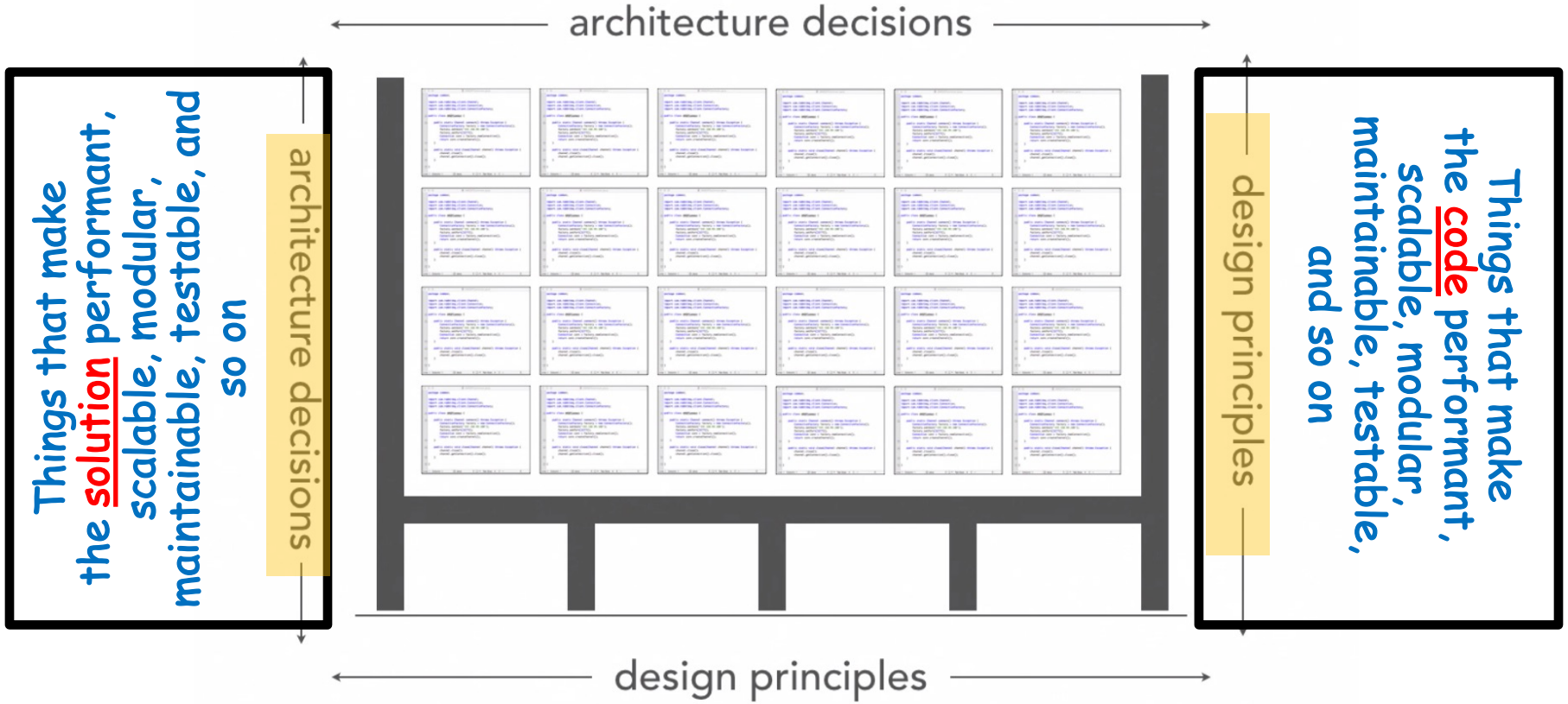
In practice, they can generally be used interchangeably – Ill generally just call them architecture styles

Architecture Styles

- Architecture Styles **Define** and **Constrain** the:
  - The **components** of the solution
    - What are the major computational units of the solution
    - Eg., Security, Data Access, Protocol Handling, …
  - The **connectors** of the solution
    - How are the interactions between the components realized
    - Eg., procedure calls, network calls, events, broadcasts, …
  - The **properties** of the solution
    - What are the quality attributes and non-functional requirements
    - Eg., pre/post conditions, availability specifications, interface semantics, …

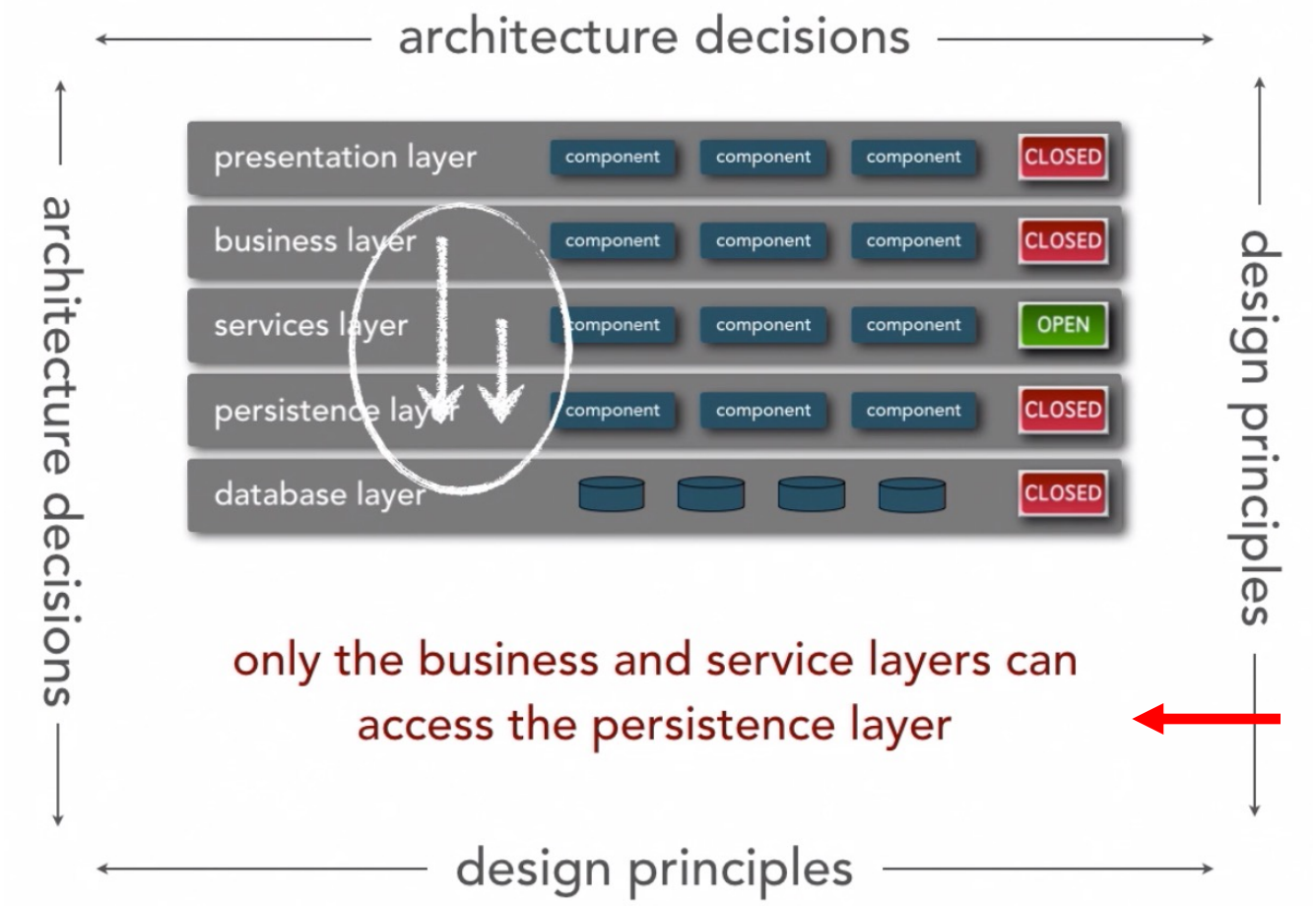
Architecture decisions and design principles also influence the system structure

Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals



Architecture principal example

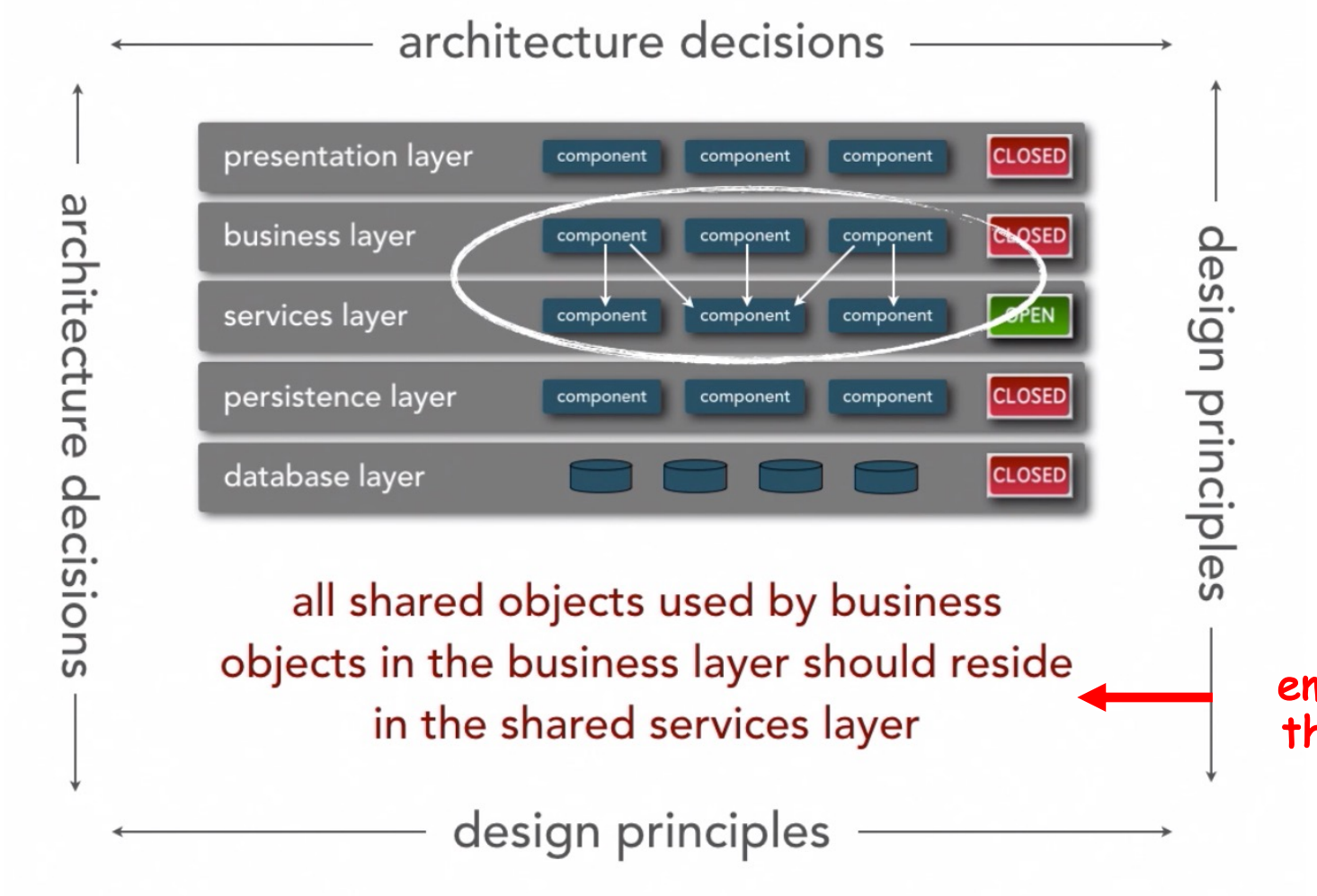
Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals



Reduces coupling by only allowing components to communicate to adjacent layers or within the same layer

Another architecture principal example…

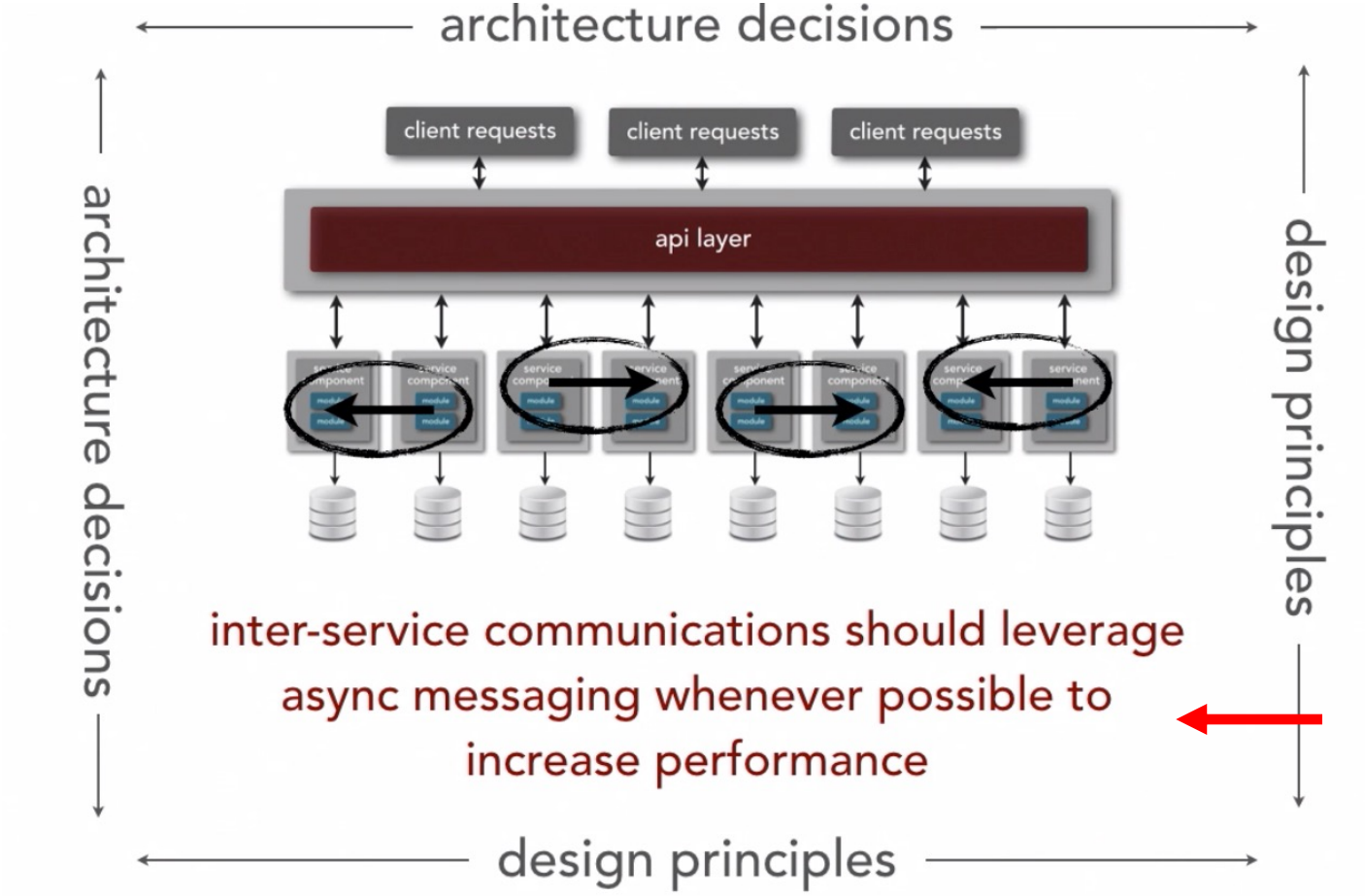
Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals



Implies reuse emphasis will be in the services layer

Design principles can impact many platform and technology aspects

Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals

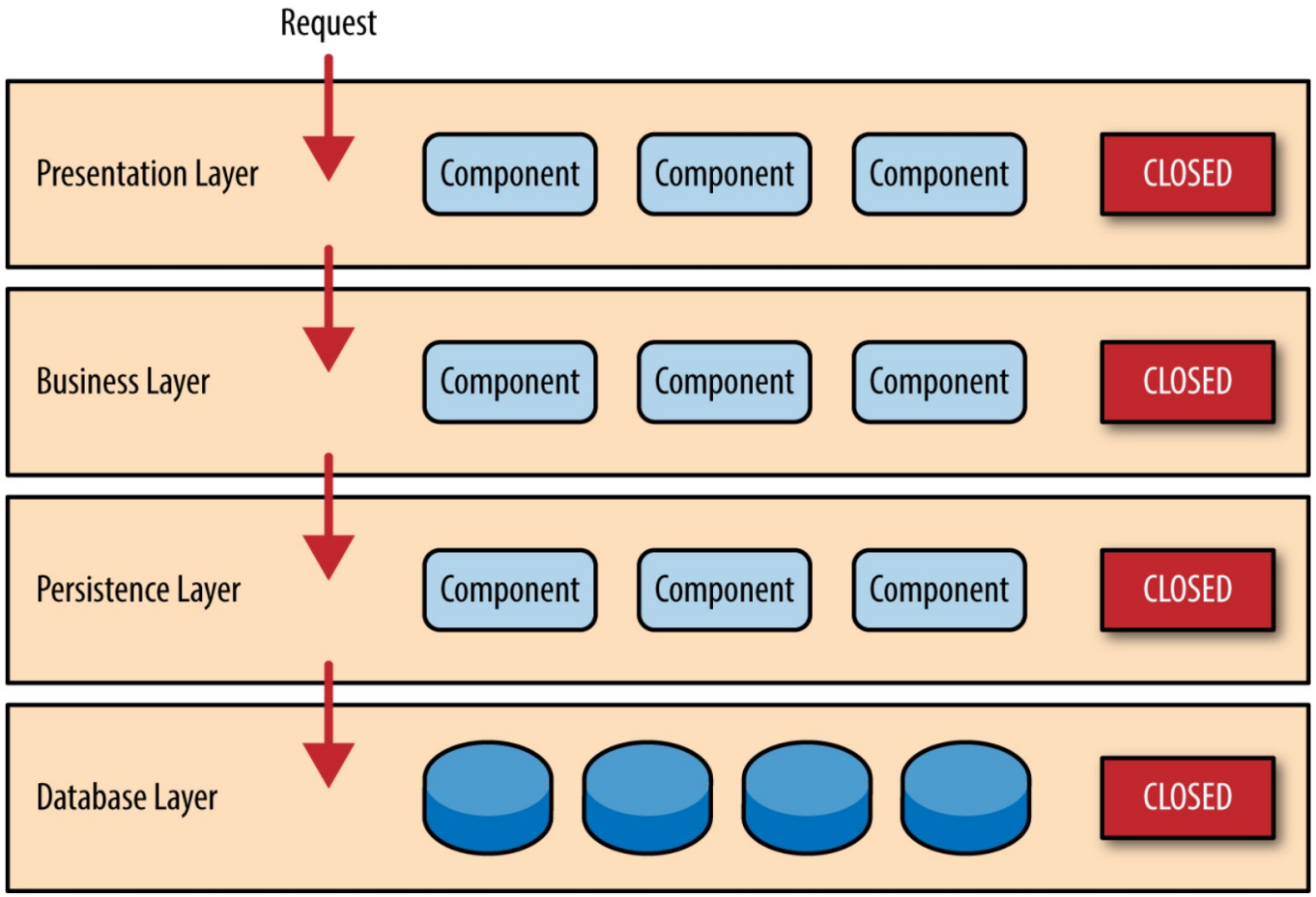


A design decision that has been shown to have good performance characteristics… Tradeoff is introduction of complexity

Back to Architecture

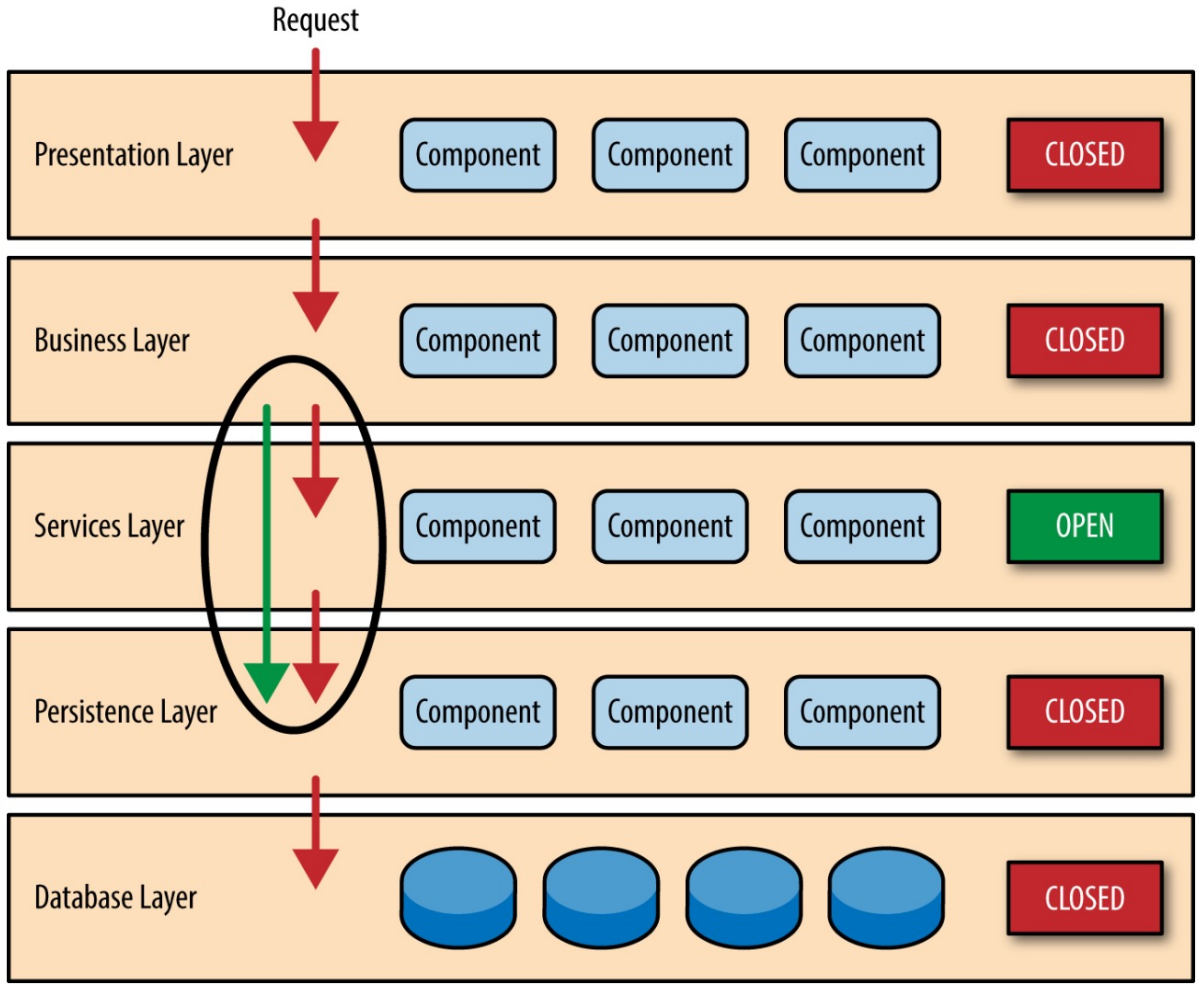
- The previous definitions presented focus on the vocabulary of architecture components
  - Terms used: Components, Connectors, Structures, and so on.
- I like to think about architecture in terms of laying out the foundation for design
  - What **design decisions** do we need to make, and/or what **structures do we need to document** in order to realize all of the constraints imposed on the system
  - Constraints typically come from the non-functional requirements – time to market, budget, technology standards, skillsets, etc
- Good architecture makes important decisions early and defers less-important decisions to later
  - Do we really need to pick the database technology up front?
  - How much should the fact that the system is web-based influence the overall design, can we abstract this for now and specify it later?

Layered Style



Basic Principle is that you can only interact in one direction with an adjacent layer

