Micro Services Architecture Pattern

By: Vahid Rahimian

Software Architecture Course
Instructor: Dr. Habibi
Fall 2022

Agenda

- New Architectural Paradigm
- Monolithic, Layered Systems, and SOA
- Micro-Services Pattern Description
- Characteristics of Micro-Services Architecture
- Implementation Concerns

New Architectural Paradigm

Software Evolution Over Time

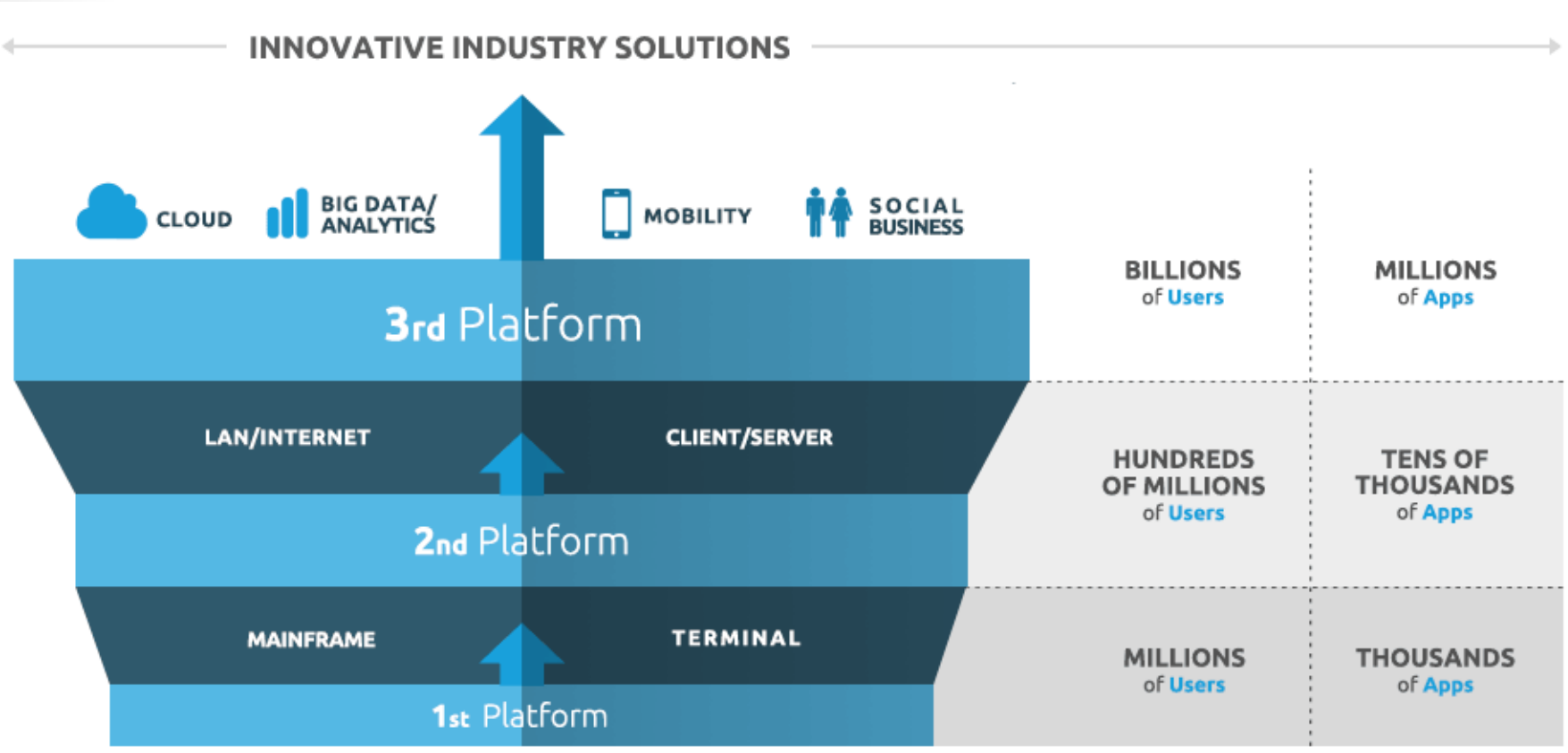
New Architectural Style for web-scale digital applications

New Architectural Paradigm

Software Evolution Over Time

- As internet systems have evolved, there is a huge increase in
  - Number of Users
  - Number of Developers
  - Line of Code
  - Number of Commits
  - Number of Deployments

Software Evolution Over Time

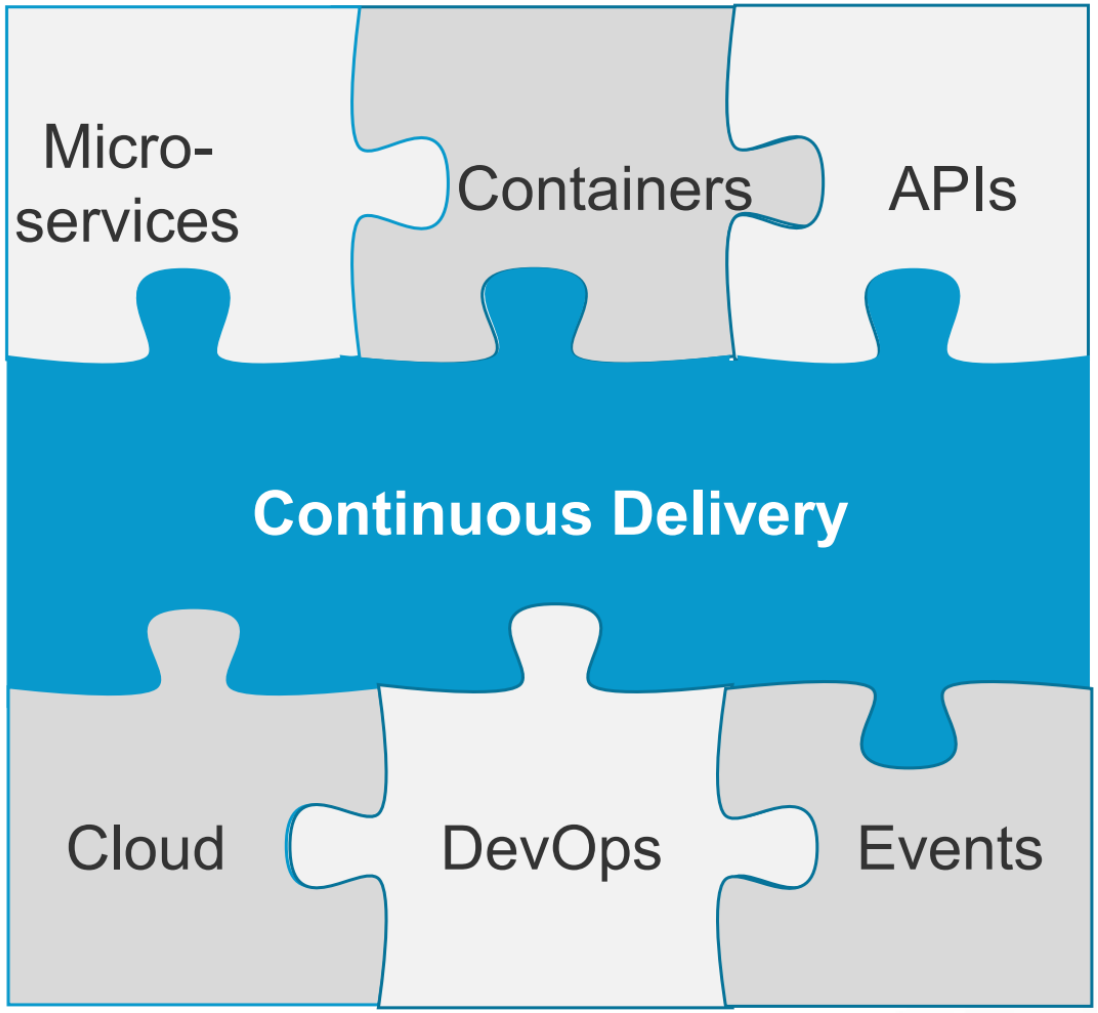


Let's Talk Some Numbers

- Google.com
  - 3.5 billion searches per day [1]
- Netflix
  - 110 million streaming subscribers [2]
- Amazon
  - 5 million deployments per month [3]
- Google
  - 2 billion lines of code [4]

[1]. http://www.internetlivestats.com/google-search-statistics/
[2]. https://www.statista.com/statistics/250934/quarterly-number-of-netflix-streaming-subscribers-worldwide/
[3]. http://www.allthingsdistributed.com/2014/11/apollo-amazon-deployment-engine.html
[4]. https://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/

New Architectural Paradigm



Monolithic, Layered Systems, and SOA

Software Monolith, Benefits and Problems
Layered Systems, Benefits and Problems
Service-Oriented Architecture

Software Monolith

- A Software Monolith
  - One build and deployment unit
  - One code base
  - One technology stack (Linux, JVM, Tomcat, Libraries)

- Benefits
  - Simple mental model for developers
    - one unit of access for coding, building, and deploying
  - Simple scaling model for operations
    - just run multiple copies behind a load balancer

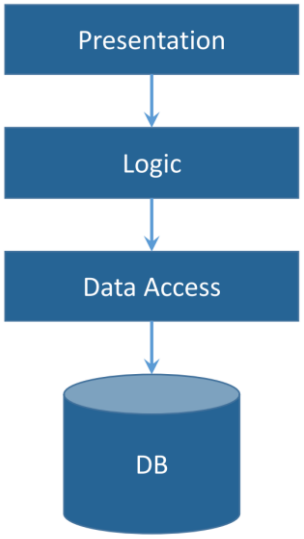
Problems of Software Monoliths

- Huge and intimidating code base for developers
- Slow development process
  - re-factorings take minutes, builds take hours, testing takes days
- Scaling is limited
  - Running a copy of whole system is resource-intense
  - Doesn’t scale with the data volume out-of-the-box
- Deployment frequency is limited
  - Re-deploying means halting the whole system
  - Re-deployments fail increase the perceived risk of deployment

Layered Systems

- A layered system decomposes a monolith into layers
- Usually: presentation, logic, data access
- At most one technology stack per layer
  - Presentation: Linux, JVM, Tomcat, Libs, EJB client, JavaScript
  - Logic: Linux, JVM, EJB container, Libs
  - Data Access: Linux, JVM, EJB JPA, EJB container, Libs

