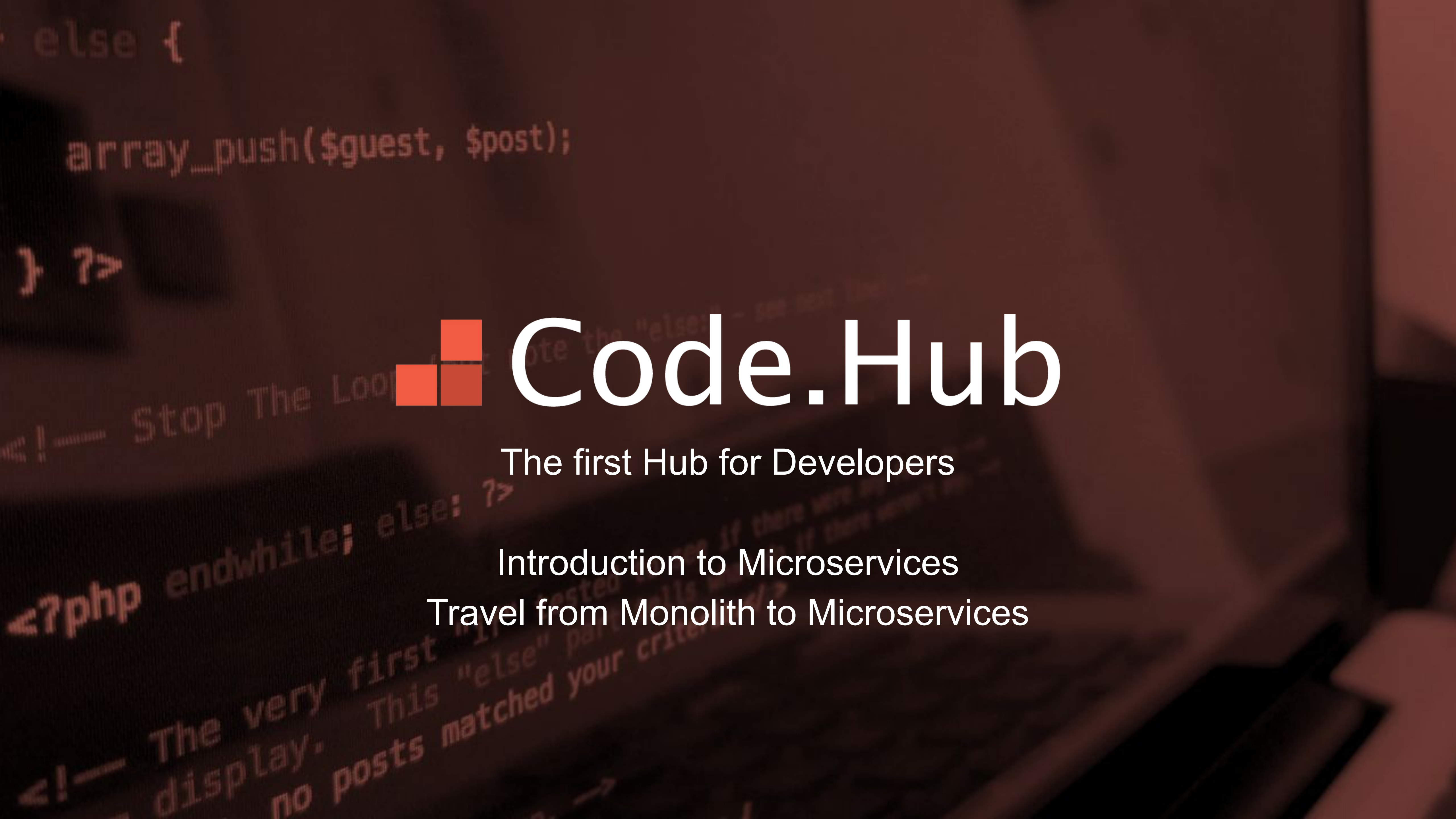


Code.Hub

The first Hub for Developers

Introduction to Microservices

Travel from Monolith to Microservices

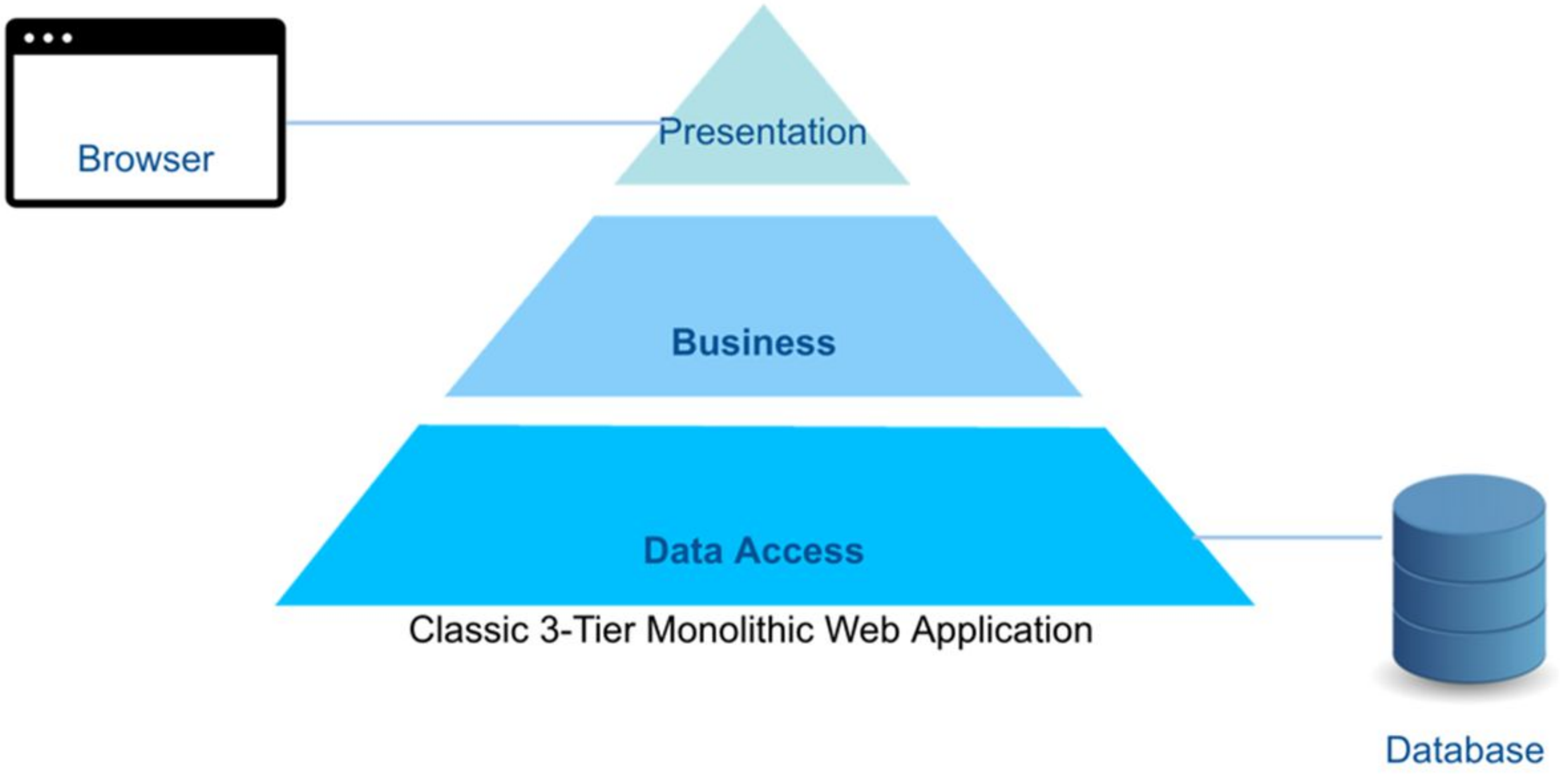
Microservice Architecture

Code.Hub

Monolithic Applications

Code.Hub

Monolithic Applications



Classic 3-Tier Monolithic Web Application

Code.Hub

Traditional Applications

Everything is integrated

But what happens if we want to:

- We want a double cassette
- Or a CD player
- Or a digital radio
- Or… anything else?