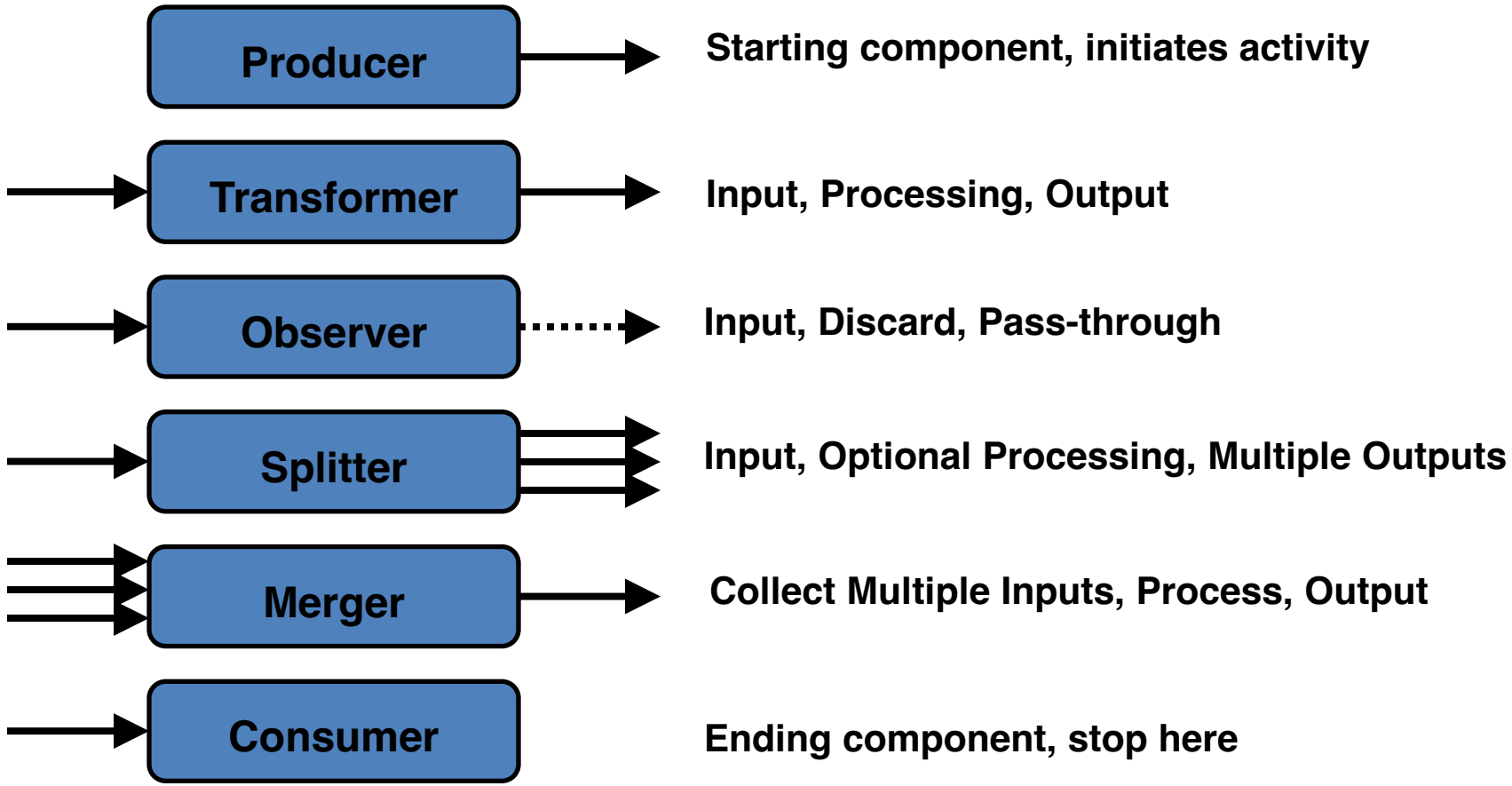
Layered Style



Sometimes Layers are Open To Bypassing

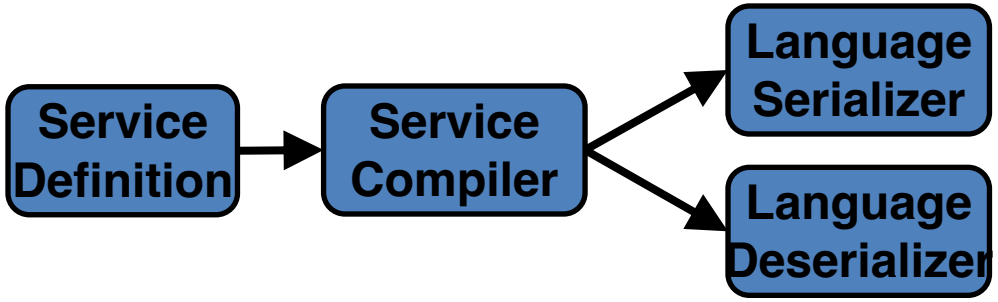
Pipe and Filter

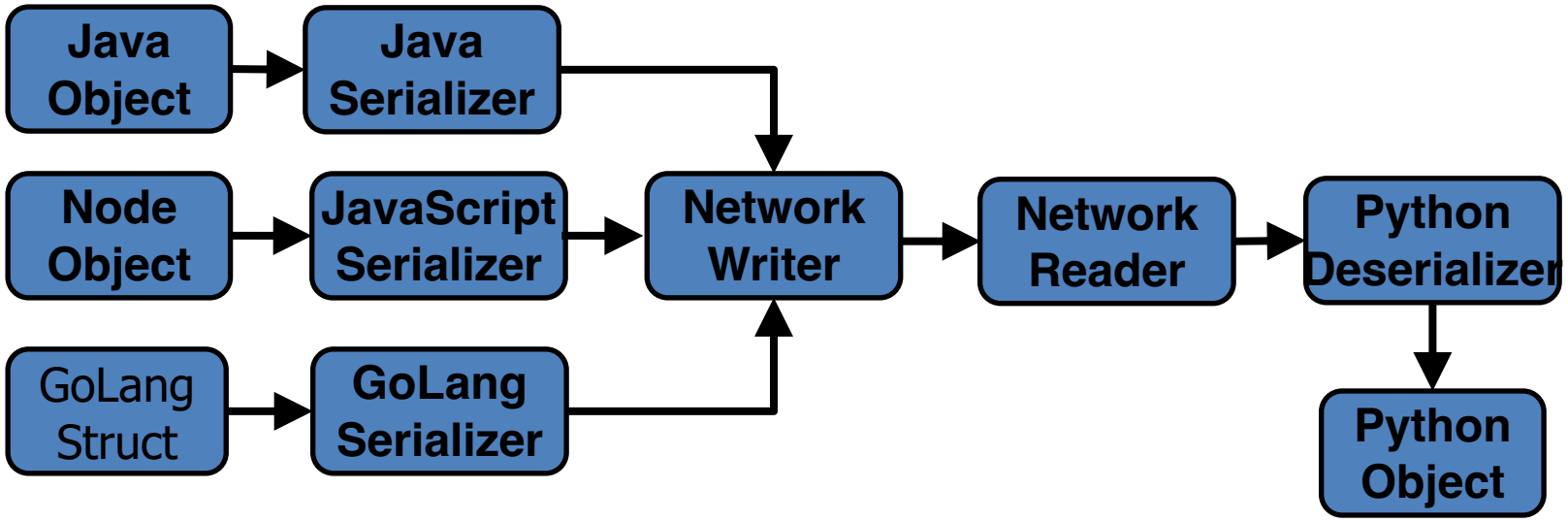
The pipe & filter architecture style is used to wire together processing components



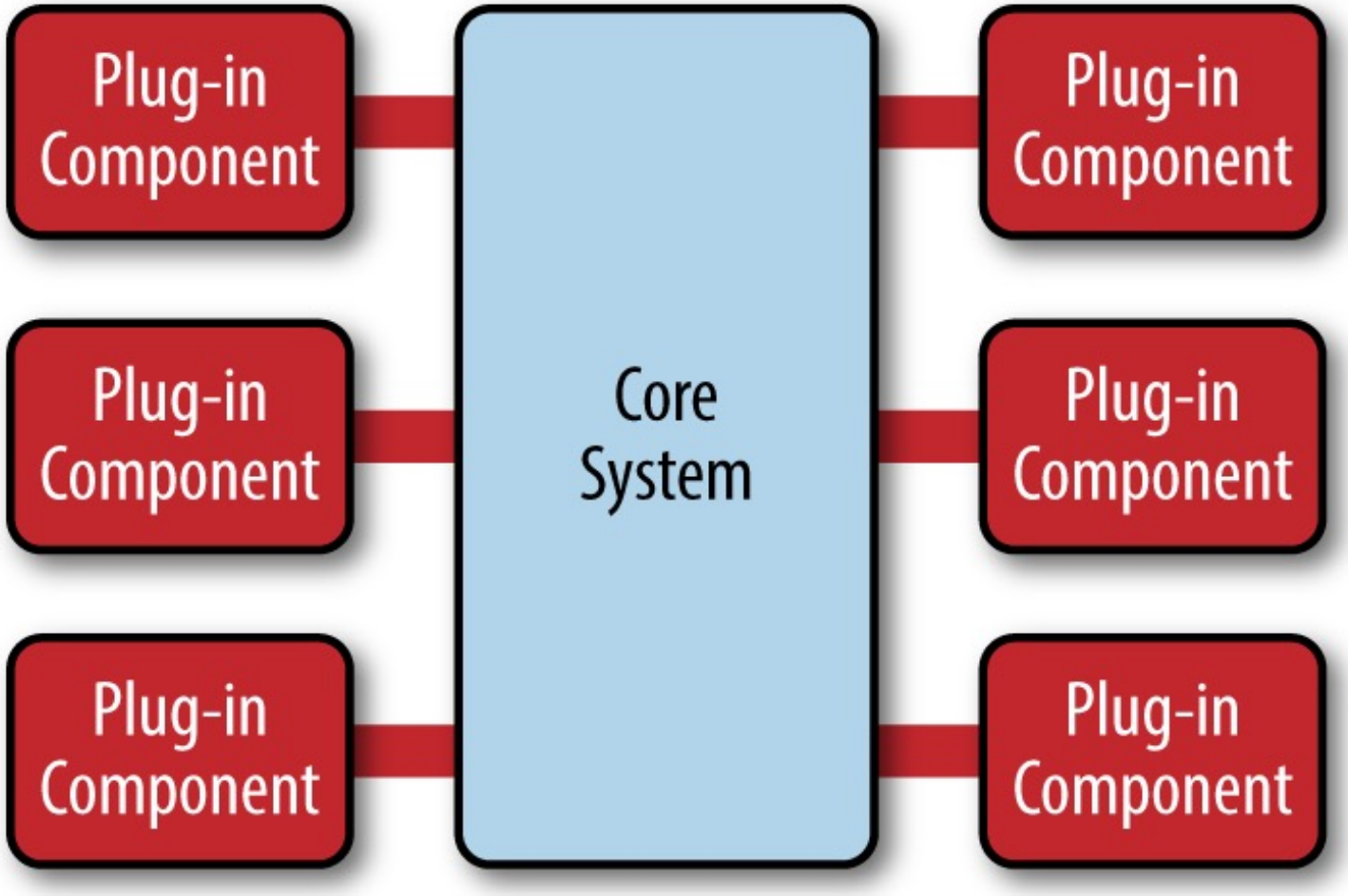
Pipe and Filter

The pipe & filter architecture style is used to wire together processing components