- Benefits
  - Simple mental model, simple dependencies
  - Simple deployment and scaling model



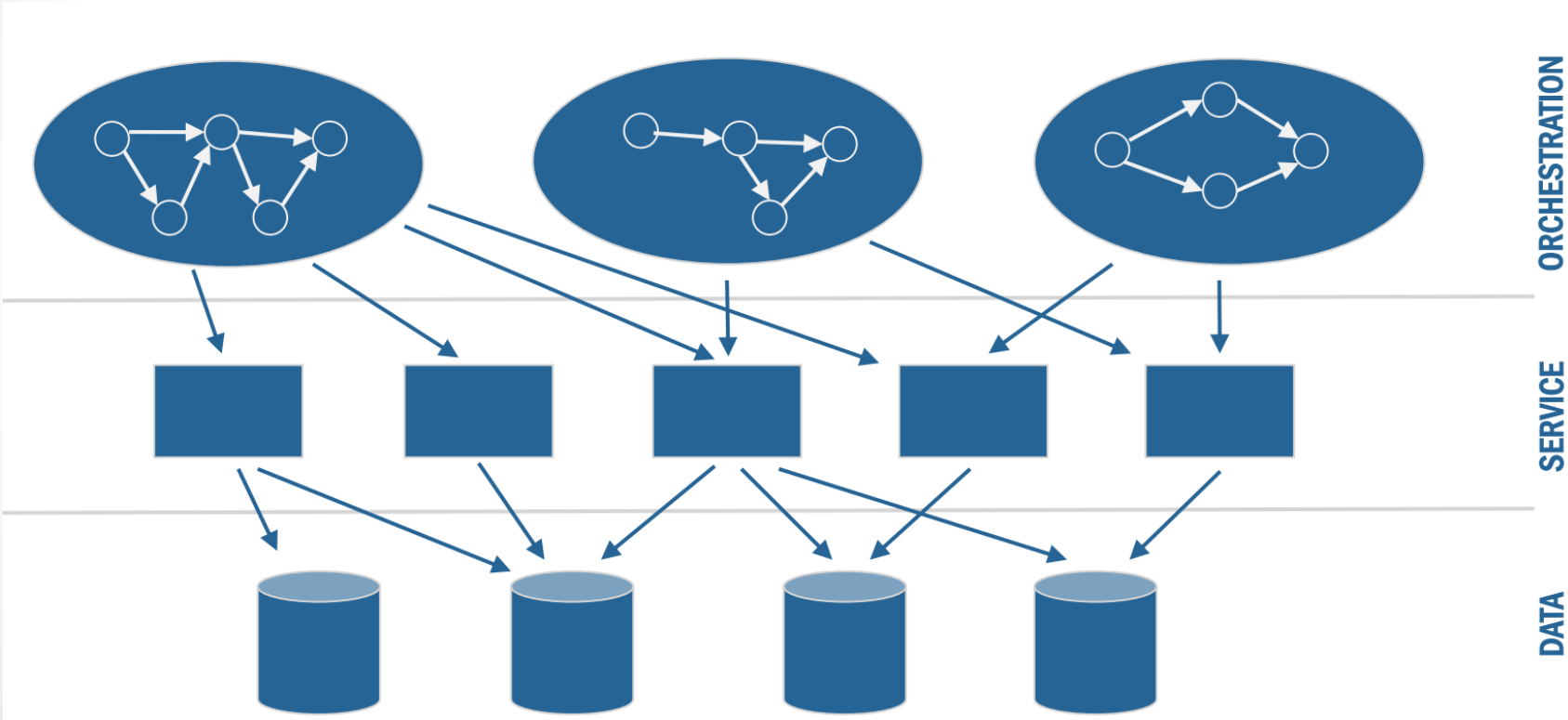
Problems of Layered Systems

- Still huge codebases (one per layer)
  - with the same impact on development, building, and deployment
- Scaling works better, but still limited
- Staff growth is limited: roughly speaking, one team per layer works well
  - Developers become specialists on their layer
  - Communication between teams is biased by layer experience (or lack thereof)

Growing Systems Beyond Limits

- Observing and documenting successful companies (e.g. Amazon, Netflix) lead to the definition of micro-services architecture principles.
  - alternative to monolithic applications and service-oriented architectures
- The evolution path has been through SOA

Service-Oriented Architecture



Service-Oriented Architecture

- SOA systems focus on functional decomposition, but
  - services are not required to be self-contained with data and UI, most of the time the contrary is pictured.
  - It is often thought as decomposition within tiers, and introducing another tier – the service orchestration tier

Service-Oriented Architecture

- In comparison to micro-services
  - SOA is focused on enabling business-level programming through business processing engines and languages such as BPEL and BPMN
  - SOA does not focus on independent deployment units and its consequences
  - Micro-services can be seen as “SOA – the good parts”

Service-Oriented Architecture

- Some say
  - SOA focuses on Reuse and Composition
  - Micro-Services adds Agility and Scalability

Micro-Services Pattern: History

- 2011: First discussions using this term at a software architecture workshop near Venice
- May 2012: microservices settled as the most appropriate term
- March 2012: “Java, the Unix Way” at 33rd degree by James Lewis
- September 2012: “µService Architecture“ at Baruco by Fred George
- All along, Adrian Cockroft pioneered this style at Netflix as “fine grained SOA”

Micro-Services Pattern Description

From Monolithic Apps to Microservices
From SOA to Microservices
Principle: Separately Deployed Units
Principle: Distributed Architecture
Defining Microservices

Micro-Services

- This architecture pattern is still evolving
  - there's a lot of confusion in the industry about what this pattern is all about and how it is implemented
- We will focus on key principles

Micro-Services

- Evolved from two main sources
  - Monolithic applications developed using the layered architecture pattern
  - Distributed applications developed through the service-oriented architecture pattern.

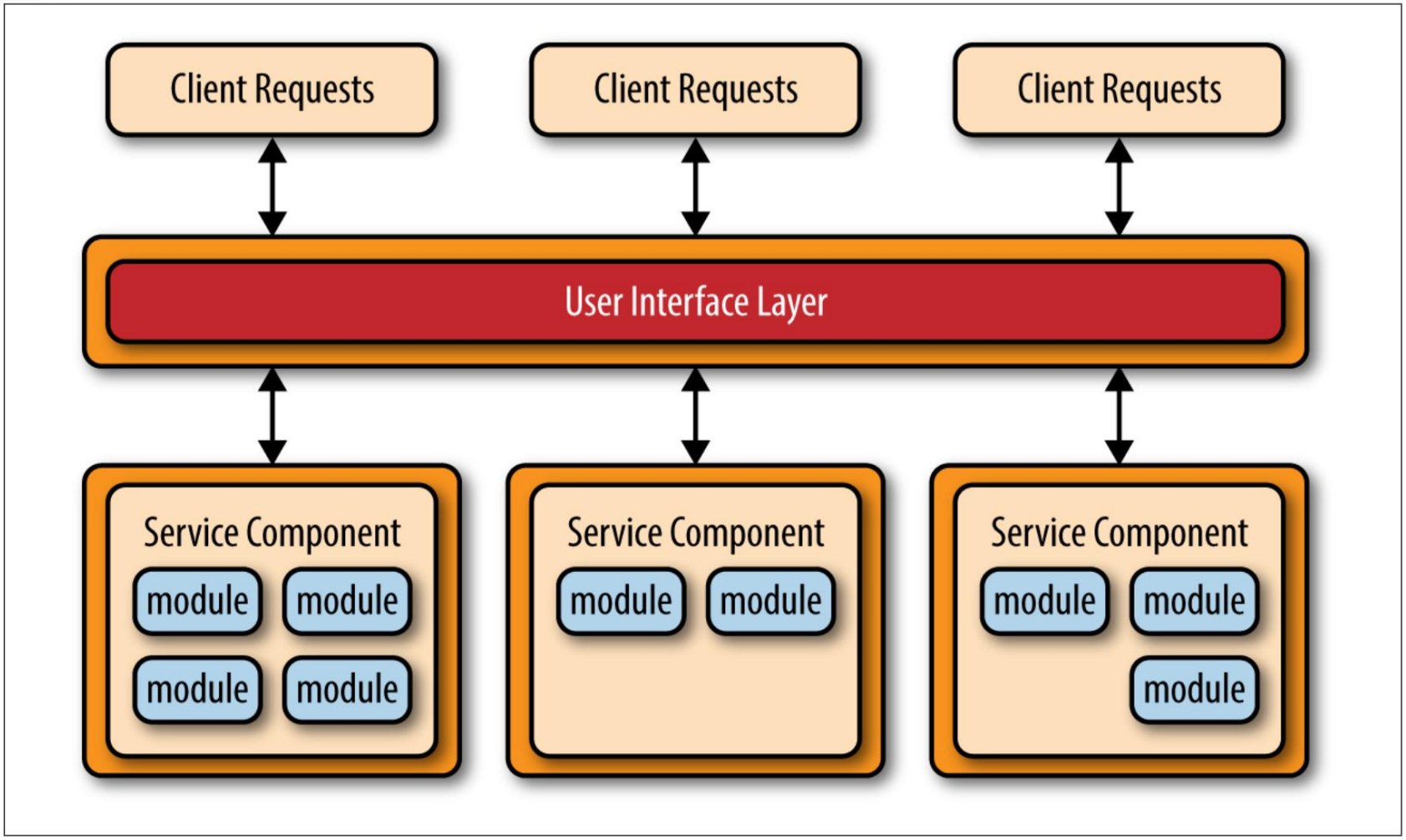
From Monolithic Apps to Microservices

- Through the development of continuous delivery
  - Monolithic applications typically consist of tightly coupled components that are part of a single deployable unit, making it cumbersome and difficult to change, test, and deploy the application (“monthly deployment” cycles)
- Microservices architecture pattern separates the application into multiple deployable units (service components)
  - that can be individually developed, tested, and deployed independent of other service components

From SOA to Microservices

- Issues found with applications implementing SOA
  - SOA offers unparalleled levels of abstraction, heterogeneous connectivity, service orchestration, and the promise of aligning business goals with IT capabilities
  - But it is complex, expensive, ubiquitous, difficult to understand and implement, and is usually overkill for most applications.
- The microservices architecture style addresses this complexity by simplifying the notion of a service, eliminating orchestration needs, and simplifying connectivity and access to service components