Stereo Radio Cassette Recorder
AIWA
●ステレオ録音に威力を発揮する
ワンポイントステレオマイク
別売：CM-2000　¥7,900

Code.Hub

With Modular Applications on the other side

- Each part is independent
- Do you need a CD Player?
- Do you need a digital input/output

Code.Hub

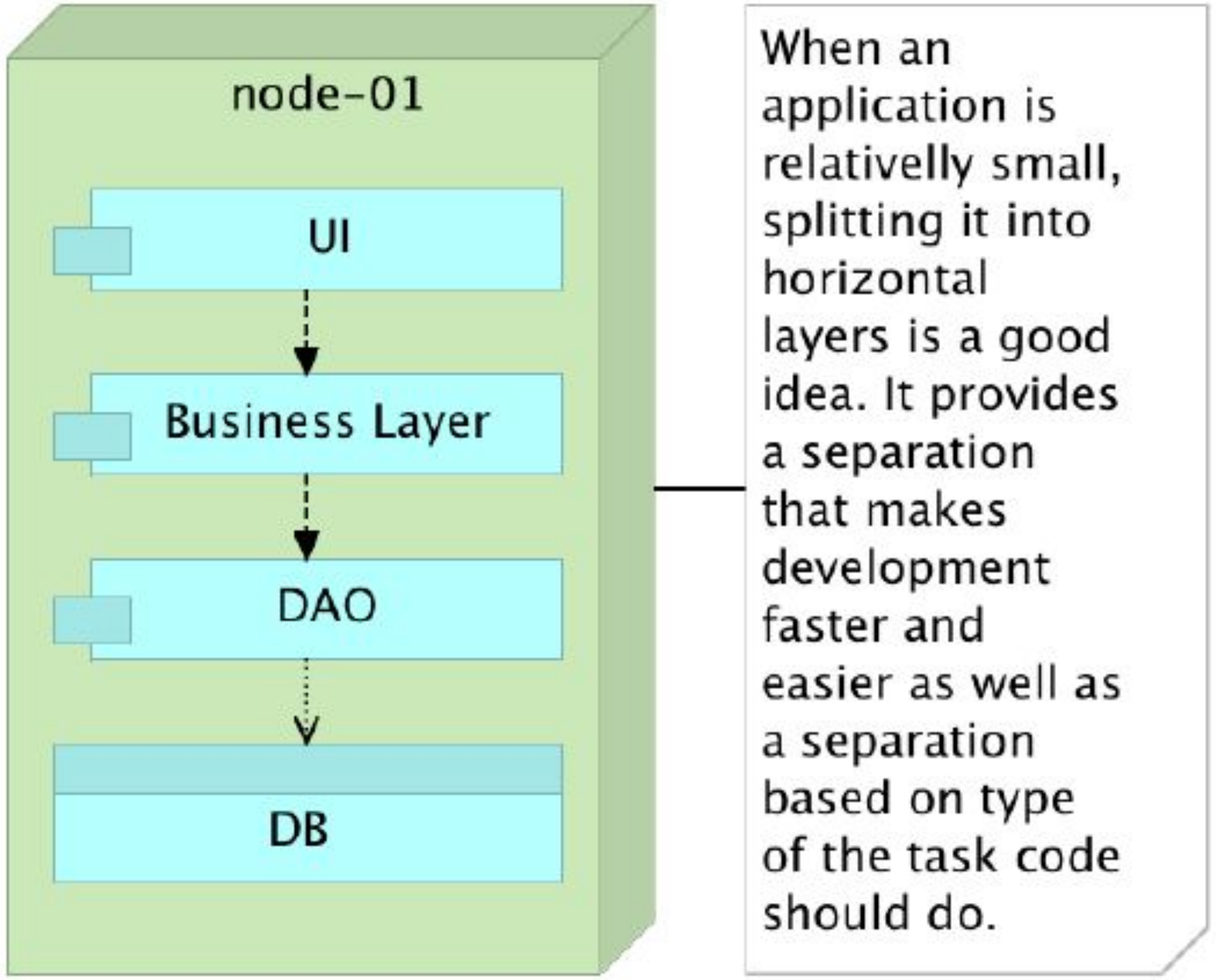
Monolithic Style

- An application built as a single unit, 3 main parts
  - a client-side user interface
  - a database
  - and a server-side application that handles HTTP requests, execute domain logic, retrieve and update data from the database, and select and populate HTML views to be sent to the browser
- This server-side application is a monolith
  - a single logical executable
  - any change requires building and deploying a new version

Monolithic Style

- Natural way to build a system
  - Code specific to each layer (UI, Logic, DB)
  - Generally required a minimum of 3 languages
  - Divided into classes, functions and namespaces
- Locally built and tested on developers' machines and in development environment
- CI/CD pipeline to secure production
- Single deployment
- Single runtime
- Single codebase
- Interaction between classes is mostly synchronous

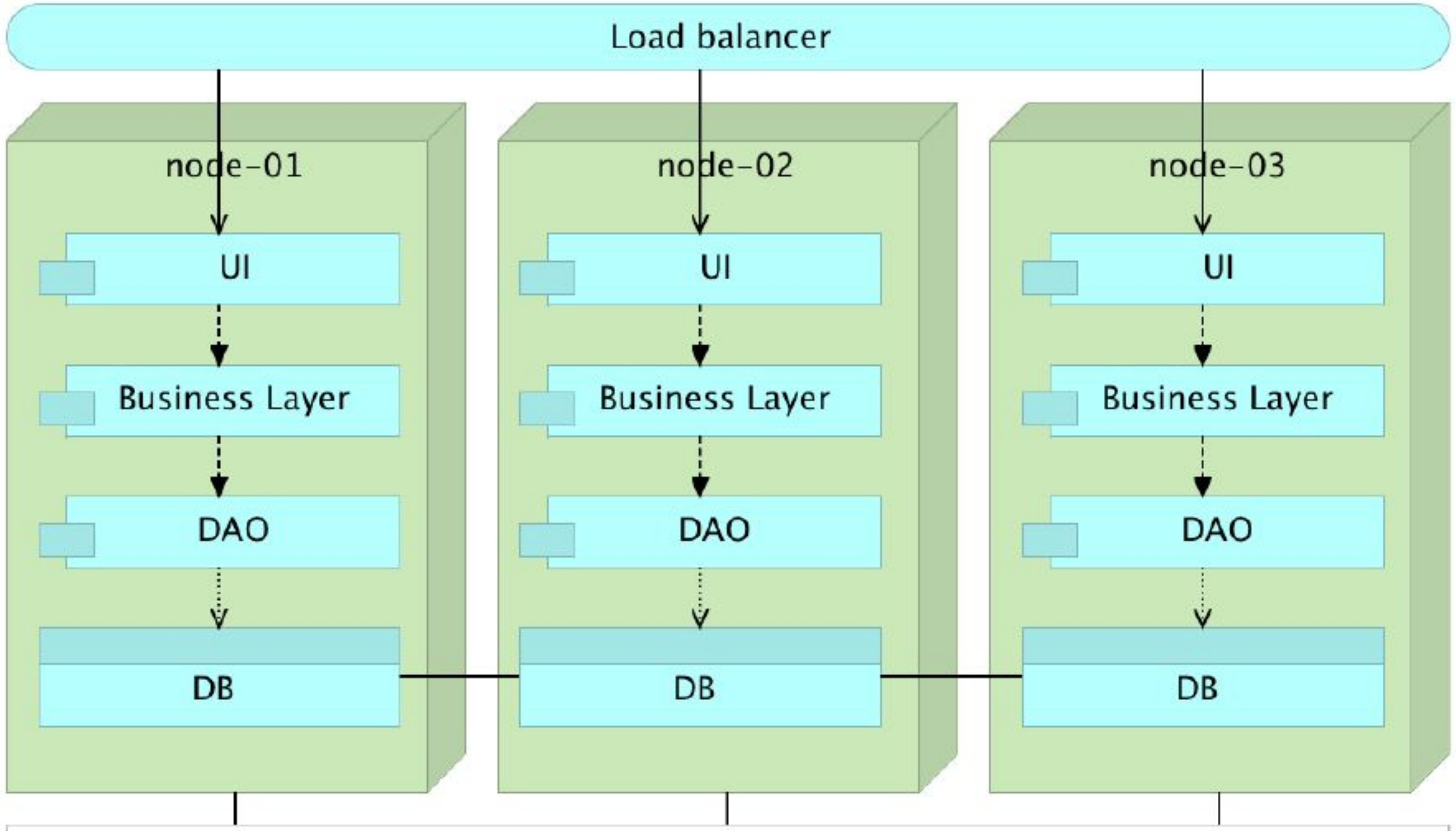
Monolithic Applications



When an application is relativelly small, splitting it into horizontal layers is a good idea. It provides a separation that makes development faster and easier as well as a separation based on type of the task code should do.