Microkernel Architecture Style



A common architecture style for Software as a Service solutions

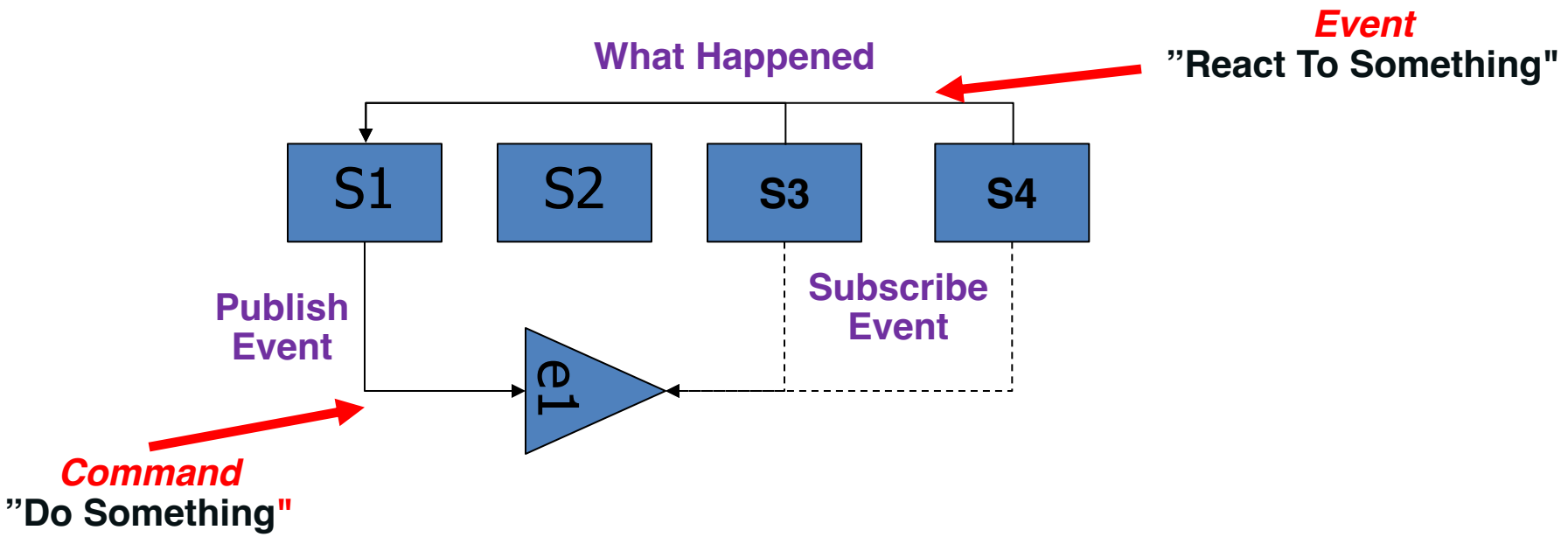
The “Core System” provides the value added functionality out of the box

The “Plug In Component” can be other purchased software specific to an industry vertical (e.g., healthcare, telecom, etc), or custom code taylored to a specific need.

Often-times the core system will prove an API or SDK to enable the plug-in ecosystem

Microkernel Architecture Style - Examples

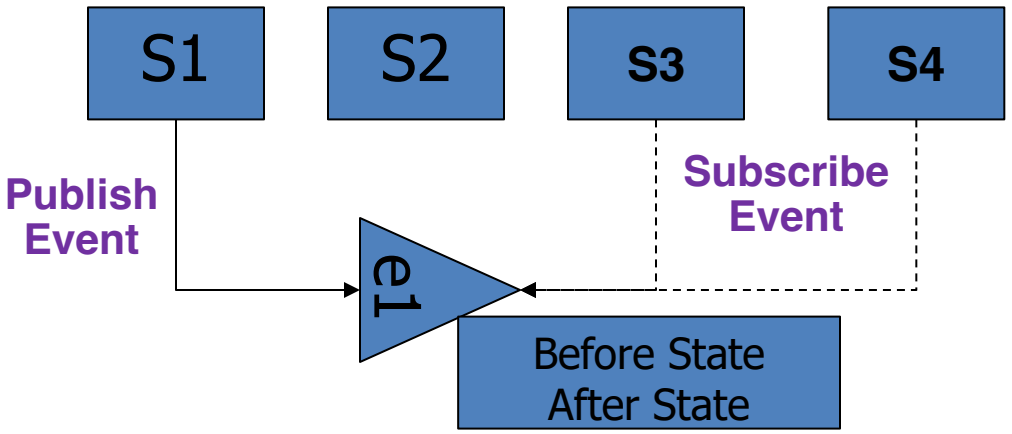
Event Architectures – Event Notification



Event Notification
Loosen coupling with pub/sub patterns
Interest is only in if the event has occurred

Source: The Many Meanings of Event-Driven Architecture – Martin Fowler

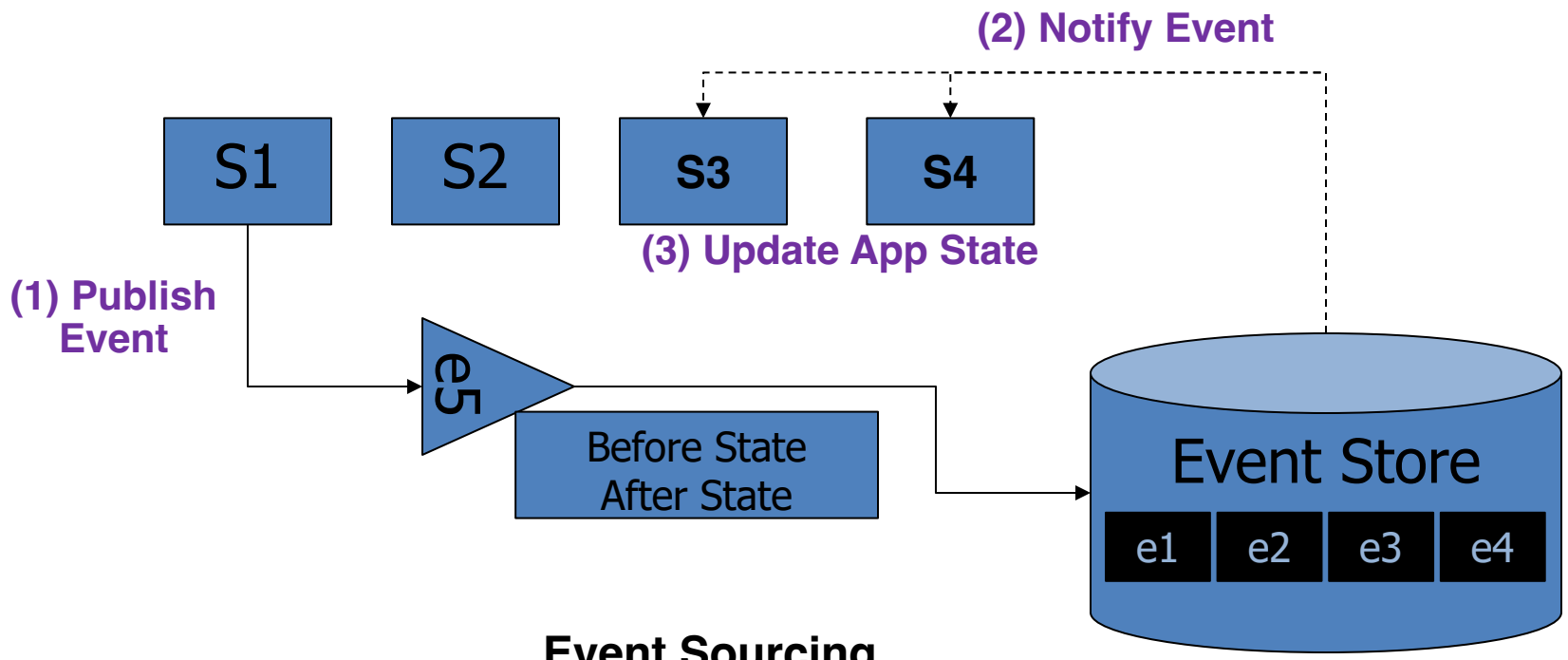
Event and Streaming Architectures



Event Carried State Transfer
Loosen coupling with pub/sub patterns
Eventual Consistency