Basic Microservices Architecture Pattern

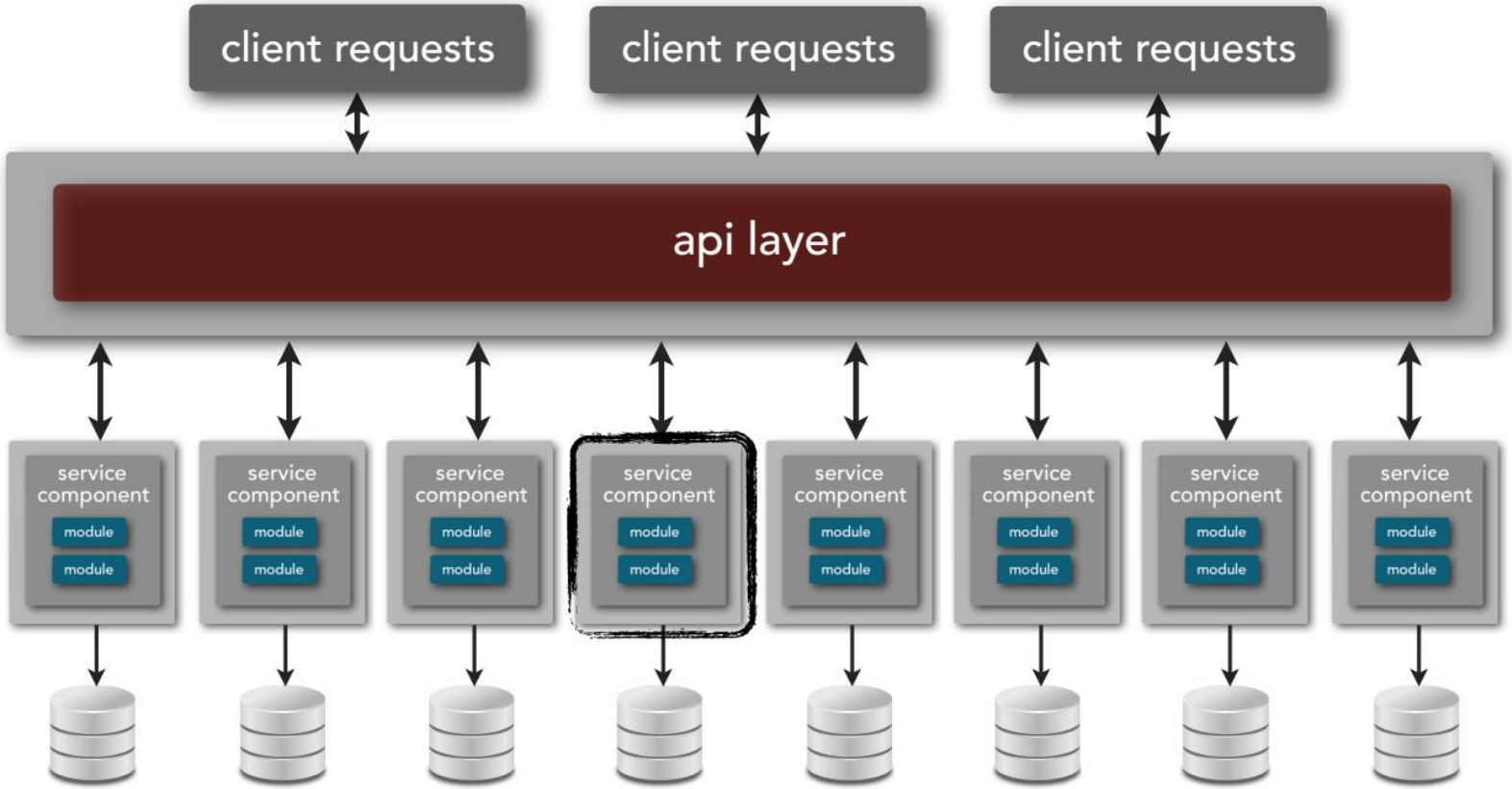


What is Service Component?

- Represent either a single-purpose function
  - e.g., providing the weather for a specific city or town
- or an independent portion of a large business application
  - e.g., stock trade placement or determining auto-insurance rates.

- Designing the right level of service component granularity is one of the biggest challenges within a microservices architecture.

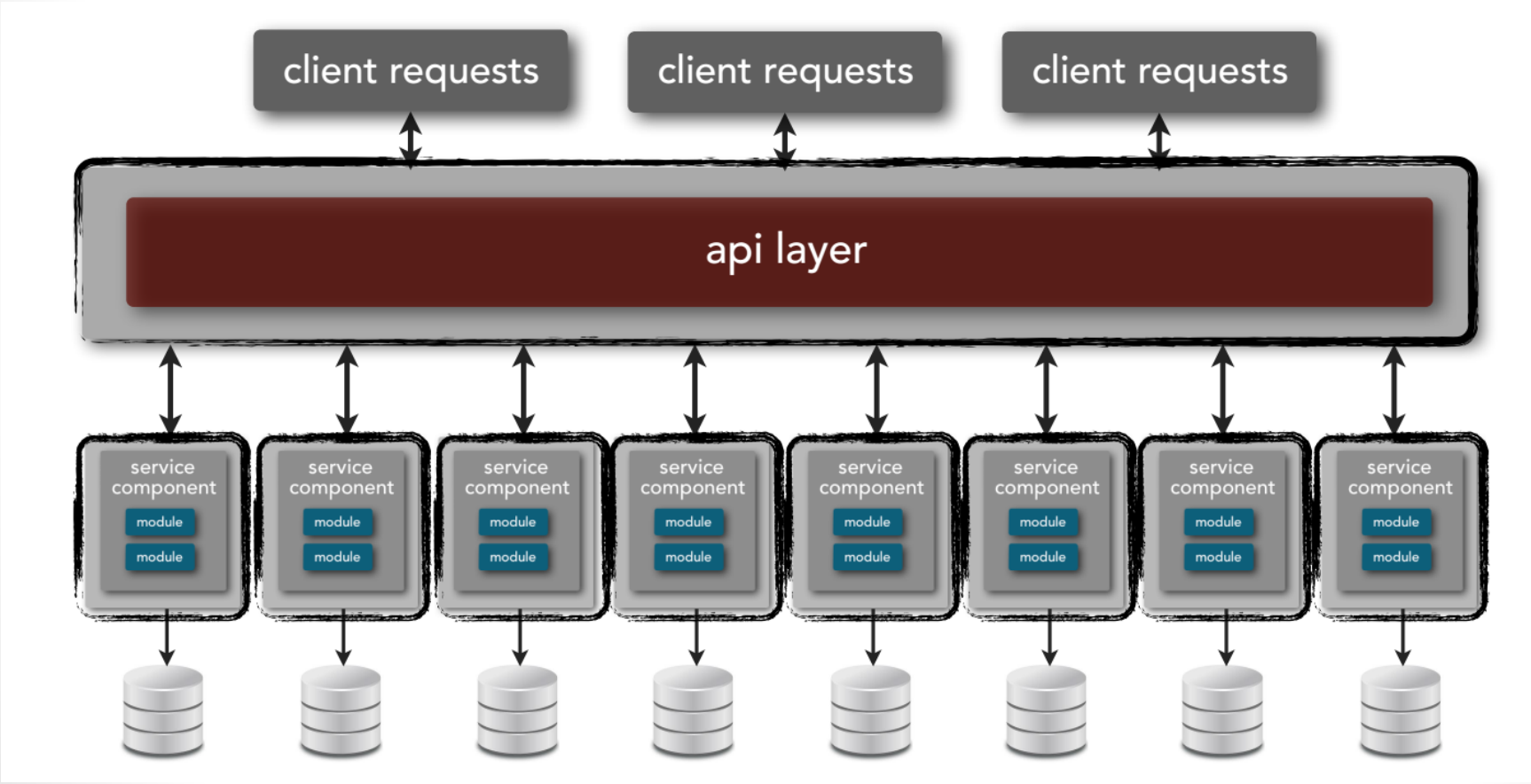
What is Service Component?



Principle: Separately Deployed Units

- Each component of microservices architecture is deployed as a separate unit, allowing for
  - Easier deployment through an effective and streamlined delivery pipeline
  - Increased scalability
  - High degree of application and component decoupling