Code.Hub

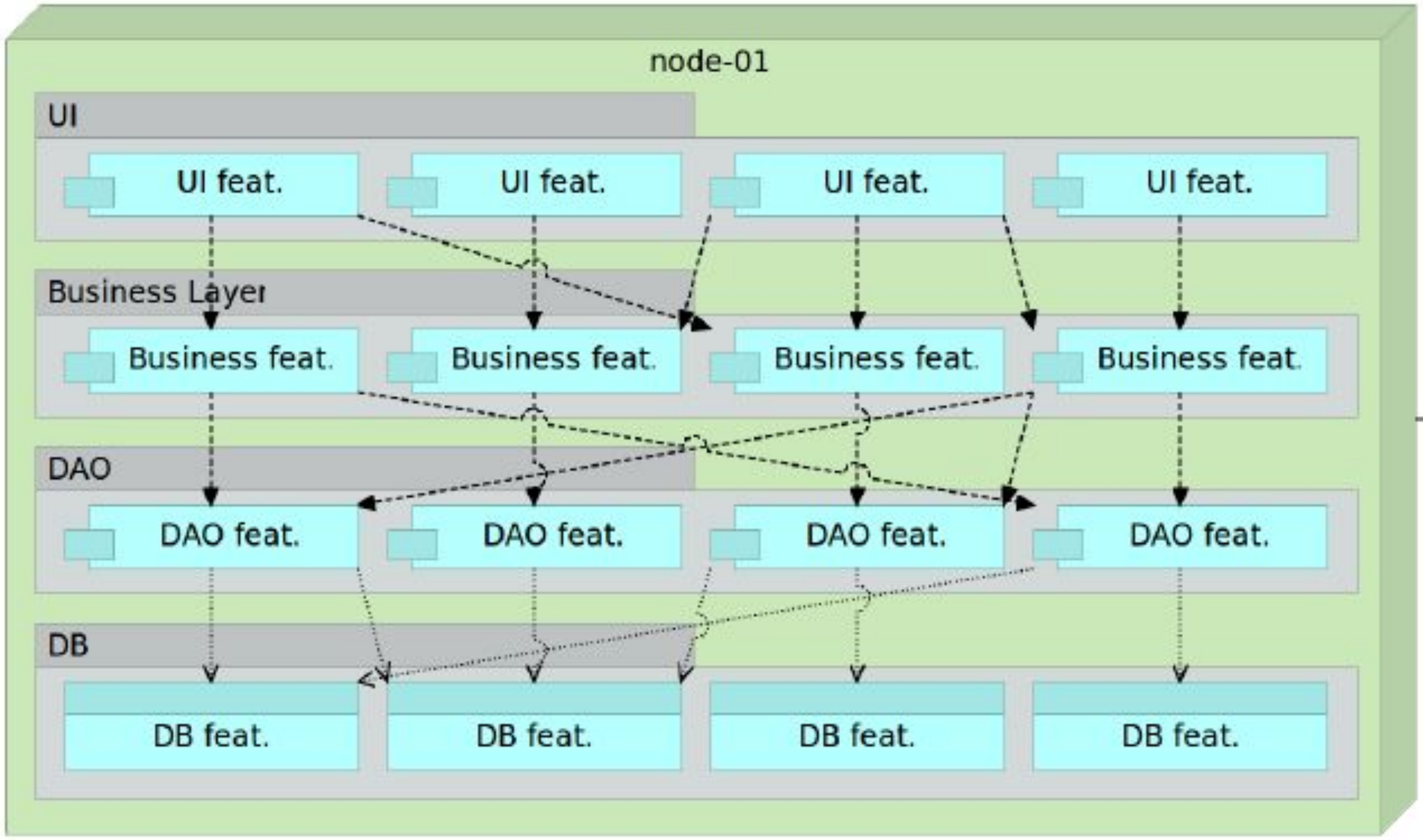
Scaling a Monolithic Application



Scaling monolithic applications is very resource inefficient since everything needs to be duplicated on multiple nodes. There is no option to detect bottlenecks and scale or separate them from the rest of the application.

Code.Hub

Monolithic Application with Increased Number of Features



When an application becomes bigger and the number of features increase, initial design based on horizontal layers becomes less efficient. Tight coupling between separate features, longer paths for potentially simple solutions, increased complexity, increased development and testing time, and so on.

Code.Hub

Scaling a Monolithic Application

- Several Monolith instances behind a Load Balancer
- Pros
  - Quickest path to scale
  - High availability
- Cons
  - Routing traffic complexity
  - Very large code base
  - Change cycles tied together
  - Limited scalability
- Overall… Lack of modularity

Monolith Downsides

- Spaghetti code
- Full deployment even for the slightest change
- Developers often violate the layer boundaries
- Classes often “leak” their implementation
- Hard to work with multiple teams
- Hard to manage
- System complexity increases
- Support is getting more complex
- No Tests or Testing is limited
- Bugs
- Tech stack becomes outdated
- Release/Testing process

Service Oriented Architecture

Code.Hub

Service Oriented Architecture

Let’s talk about tomatoes! How to buy tomatoes?

1. Look for tomato sellers
   - Yellow Pages as it contain companies that are selling tomatoes, their location, and contact information
2. Find the service offered according to my needs
   - Where, when and how can I buy tomatoes?
3. Buy the tomatoes
   - Do the transaction

Service Oriented Architecture

How to access the service?

1. Lookup the Service Provider
    - Registry: contain providers that are selling services, their location, and contact information
2. Find the service offered according to my needs
    - Where, when and how can I get the service?
3. Access the service
    - Do the transaction