Source: The Many Meanings of Event-Driven Architecture – Martin Fowler

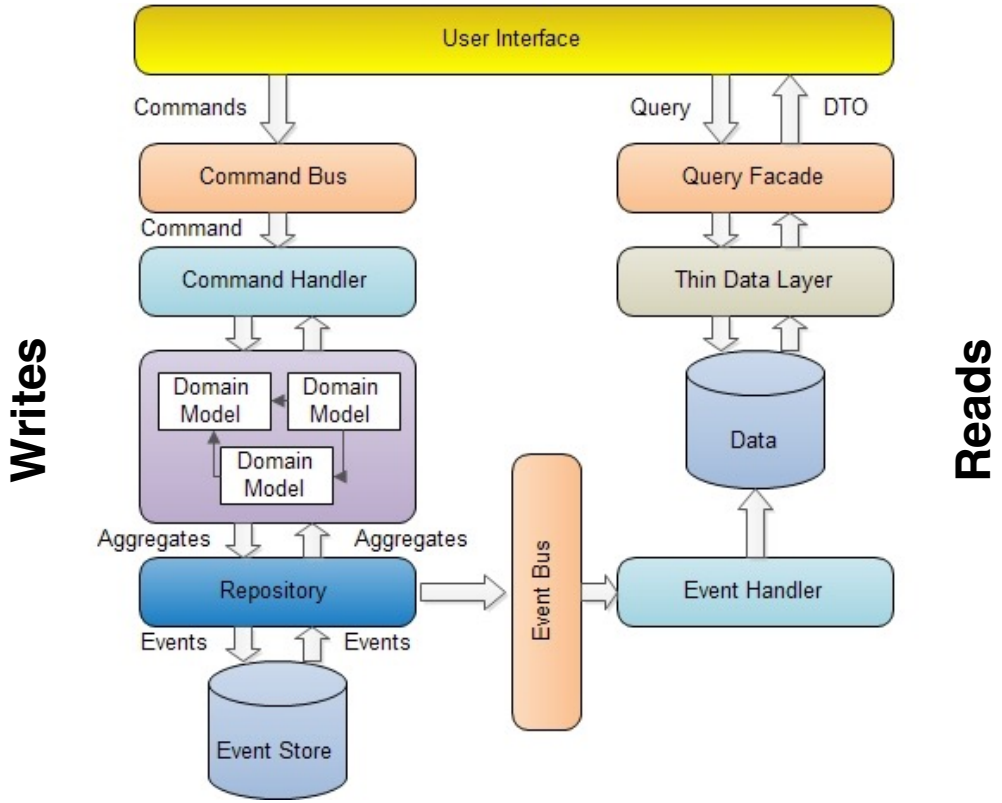
Event – Event Carried State Transfer



Event Sourcing
Event Store Maintains State Over Time
Loosens Coupling

Source: The Many Meanings of Event-Driven Architecture – Martin Fowler

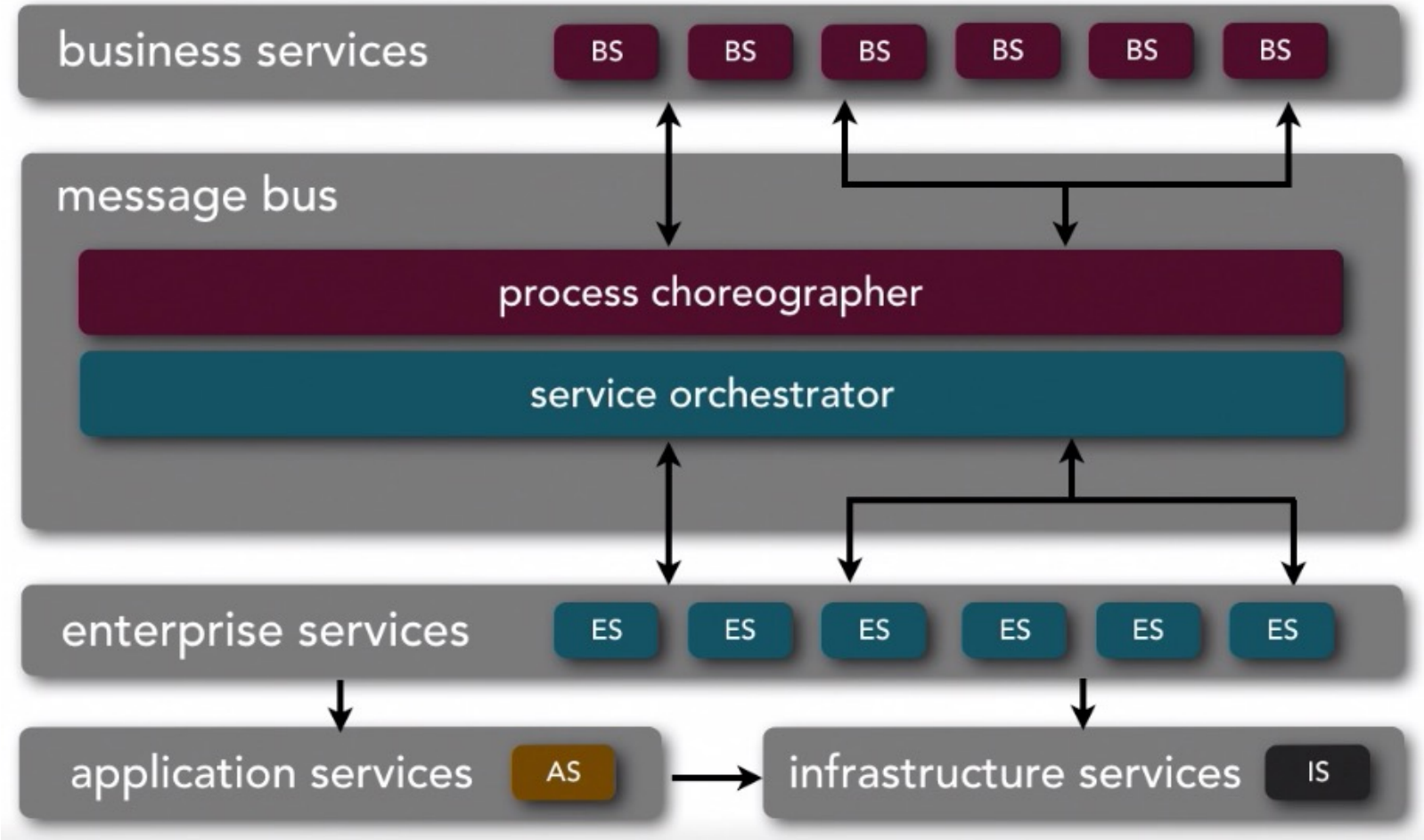
Event – Command Query Responsibility Separation



Scalability and Consistency via Separate Handling of Reads and Writes (CQRS)

Service Oriented Architecture Style

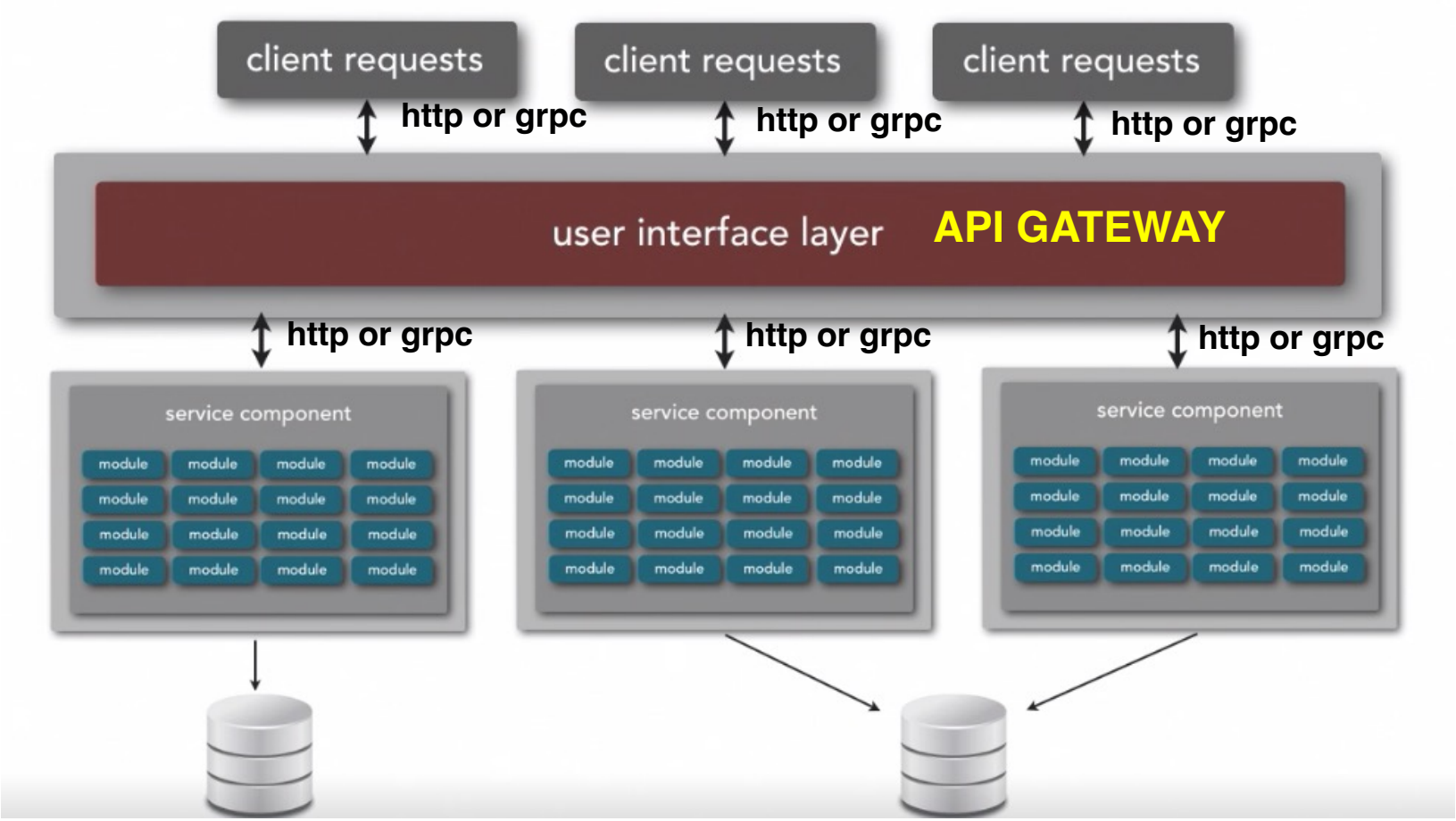
Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals



Thoughts? Good Parts? Bad Parts?