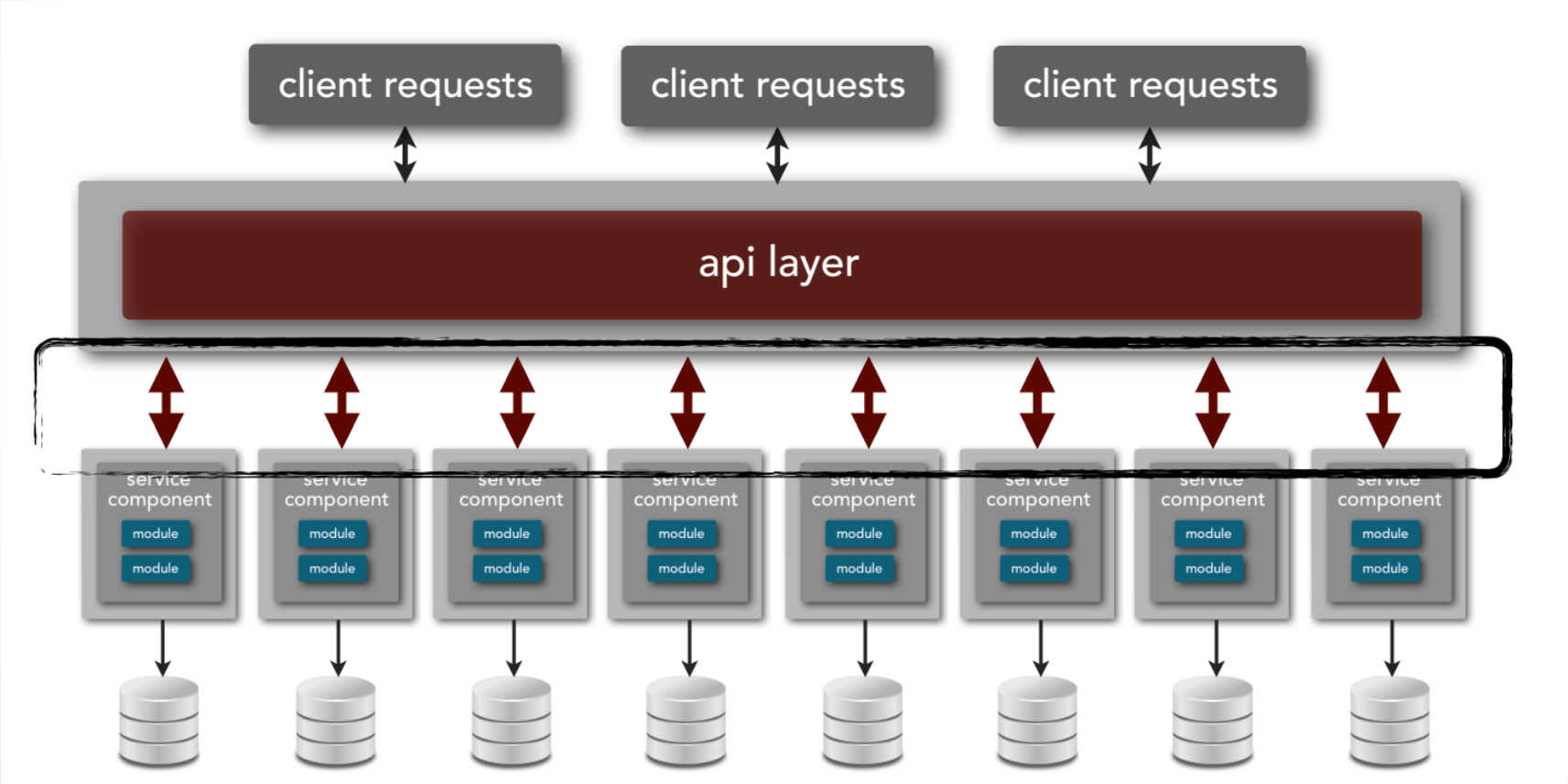
Principle: Separately Deployed Units



Principle: Distributed Architecture

- All components within the architecture are fully decoupled from one other and accessed through some sort of remote access protocol
  - e.g., JMS, AMQP, REST, SOAP, RMI, etc.
- Distributed nature: scalability and deployment characteristics.

Principle: Distributed Architecture



Defining Microservices

- ***functional system decomposition into manageable and independently deployable components***
- The term “micro” refers to the sizing
  - a microservice must be manageable by a single development team (5-9 developers)
  - Microservices are fun-sized services, as in “still fun to develop and deploy”
- Functional system decomposition means vertical slicing
  - in contrast to horizontal slicing through layers)
- Independent deployability implies no shared state and inter-process communication
  - often via HTTP REST interfaces

Characteristics of Micro-Services Architecture

Componentization via Services
Organized around Business Capabilities
Products not Projects
Smart Endpoints, Dumb Pipes
Standardize Integration, not Platform
Decentralized Data Management
Infrastructure Automation
Design for Failure

Componentization via Services

- Interaction mode: share-nothing, cross-process communication
- Explicit, REST-based public interface
- Independently deployable
- Sized and designed for replaceability
  - Upgrading technologies should not happen big-bang, all-or-nothing-style

Componentization via Services

- Downsides
  - Communication is more expensive than in-process
  - Interfaces need to be coarser-grained
  - Re-allocation of responsibilities between services is harder

Organized around Business Capabilities

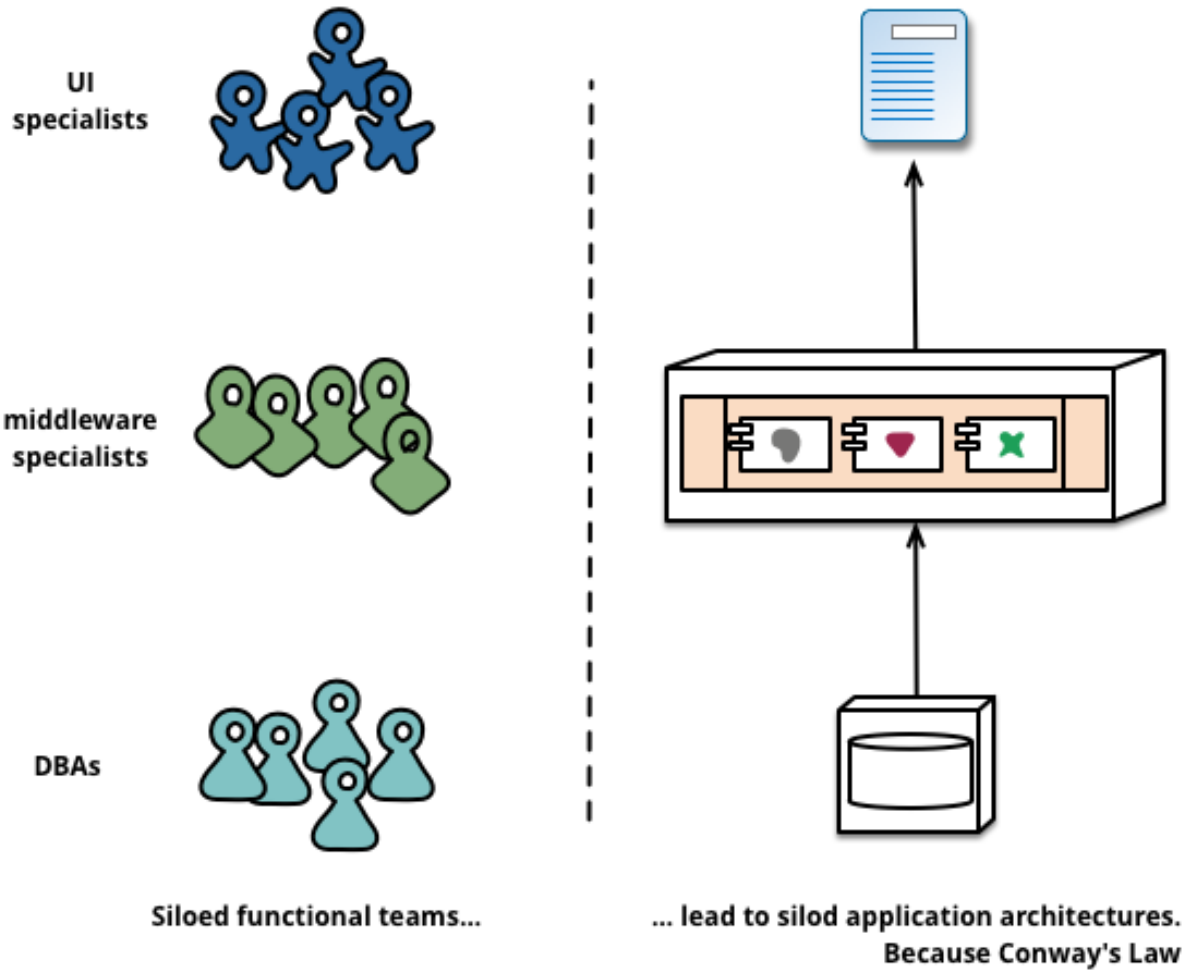
- *Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.*

-- Melvyn Conway, 1967

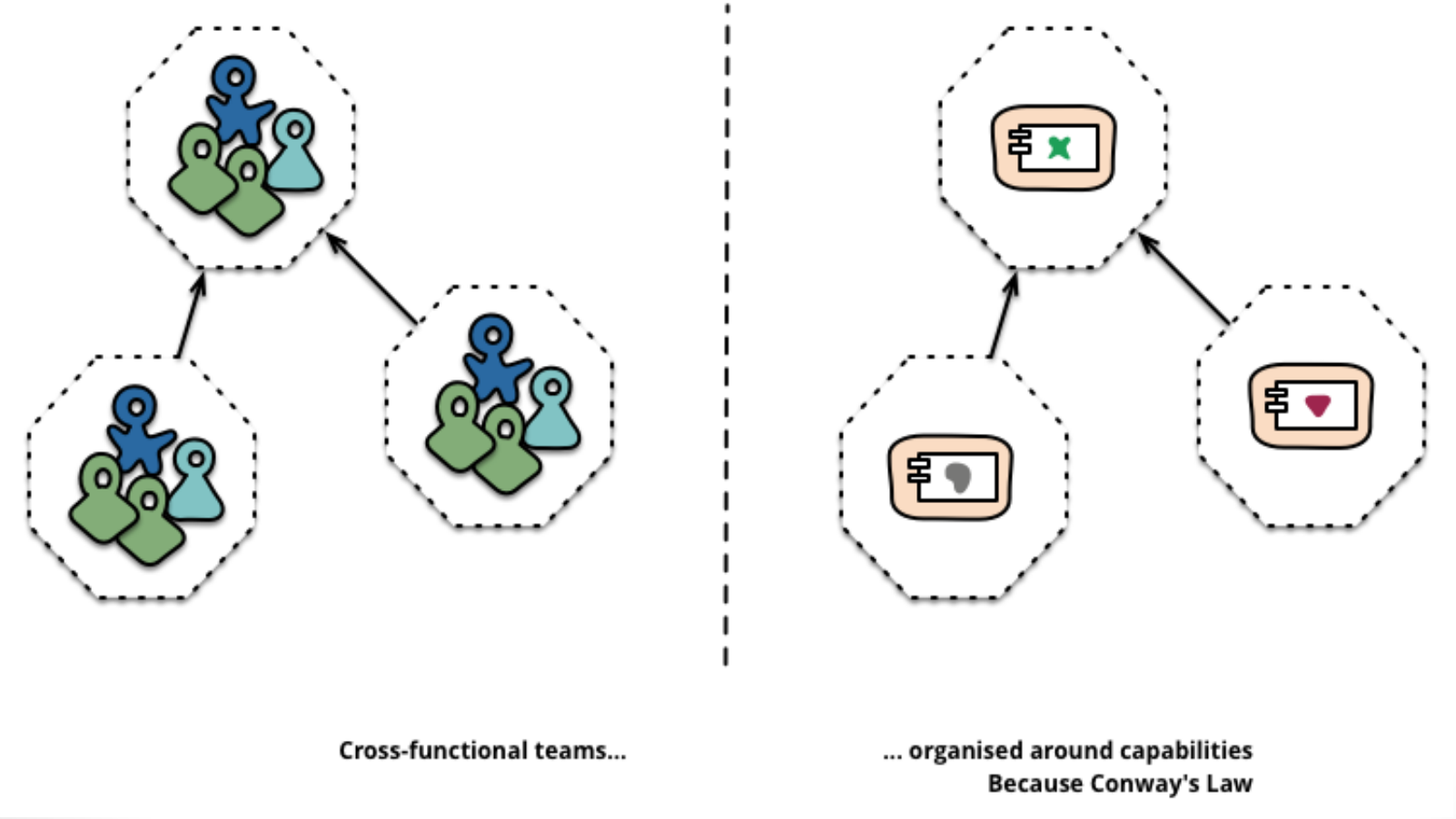
Organized around Business Capabilities

- Traditional problem: management focuses on the technology layer
  - leading to UI teams, server-side logic teams, and database teams.
- When teams are separated along these lines, even simple changes can lead to a cross-team project taking time and budgetary approval.