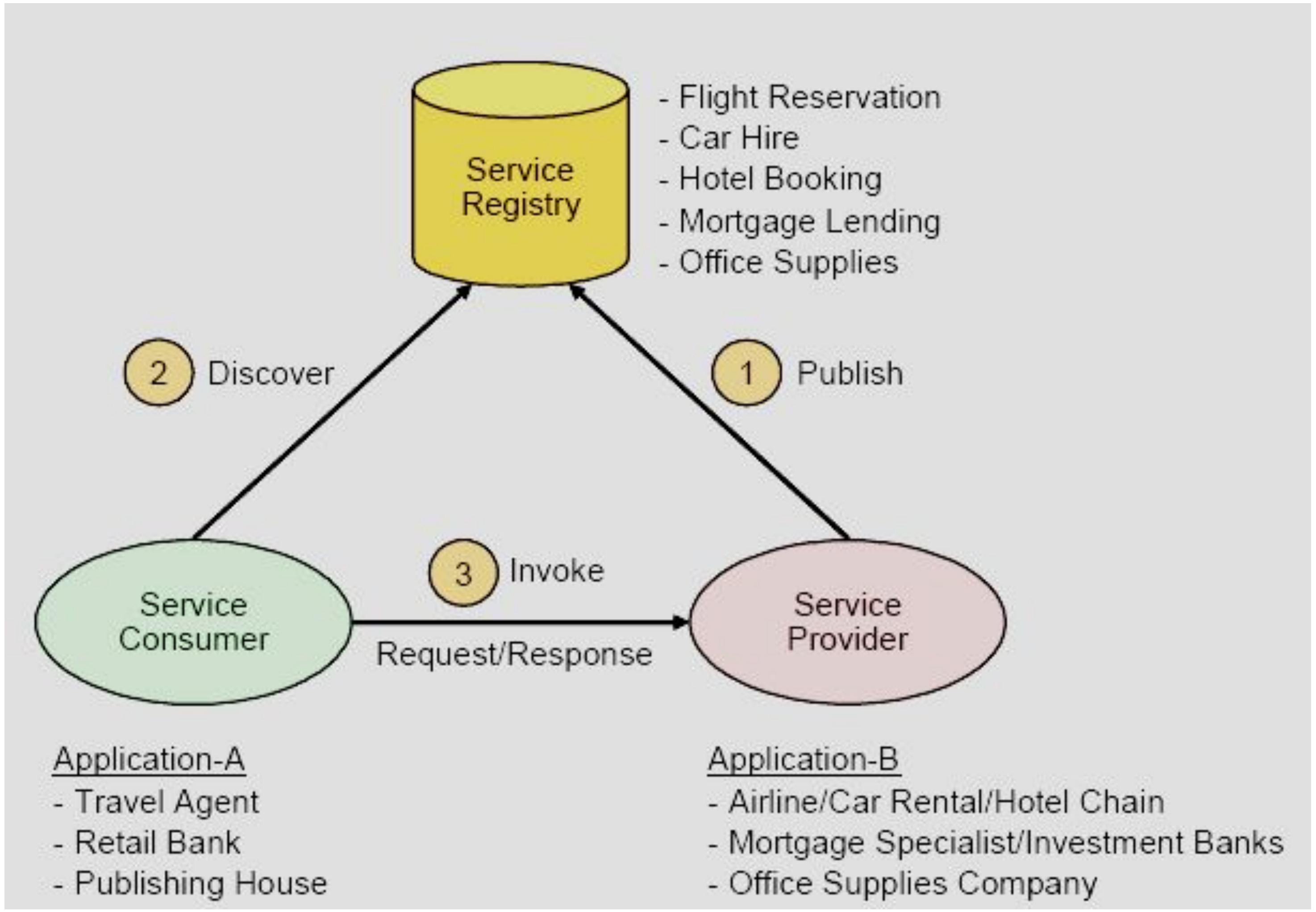
What is Service Oriented Architecture?

- Service Oriented Architecture is an architectural approach in software development where the application is organized as "Services".
- Services are a group of methods that contain the business logic to connect a DB or other services. Methods have clearly defined and published methods for use by the clients as a black box.
- So, what is a black box? It's nothing but a system or an object that can be viewed in terms of its input, output and transfer characteristics without knowledge of its internal workings.
- Across the platforms these methods can be accessed, no matter what your client developing a UI in C# or Java or any other technology.
- It decouples the business services from the technical services, in other words the service methods having the business logic is not coupled with the specific programming language, both will react independently.

SOA Overview

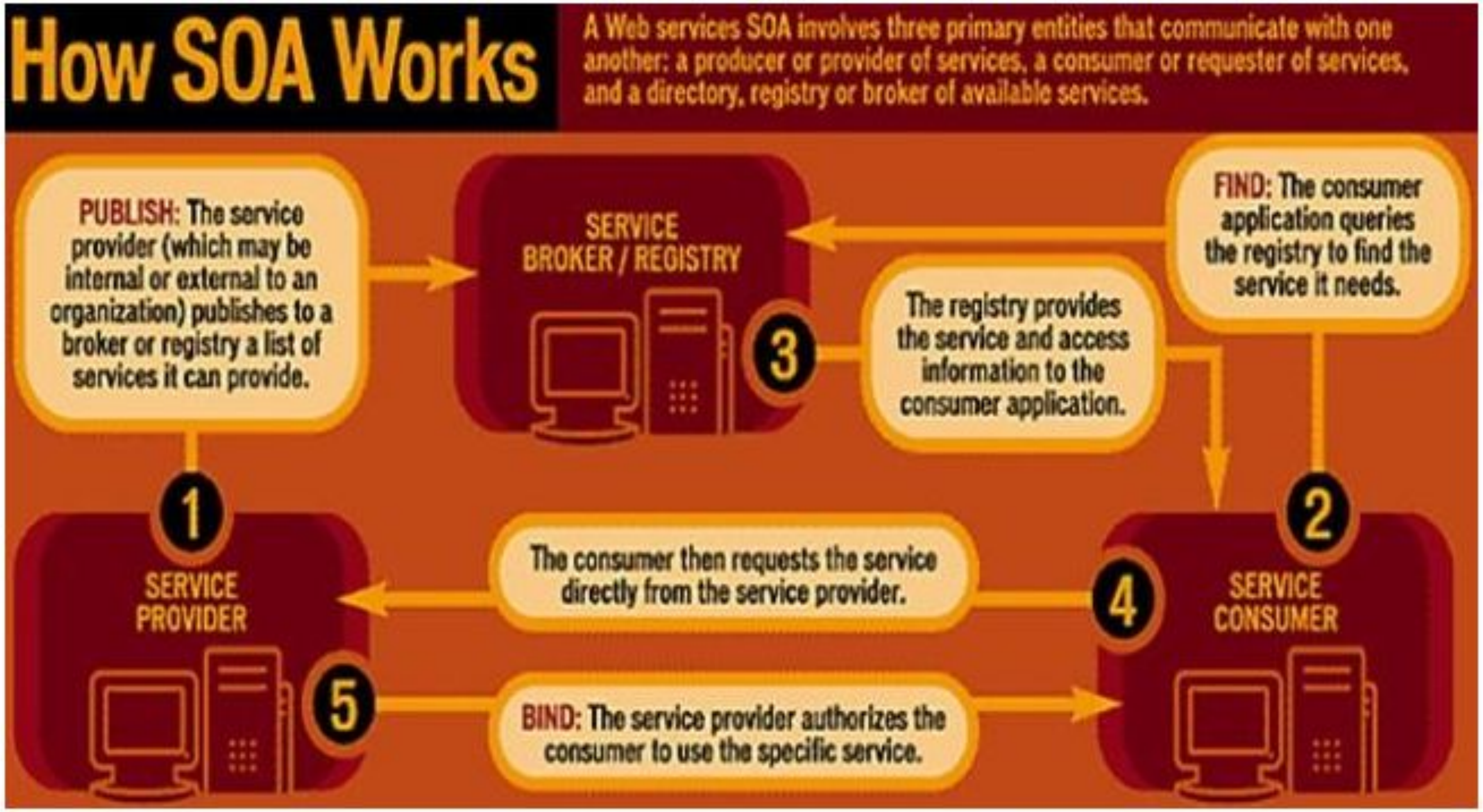
- A Service Oriented Architecture is based on four key abstractions:
  - An Application Front End
  - A Service
  - A Service Repository
  - A Service Bus