Service Based Style

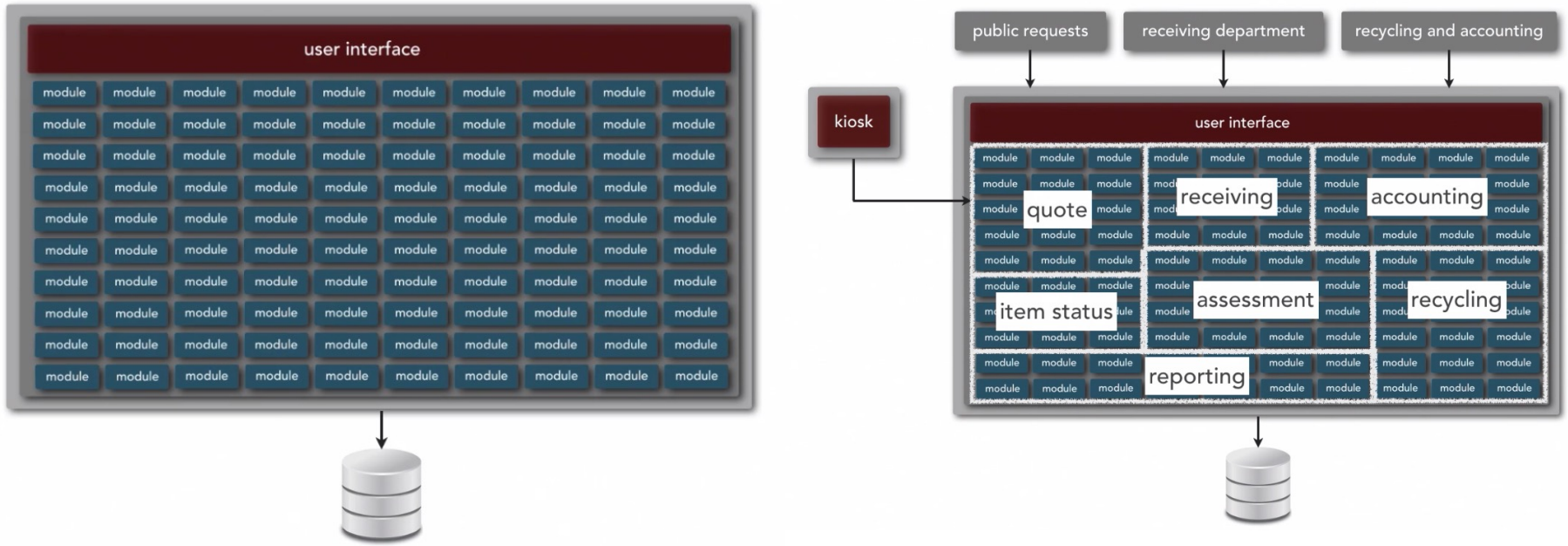
Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals



Works Well To Partition Functionality

Service Based Style - Example

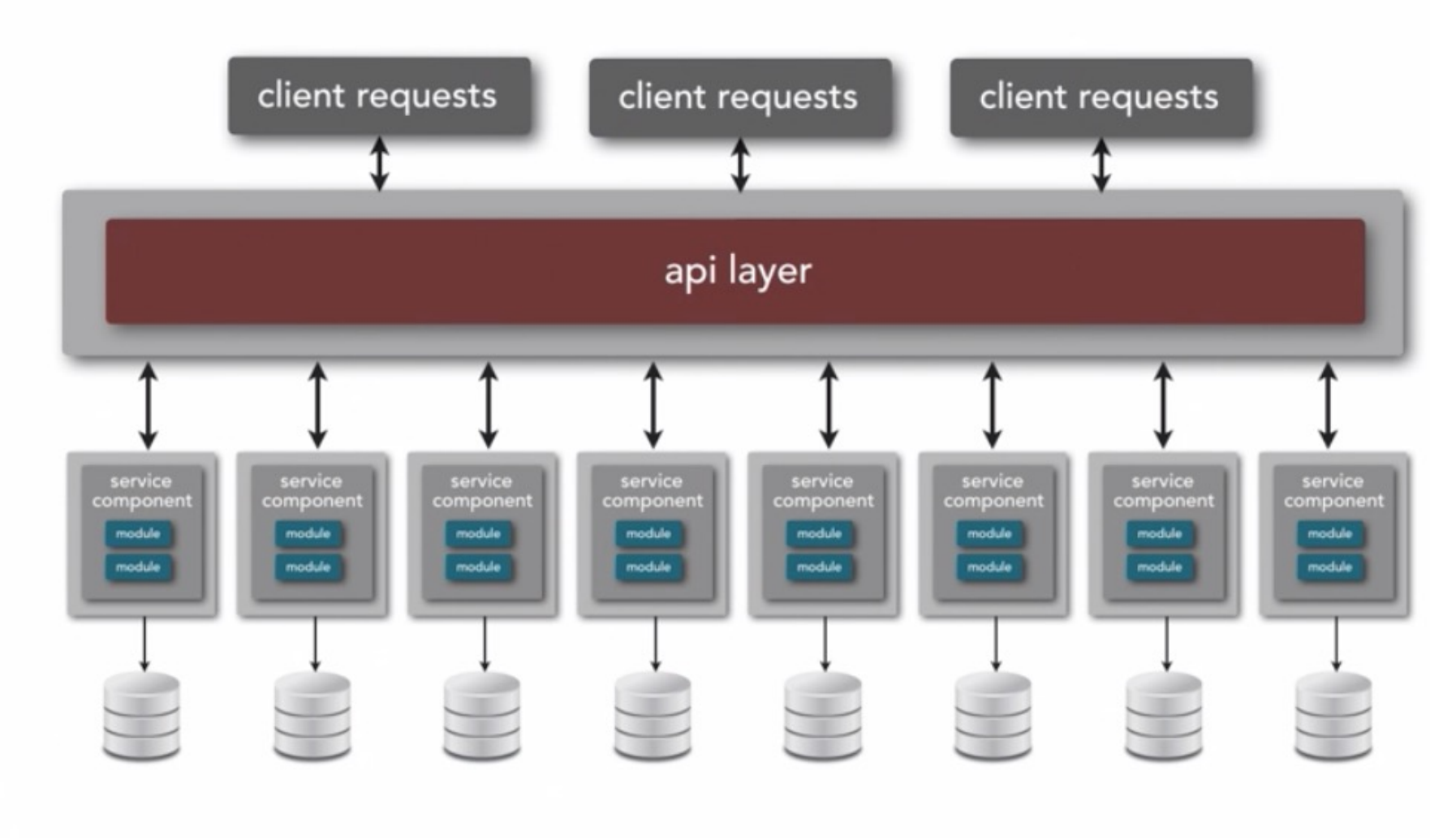
Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals



Generally Good To Decompose a Monolithic Application

Microservice Architecture

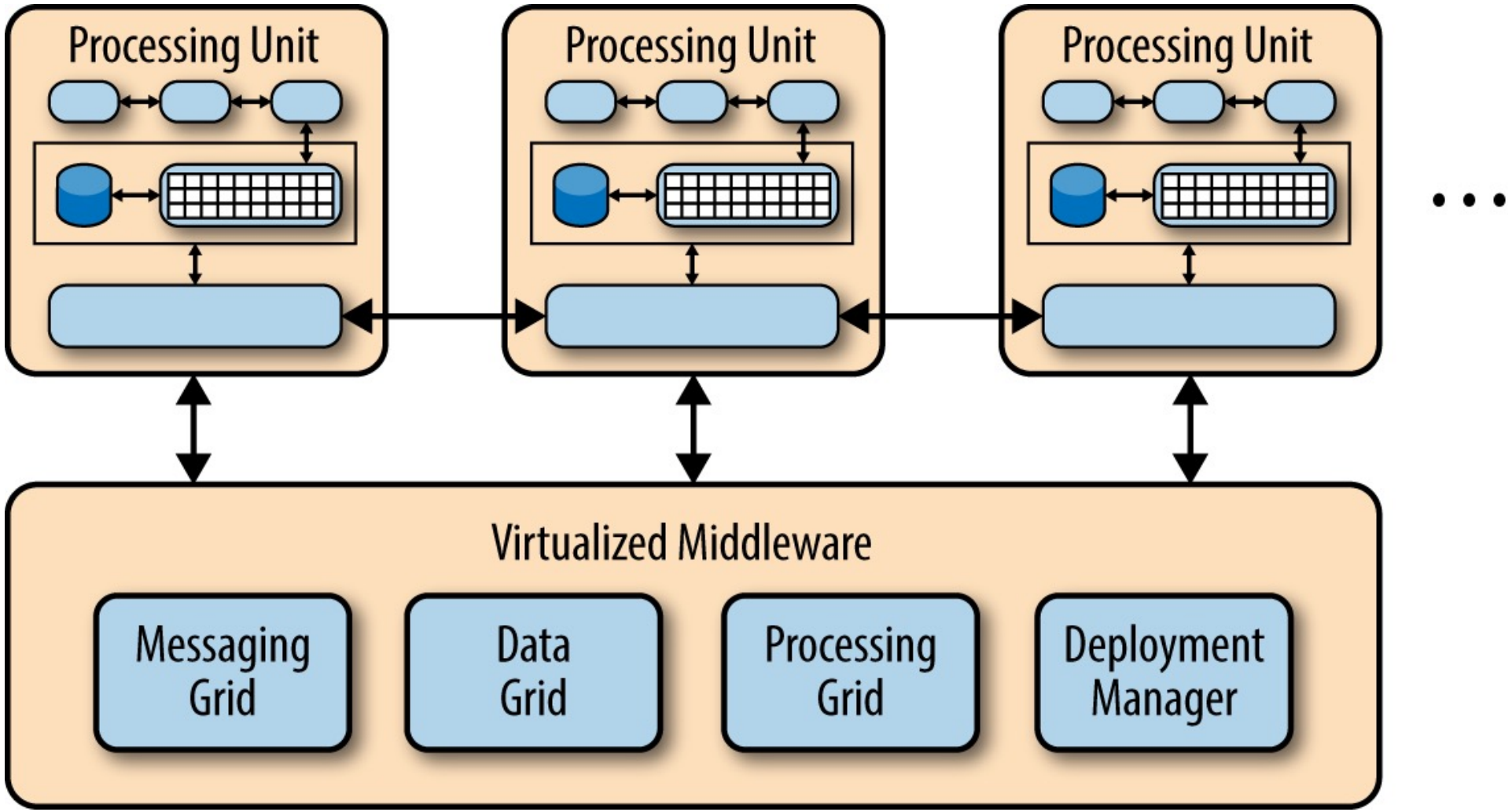
Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals



Basic Idea – SHARE NOTHING

Space-Based Architecture (aka Cloud)

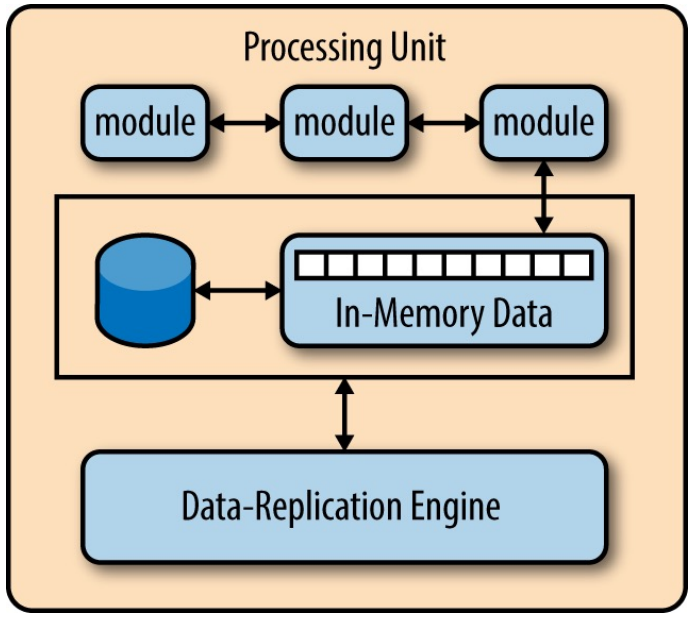
Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals



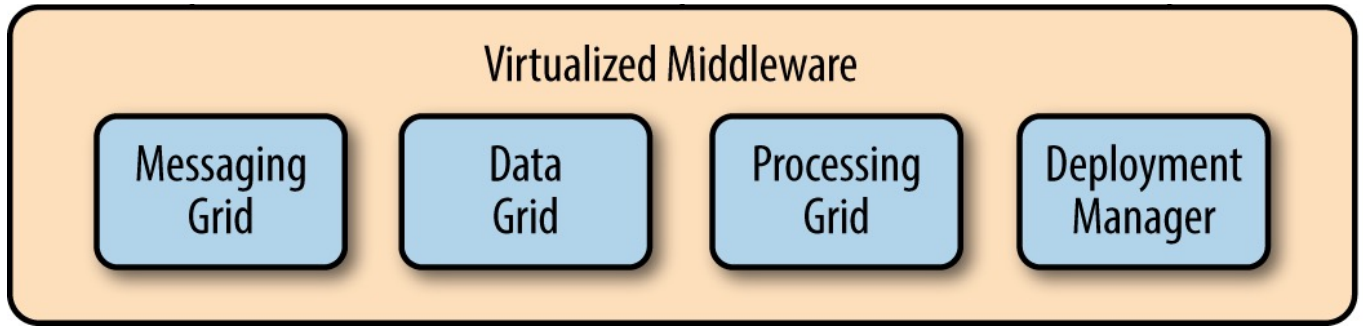
Basic Idea – Minimize Factors that Limit Scaling

Space-Based Architecture– Key Components

Ref: Neil Ford & Mark Richards
Software Architecture Fundamentals



The processing unit provides a virtualized environment that enables modules to have everything that they need appear to be local

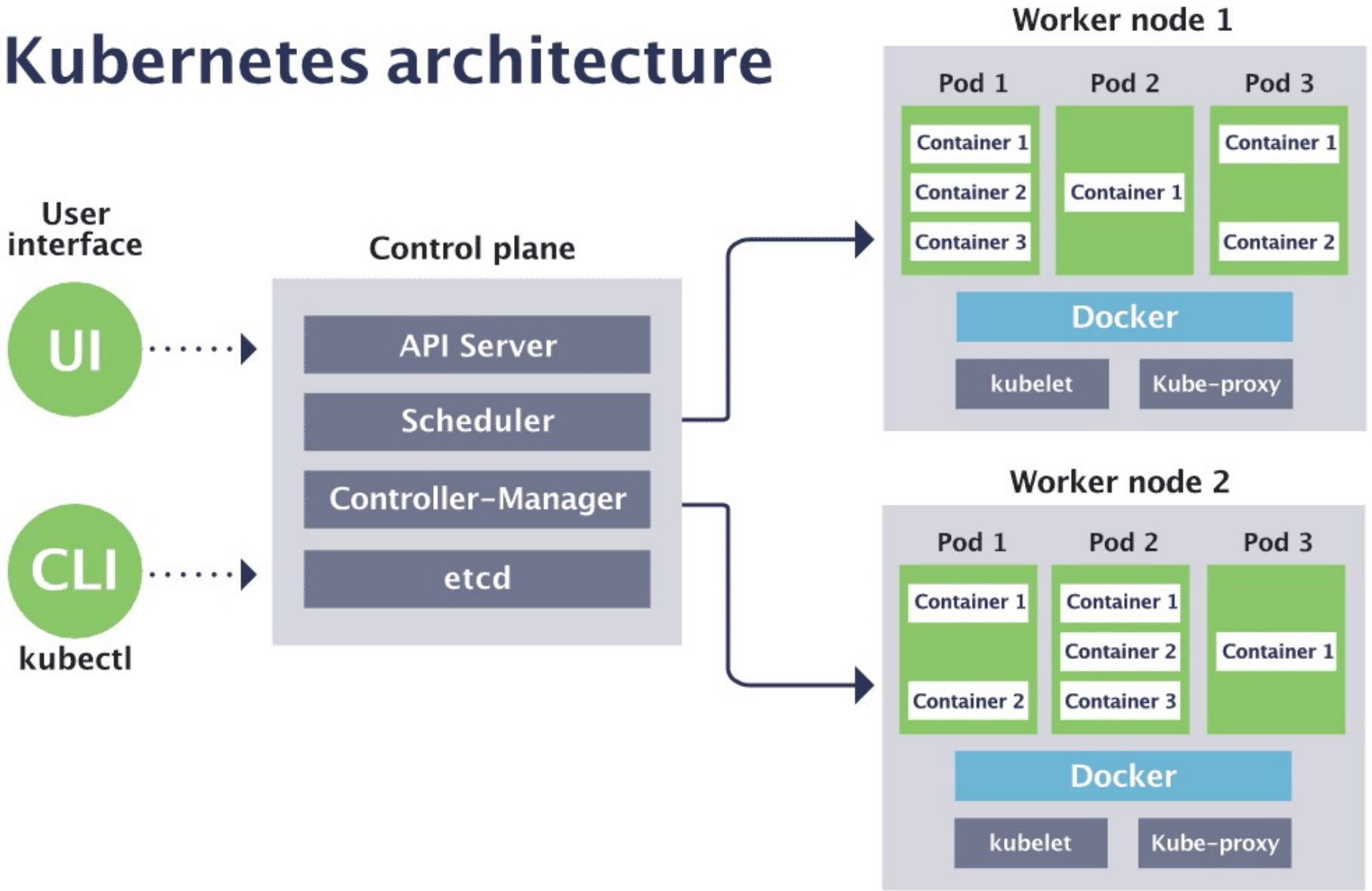


The virtual middleware connects to all processing units

- Messaging Grid: Behaves like a smart load balancer, intelligently routing messages to processing units
- Data Grid: Works with the data replication components in each processing unit to make sure the right data is replicated to the right Processing Unit and kept coherent
- Processing Grid: Mediates and orchestrates distributed work that is executed on multiple processing units
- Deployment Manager: Ensures the right code is deployed to the right processing unit as well as handling scaleup and scale down

Basic Idea – Minimize Factors that Limit Scaling

Space-Based Architecture– Example: Kubernetes



Space-Based Architecture–
Example: Service Mesh on Kubernetes