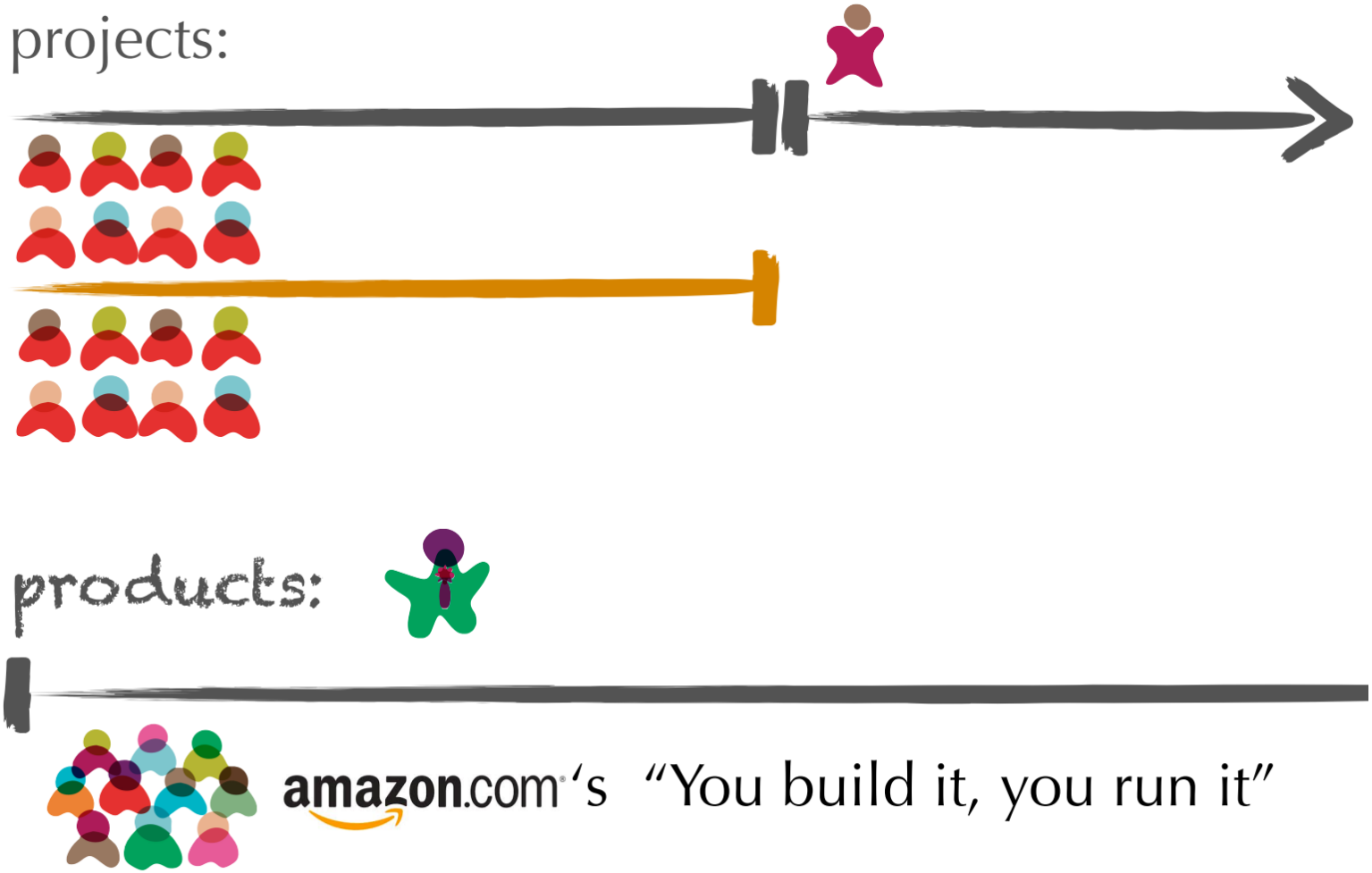
Monolithic Apps Organized around Technology Layer



Micro-Services Organized around Business Capabilities



Products not Projects

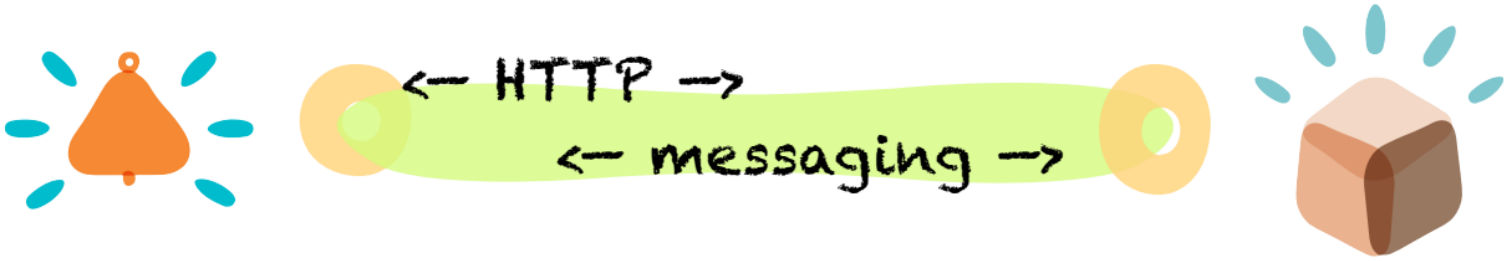


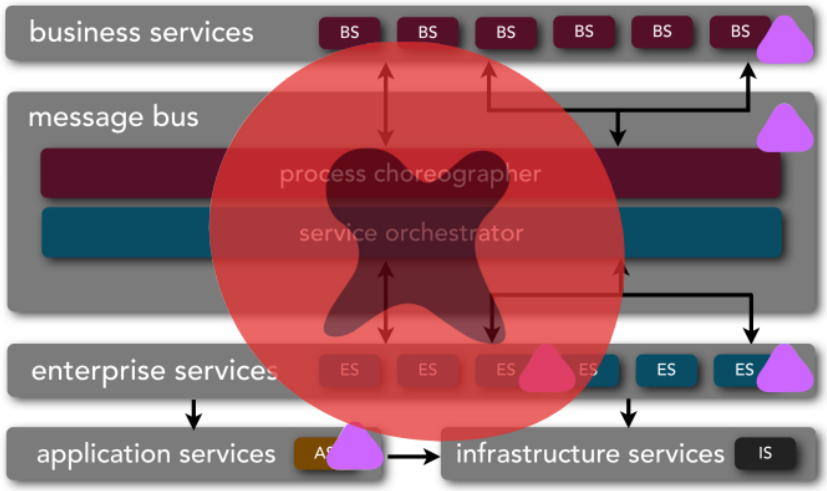
Smart Endpoints, Dumb Pipes

- Enterprise Service Bus (ESB) often include sophisticated facilities for message routing, choreography, transformation, and applying business rules.

- Microservices use simple RESTish protocols rather than complex protocols such as WS-Choreography or BPEL or orchestration by a central tool.