SOA Components and Operations



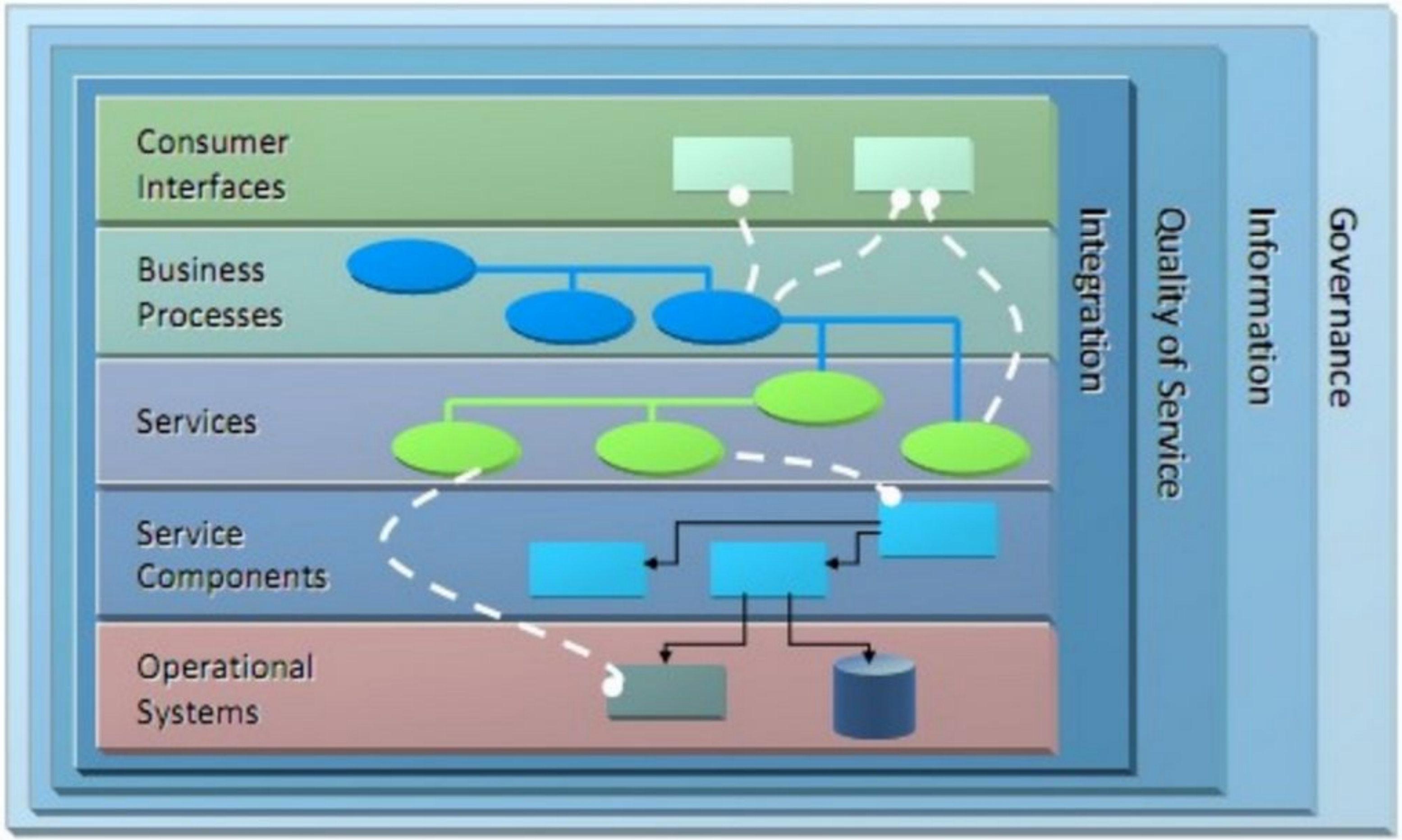
Code.Hub

How SOA Works?


How SOA Works
A Web services SOA involves three primary entities that communicate with one another: a producer or provider of services, a consumer or requester of services, and a directory, registry or broker of available services.
PUBLISH: The service provider (which may be internal or external to an organization) publishes to a broker or registry a list of services it can provide.
SERVICE BROKER / REGISTRY
FIND: The consumer application queries the registry to find the service it needs.
3
The registry provides the service and access information to the consumer application.
1
2
SERVICE PROVIDER
The consumer then requests the service directly from the service provider.
4
SERVICE CONSUMER
5
BIND: The service provider authorizes the consumer to use the specific service.

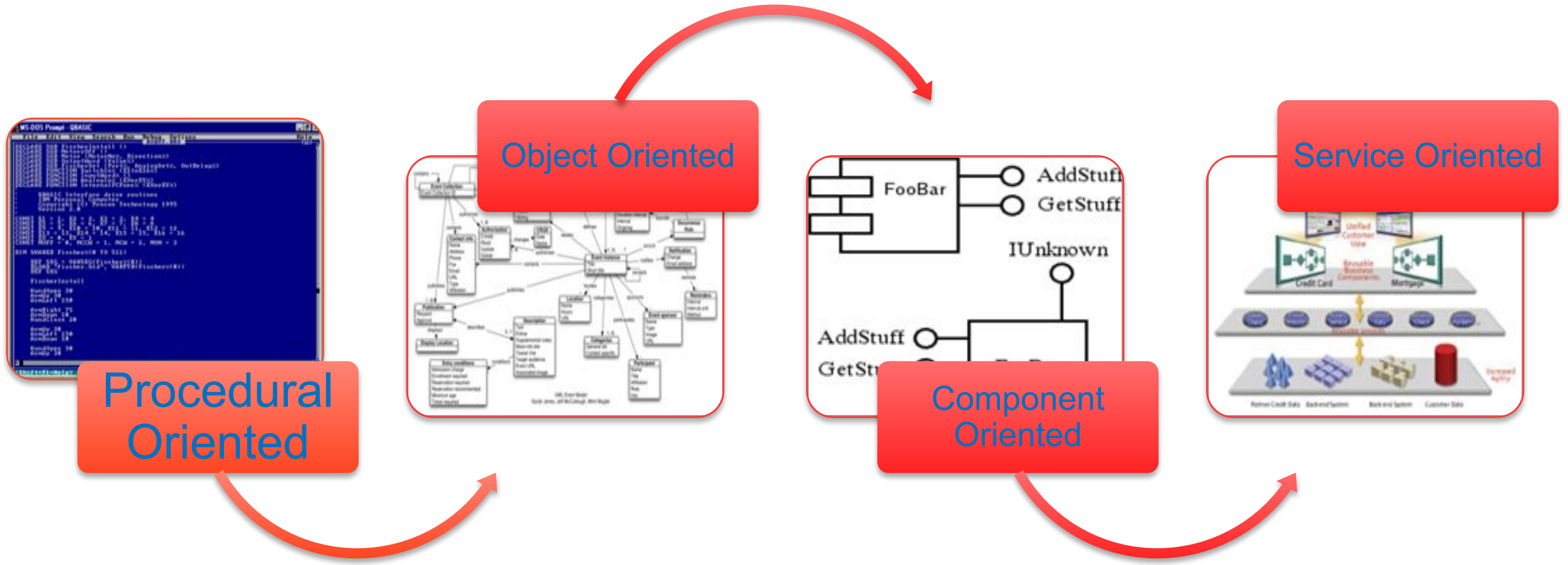
Code.Hub

Open Group SOA Reference Architecture (SOA RA)



Code.Hub

How did we get to SOA?



Code.Hub

SOA Challenges

SOA Alphabet Soup

What Would You Choose?