Smart Endpoints, Dumb Pipes

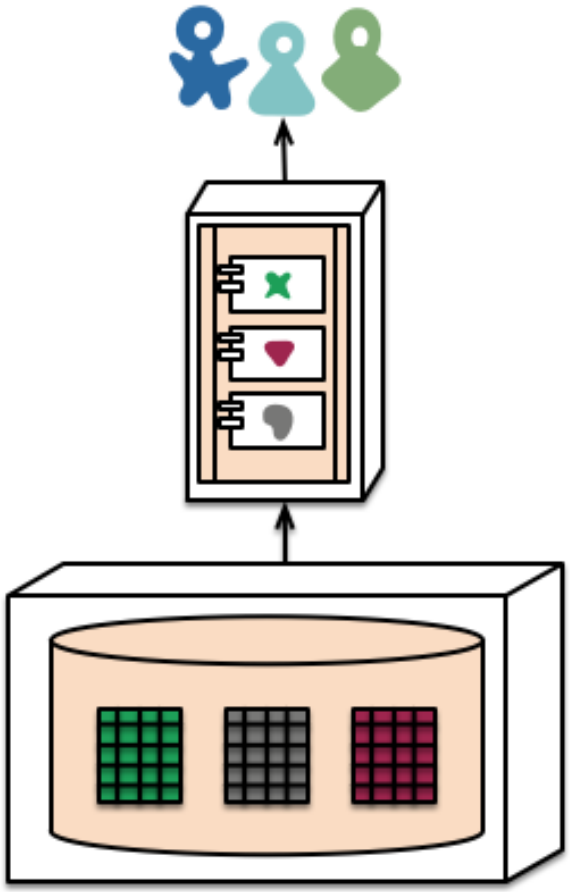




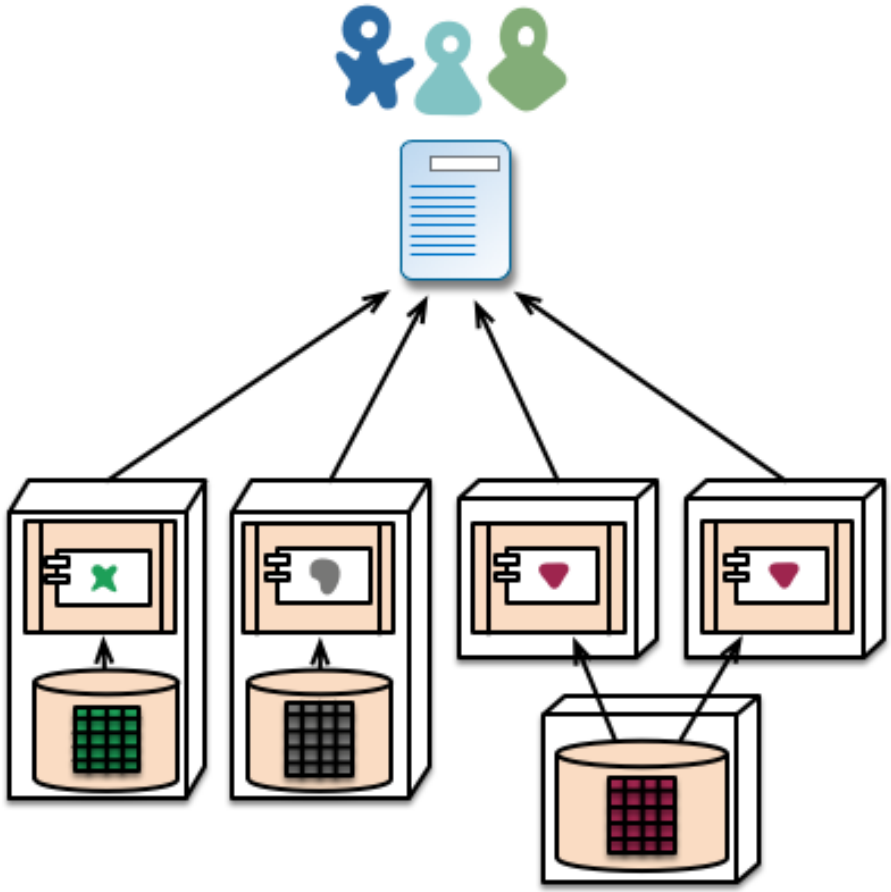
Standardize Integration, not Platform

- Embrace different solutions where sensible
  - You want to use Node.js to standup a simple reports page? Go for it. C++ for a particularly gnarly near-real-time component? Fine.
- Standardize in the gaps between services - be flexible about what happens inside the boxes
- Even for communication contracts, patterns like Tolerant Reader are often applied to microservices.

Decentralized Data Management



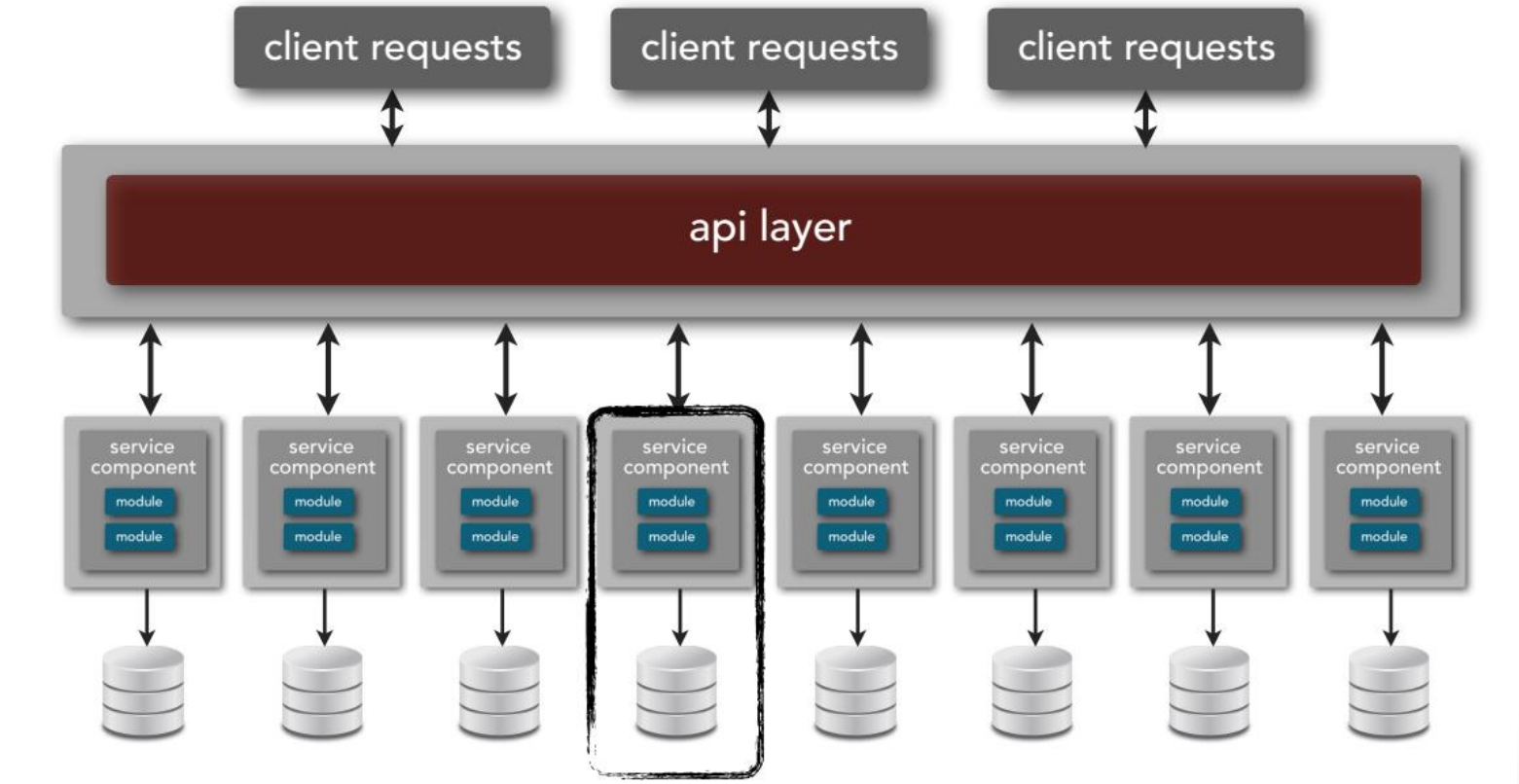
monolith - single database



microservices - application databases

Decentralized Data Management

- Bounded Context in Domain-Driven Design



Decentralized Data Management

- Decentralizing responsibility for data across microservices has implications for managing updates.
- The common approach to dealing with updates has been to use transactions to guarantee consistency when updating multiple resources.
- This approach is often used within monoliths.

Decentralized Data Management

- Using transactions as before, helps with consistency
  - but imposes significant temporal coupling, which is problematic across multiple services.
- Distributed transactions are difficult to implement
- Microservice architectures emphasize transaction-less coordination between services
  - consistency may only be eventual consistency and problems are dealt with by compensating operations.

Decentralized Data Management

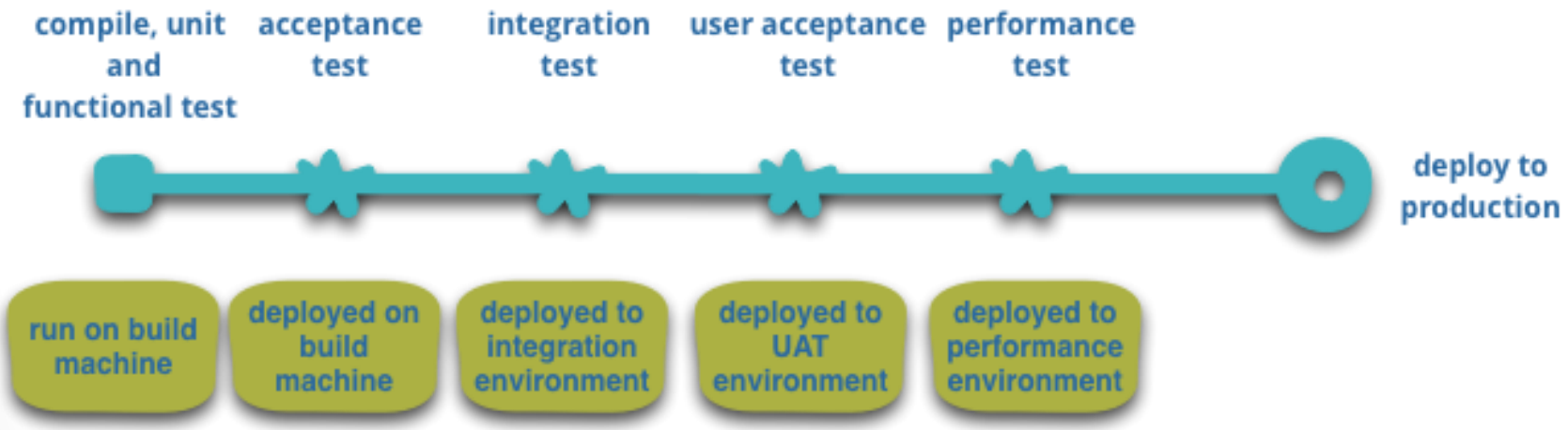
- Often businesses handle a degree of inconsistency in order to respond quickly to demand, while having some kind of reversal process to deal with mistakes.

- The trade-off is worth it as long as the cost of fixing mistakes is less than the cost of lost business under greater consistency.

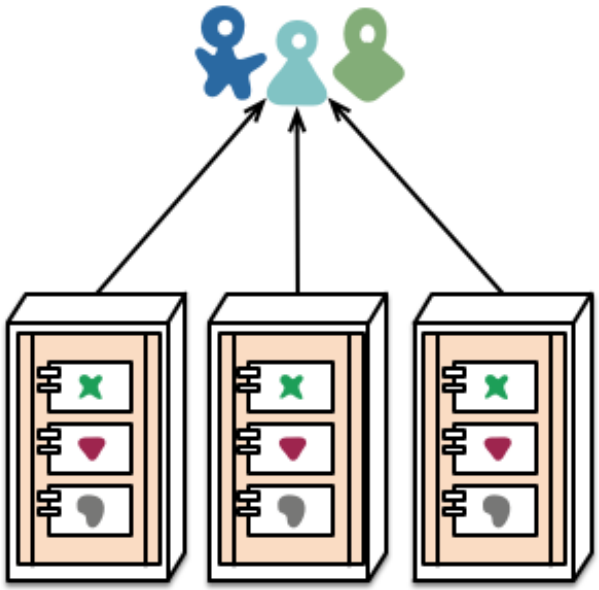
Infrastructure Automation

- Evolution of the cloud and AWS in particular has reduced the operational complexity of building, deploying and operating microservices.
- Many of the products or systems being build with microservices are being built by teams with extensive experience of Continuous Delivery and it's precursor, Continuous Integration.

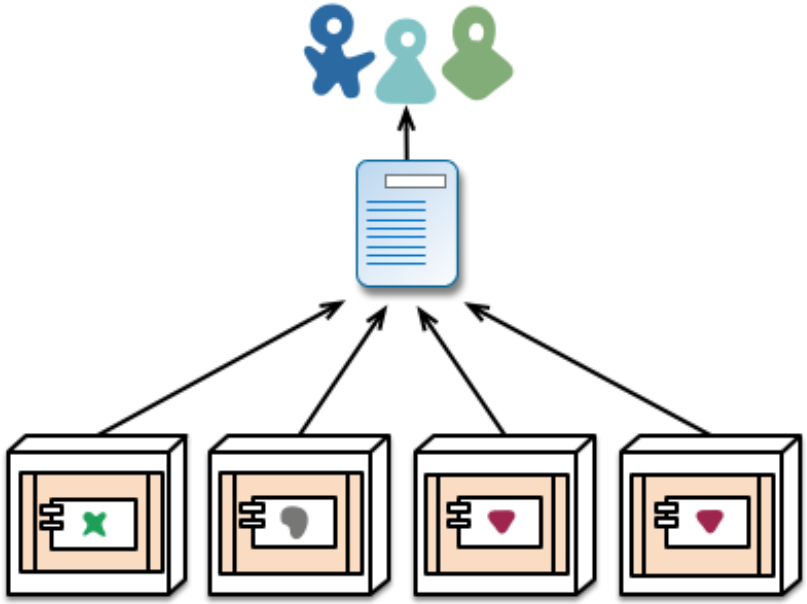
Infrastructure Automation



Infrastructure Automation: Easier Deployment



monolith - multiple modules in the same process



microservices - modules running in different processes

Design for Failure

- A consequence of using services as components, is that applications need to be designed so that they can tolerate the failure of services.
- Any service call could fail due to unavailability of the supplier, the client has to respond to this as gracefully as possible.
- This is a disadvantage compared to a monolithic design as it introduces additional complexity to handle it.

Design for Failure

- Microservice applications put a lot of emphasis on real-time monitoring of the application
  - checking both architectural elements (how many requests per second is the database getting)
  - and business relevant metrics (such as how many orders per minute are received).
- Semantic monitoring can provide an early warning system of something going wrong that triggers development teams to follow up and investigate.

Implementation Concerns

Implementation Topologies
Avoid Dependencies and Orchestration
Where NOT to Use Microservices
Pattern Analysis

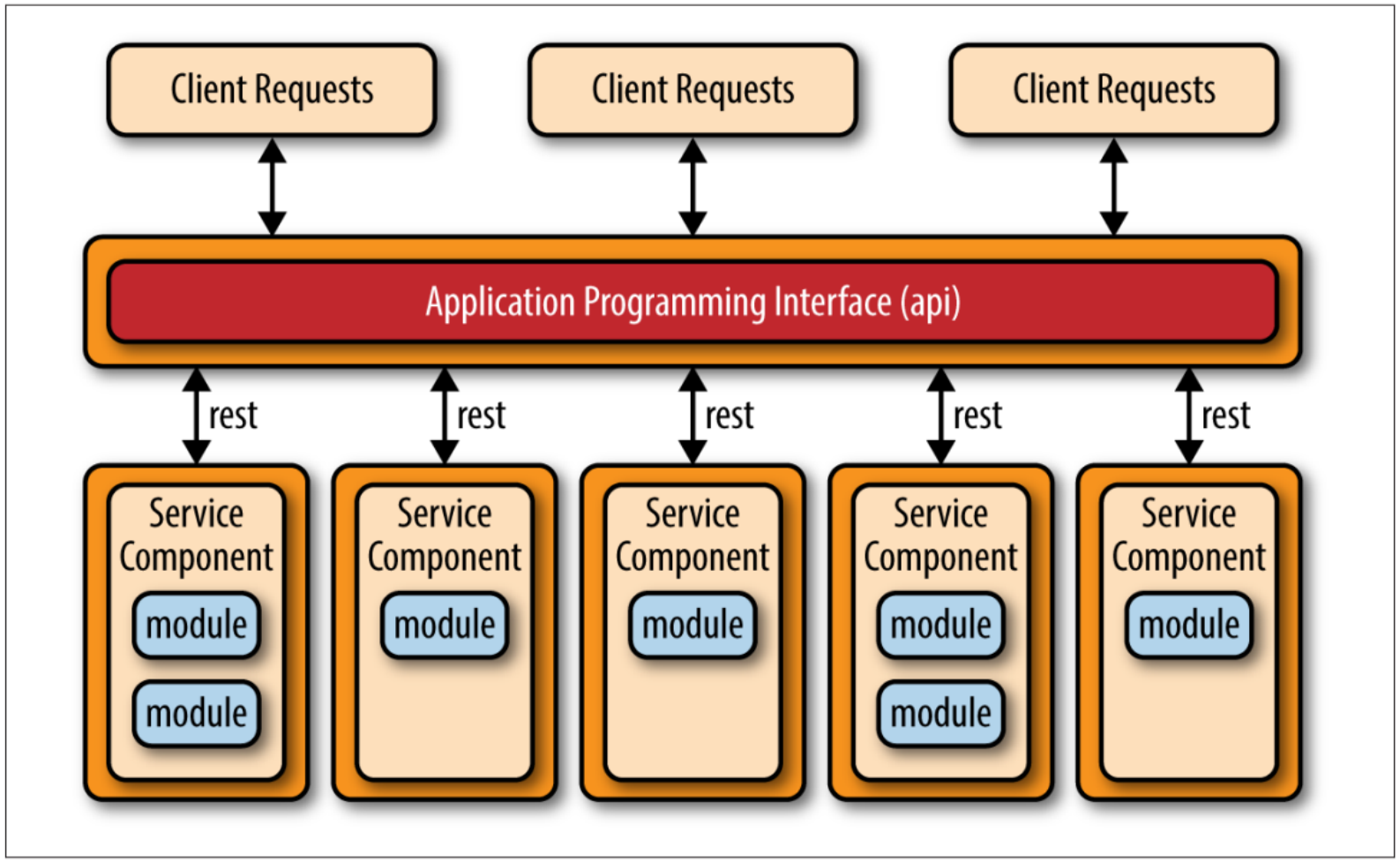
Implementation Topologies

- API REST-based topology
- Application REST-based topology
- Centralized messaging topology

Implementation Topologies: API REST-based topology

- useful for websites that expose small, self-contained individual services through some sort of API
- consists of very fine-grained service components that contain one or two modules that perform specific business functions independent from the rest of the services

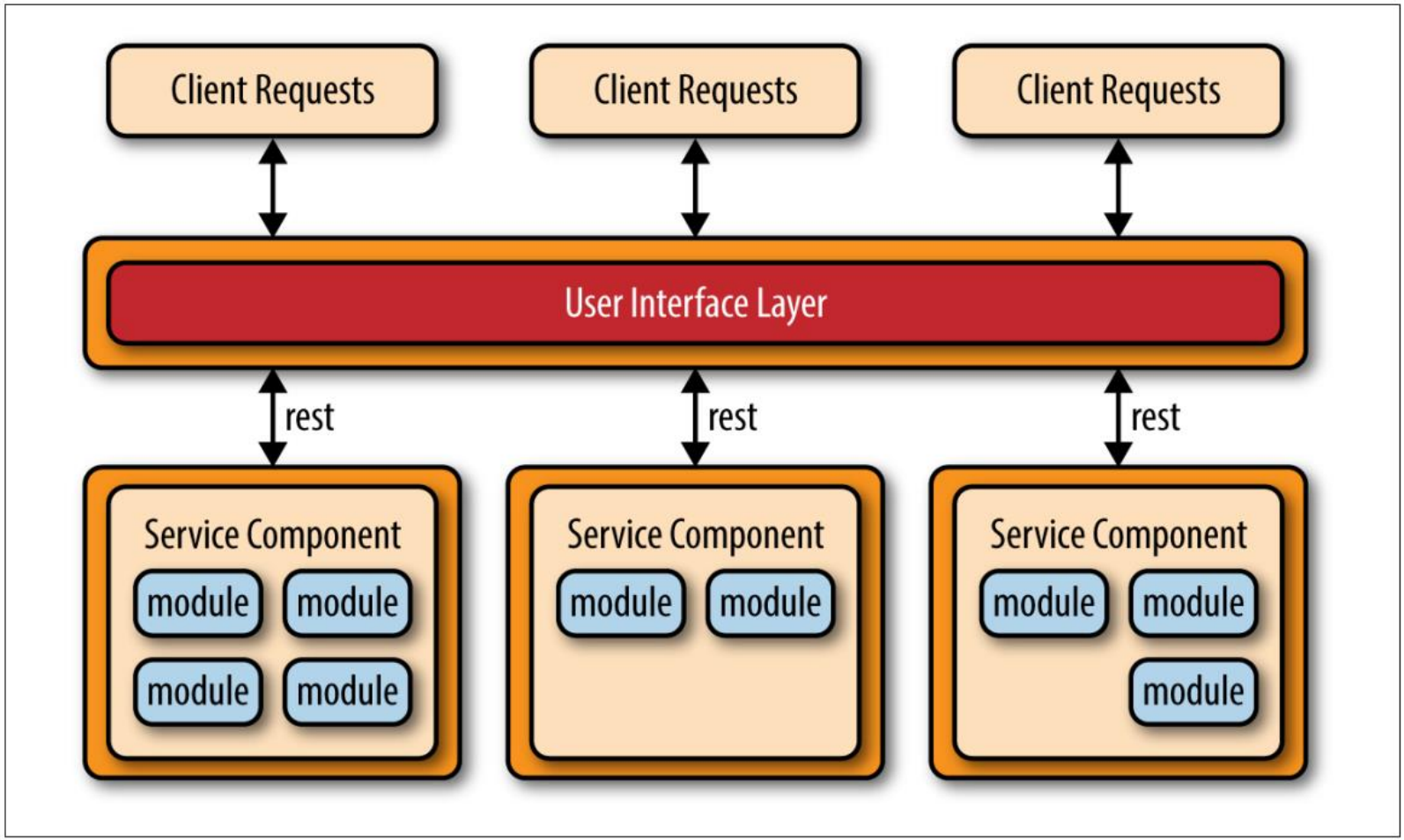
Implementation Topologies: API REST-based topology