DISCO BEPL MoM

EAI rpc OASIS XML UDDI

XPath WS-I SODA

SOA ReST JBI XSD

MESSAGING SOAP XSLT

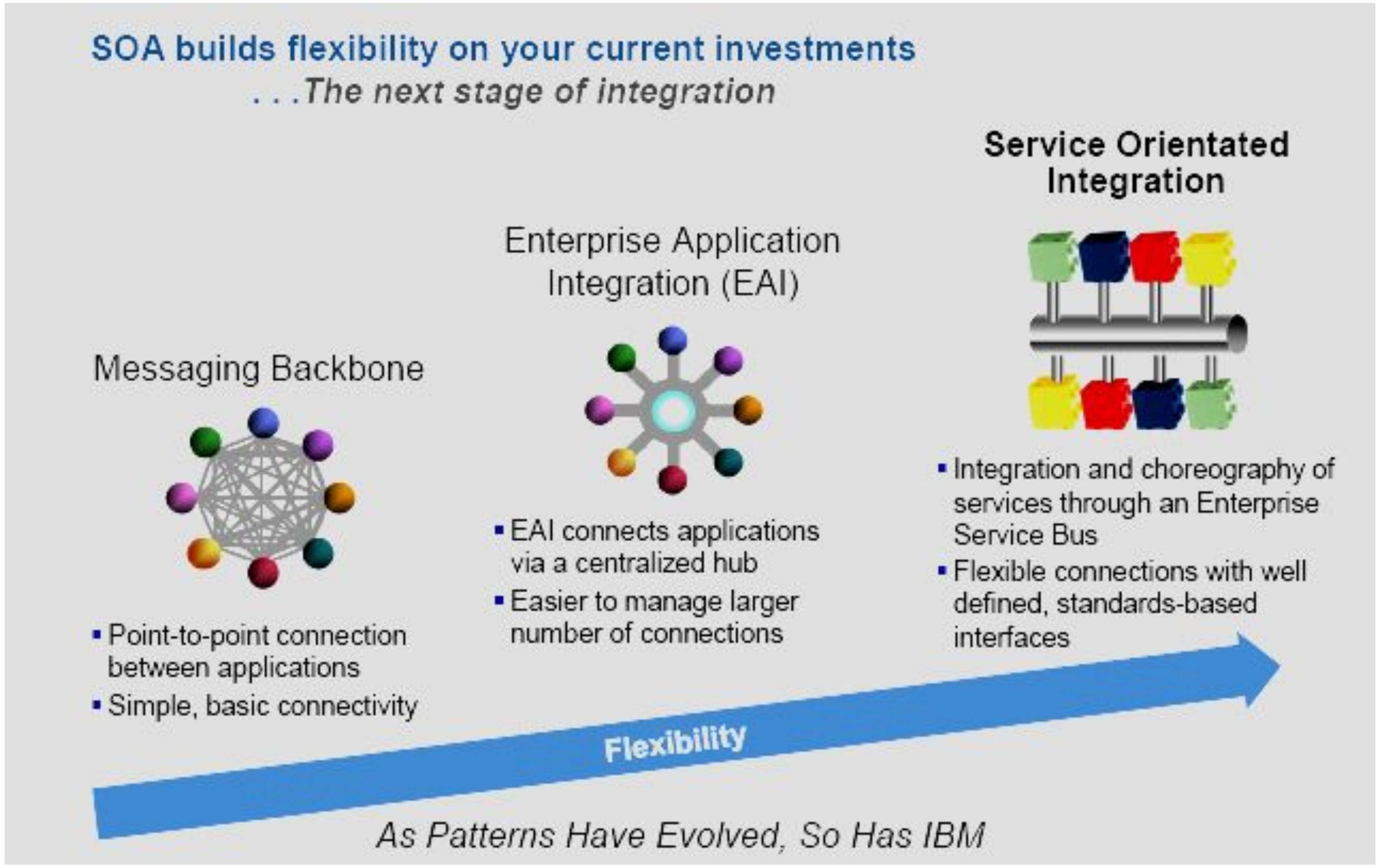
SAML Web Service Axis DIME

WSDL SAX Digital Signature DTD

BEPL4WS DOM XSLT BPM Schema

Code.Hub

Evolution of Application Integration Patterns



SOA builds flexibility on your current investments

. . .The next stage of integration

Messaging Backbone

- Point-to-point connection between applications
- Simple, basic connectivity

Enterprise Application Integration (EAI)

- EAI connects applications via a centralized hub
- Easier to manage larger number of connections

Service Orientated Integration

- Integration and choreography of services through an Enterprise Service Bus
- Flexible connections with well defined, standards-based interfaces

As Patterns Have Evolved, So Has IBM

Code.Hub

Microservices

Code.Hub

Microservices

MONOLITHS
Hard to deliver, even harder to test and impossible to maintain

Code.Hub

Microservices

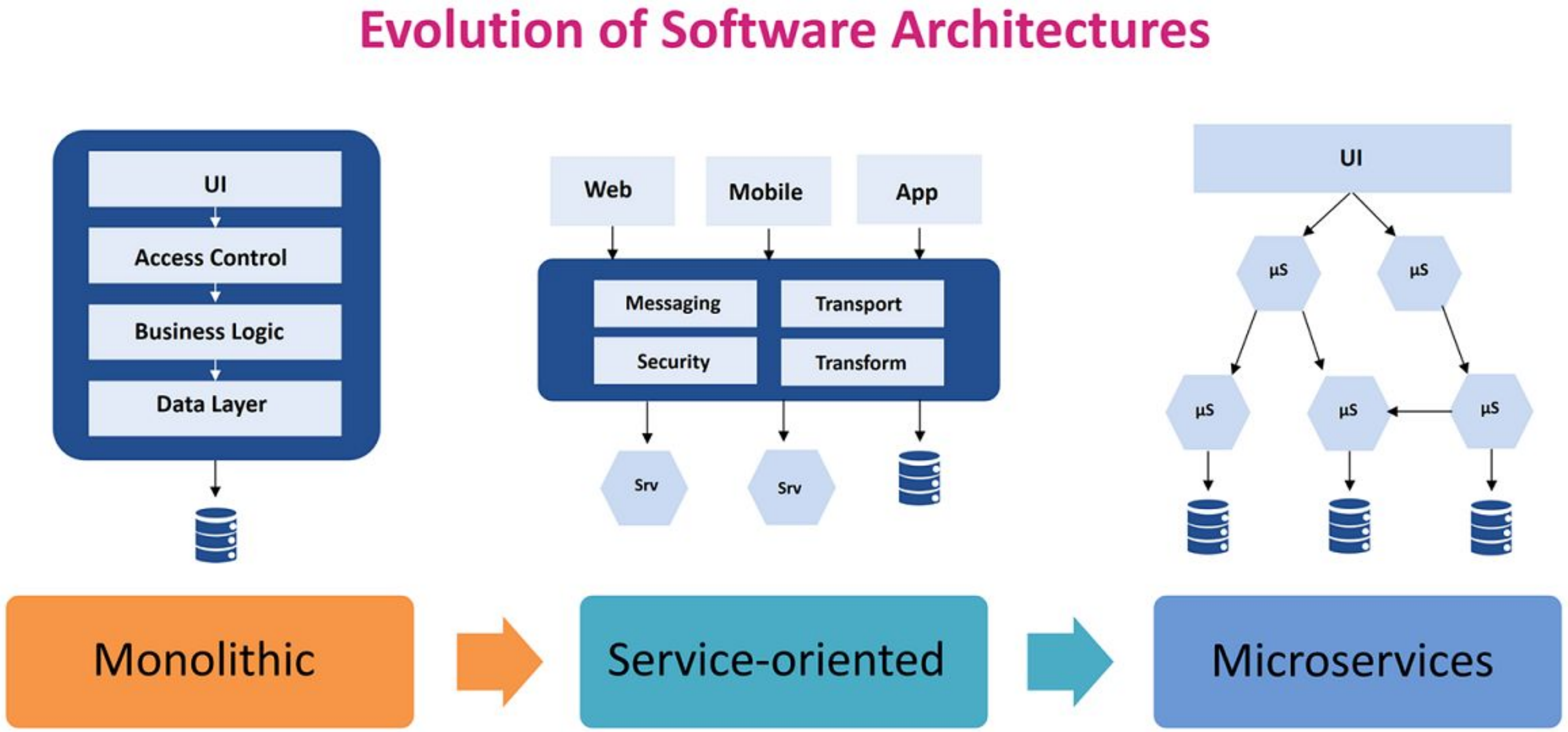
Code.Hub

Road to Microservices

1990s and earlier	2000s	2010s
Coupling		
Pre-SOA (monolithic) Tight coupling	**Traditional SOA** Looser coupling	**Microservices** Decoupled