Implementation Topologies: Application REST-based topology

- client requests are received through traditional web-based or fat-client business application screens rather than through a simple API layer
- service components tend to be larger, more coarse-grained, and represent a small portion of the overall business application rather than fine-grained, single action services

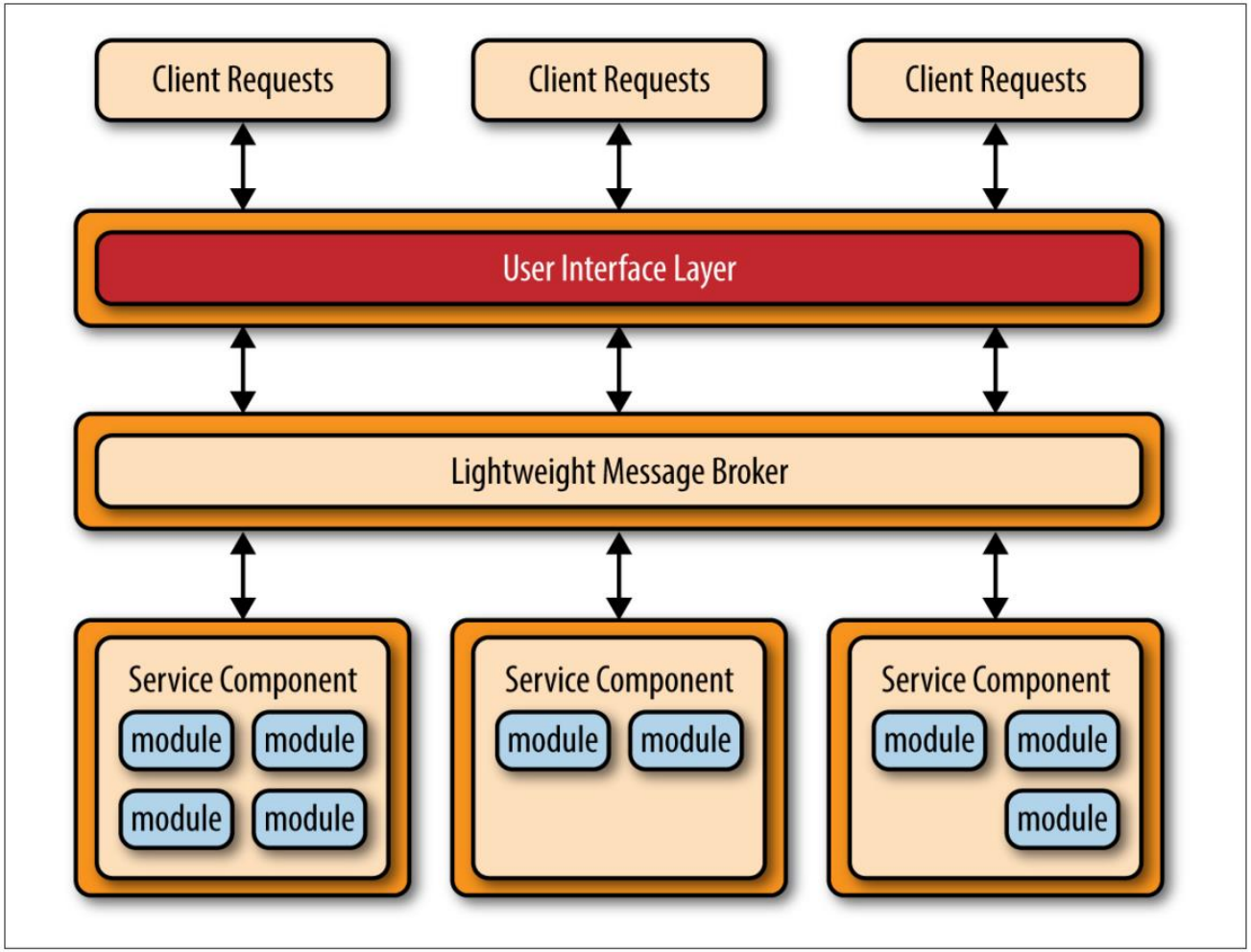
Implementation Topologies: Application REST-based topology



Implementation Topologies: Centralized Messaging topology

- Similar to application REST- based topology except that instead of using REST for remote access, this topology uses a lightweight centralized message broker (e.g., ActiveMQ, HornetQ, etc.).
- NOT to confuse it with the service-oriented architecture pattern or consider it “SOA-Lite.“
  - The lightweight message broker found in this topology does not perform any orchestration, transformation, or complex routing; rather, it is just a lightweight transport to access remote service components

Implementation Topologies: Centralized Messaging topology



Implementation Topologies: Centralized Messaging topology

- Benefits of this topology: advanced queuing mechanisms, asynchronous messaging, monitoring, error handling, and better overall load balancing and scalability.
- The single point of failure and architectural bottleneck issues are addressed through broker clustering and broker federation
  - splitting a single broker instance into multiple broker instances to divide the message throughput load based on functional areas of the system.

Avoid Dependencies and Orchestration

- Main challenge of the microservices architecture pattern:
  - Determining the correct level of granularity for the service components.
- If too coarse-grained:
  - May not realize the benefits (deployment, scalability, testability, and loose coupling)
- If too fine-grained
  - Service orchestration requirements
  - Turn into a heavyweight service-oriented architecture, complete with all the complexity, confusion, expense, and fluff typically found with SOA-based applications.

Avoid Dependencies and Orchestration

- If need to orchestrate service components from within the user interface or API layer of the application
  - Service components are too fine-grained
- if need to perform inter-service communication between service components to process a single request
  - service components are either too fine-grained or they are not partitioned correctly from a business functionality standpoint
- Inter-service data communication: shared database
- Shred functionality: repeating portions of business logic

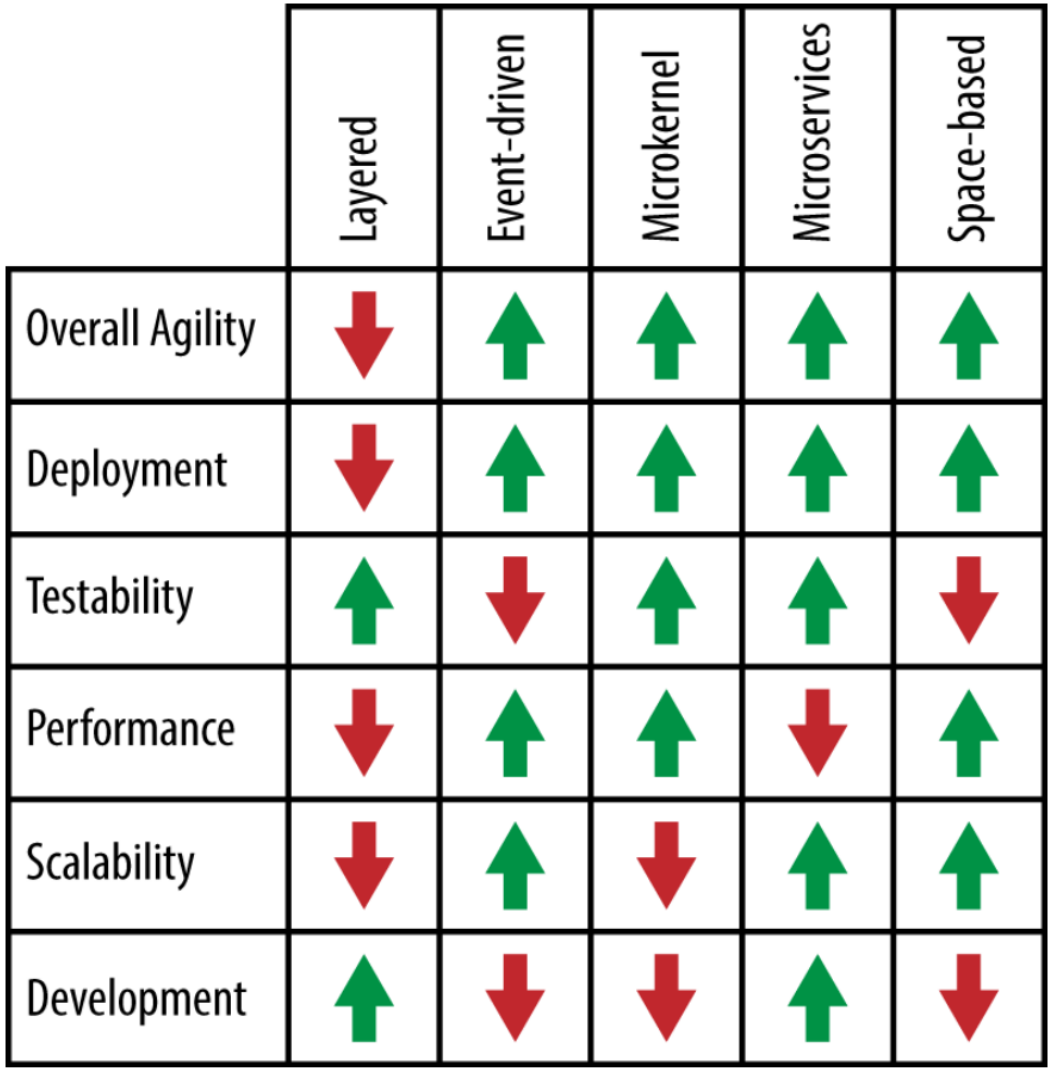
Where NOT to Use Microservices

- If regardless of granularity of service component, still cannot avoid service-component orchestration
- Because of the distributed nature, it is very difficult to maintain a single transactional unit of work across (and between) service components.
- Microservices require some sort of transaction compensation framework for rolling back transactions, which adds significant complexity to this relatively simple and elegant architecture pattern.

Pattern Analysis

- Overall Agility
- Ease of Deployment
- Testability
- Performance
- Scalability
- Ease of Development

Pattern Analysis



	Layered	Event-driven	Microkernel	Microservices	Space-based
Overall Agility	↓	↑	↑	↑	↑
Deployment	↓	↑	↑	↑	↑
Testability	↑	↓	↑	↑	↓
Performance	↓	↑	↑	↓	↑
Scalability	↓	↑	↓	↑	↑
Development	↑	↓	↓	↑	↓

Any Questions?