Code.Hub

Road to Microservices



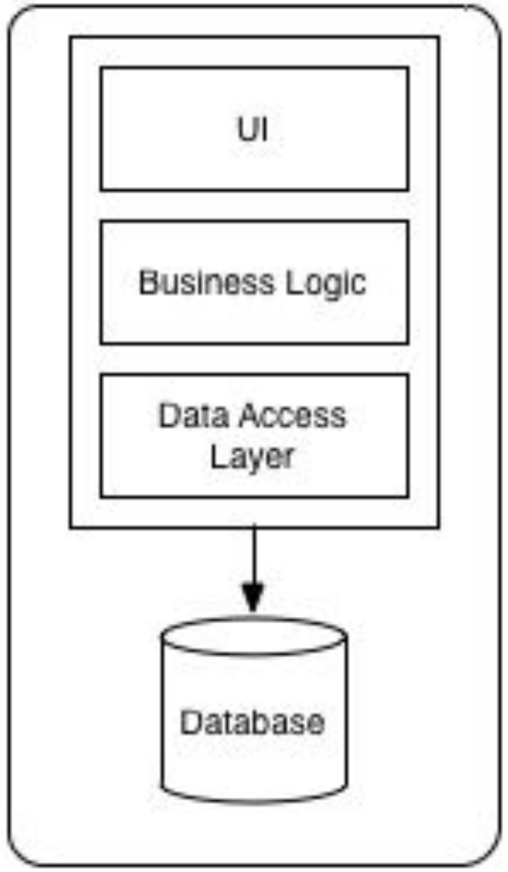
Code.Hub

Micro services

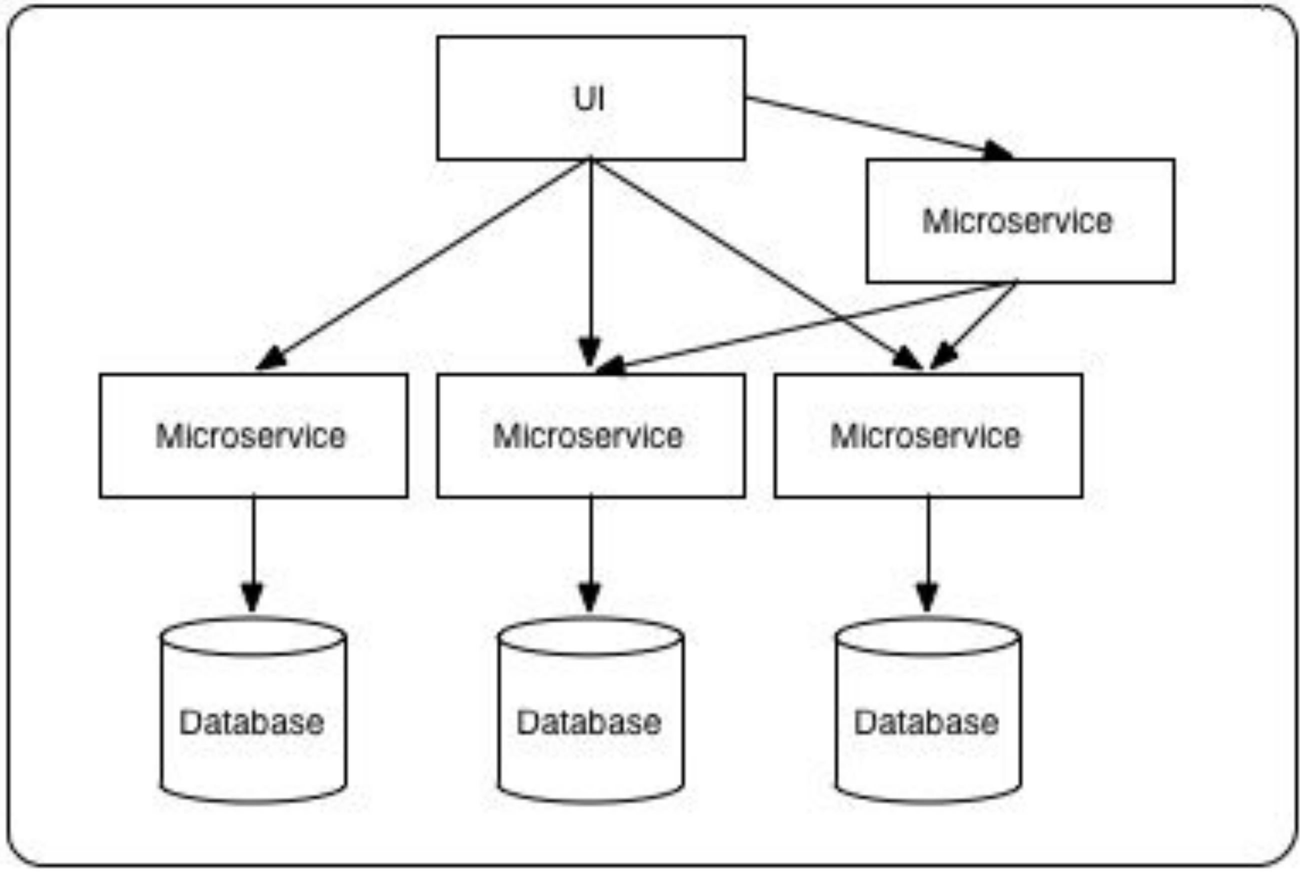
- In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.
- These services are built around business capabilities and independently deployable by fully automated deployment machinery.
- There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

Martin Fowler, James Lewis

Microservices



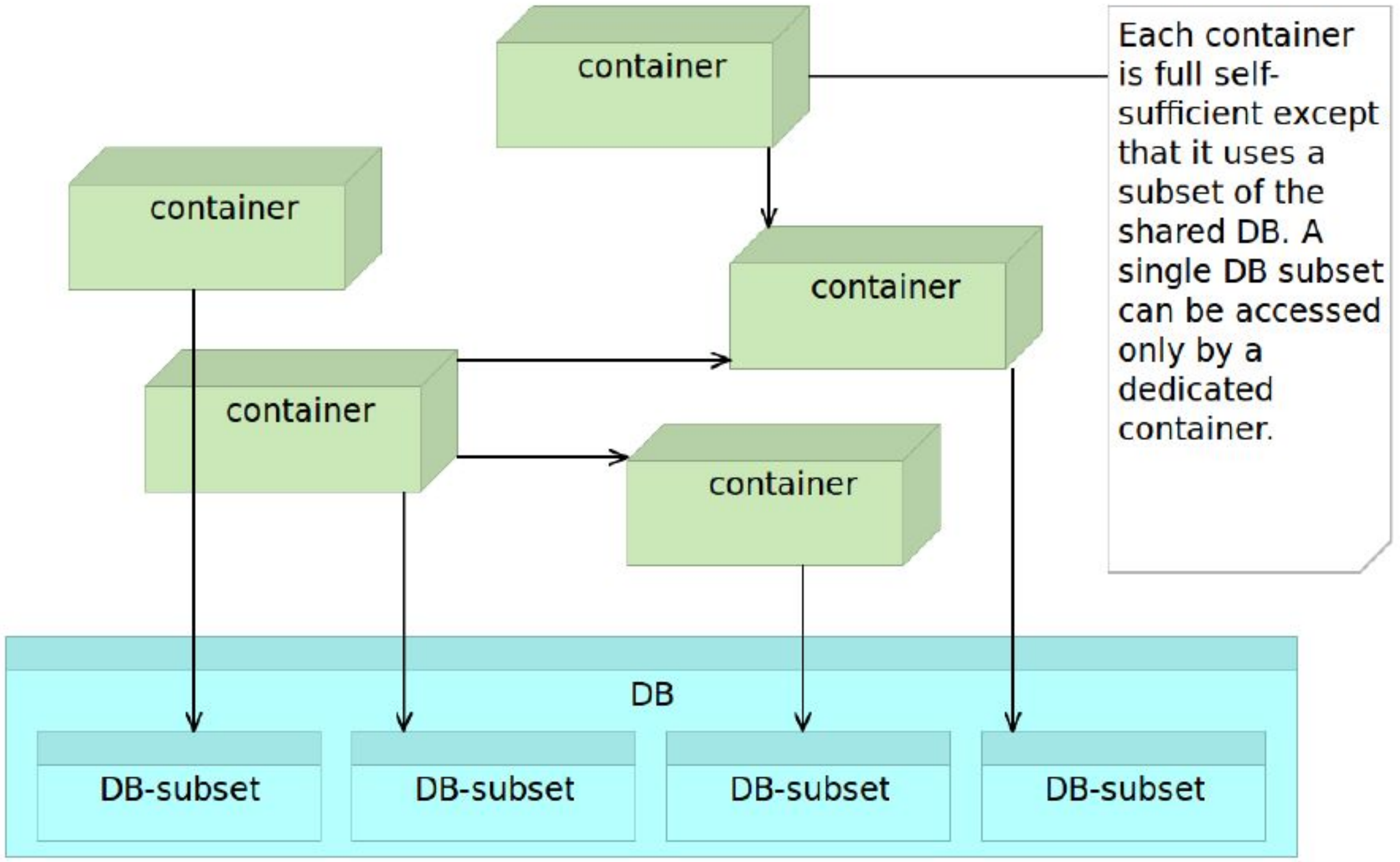
Monolithic Architecture



Microservices Architecture

Code.Hub

Microservices

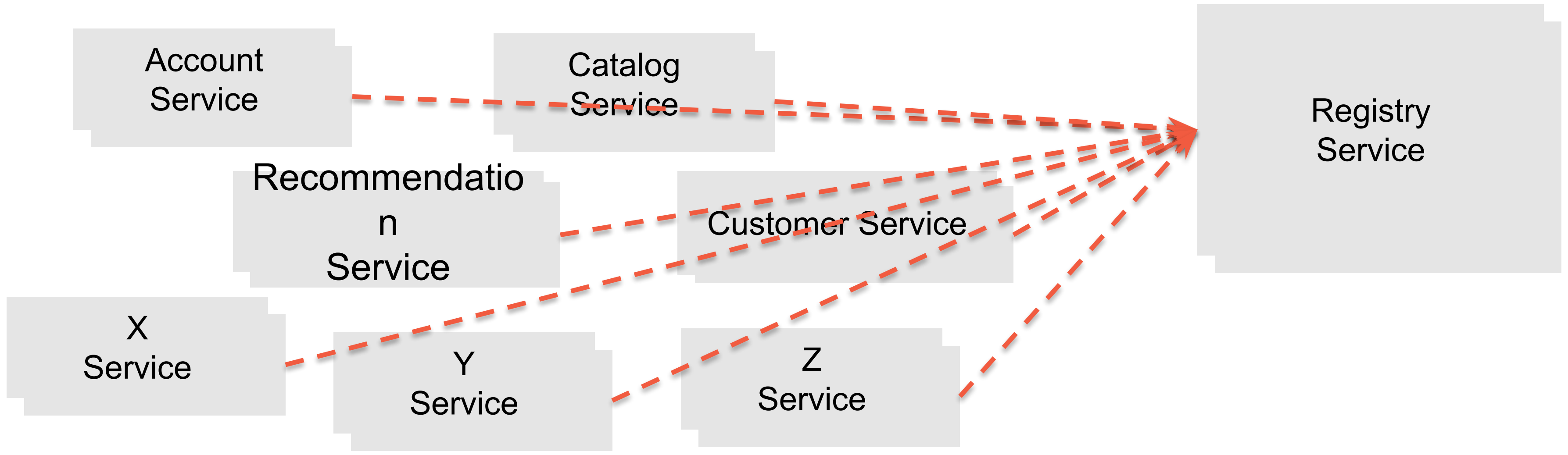

container
container
container
container
container
Each container is full self-sufficient except that it uses a subset of the shared DB. A single DB subset can be accessed only by a dedicated container.
DB
DB-subset
DB-subset
DB-subset
DB-subset

Microservices

- Many small modules with specific functionality
- More than one codebase
- Every microservice is a separate deployment
- Every microservices has its own Database service, or not…
- **Communication based on well known protocols and data serialization mechanisms**
- Module independence

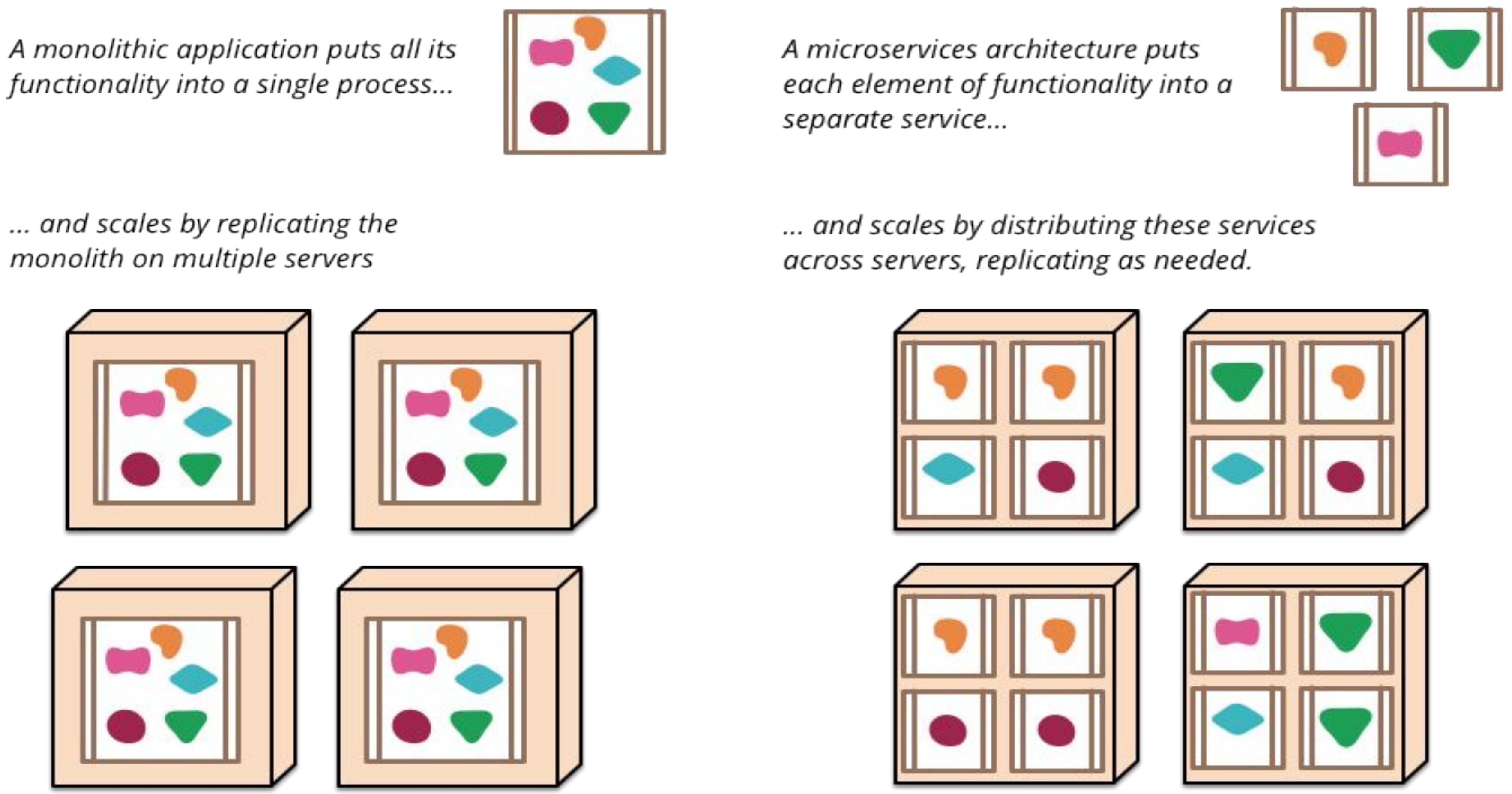
Microservices Discovery

- 100s of Microservices
- Need a Service Metadata Registry (Discovery Service)



Code.Hub

Monoliths Scalability vs Microservices Scalability


A monolithic application puts all its functionality into a single process...
... and scales by replicating the monolith on multiple servers
A microservices architecture puts each element of functionality into a separate service...
... and scales by distributing these services across servers, replicating as needed.

Code.Hub

Microservices Benefits

- Modelled around the business domain
- Deployment automation culture
- Hides implementation details
- Decentralization
- Option to use multiple languages and/or frameworks
- Separate deployment, Separate monitoring
- Problem isolation
- Enables the continuous delivery and deployment of large, complex applications
- Eliminates any long-term commitment to a technology stack

Microservices Drawbacks

- Developers must deal with the additional complexity of creating a distributed system.
- Deployment complexity. In production, there is also the operational complexity of deploying and managing a system comprised of many different service types.

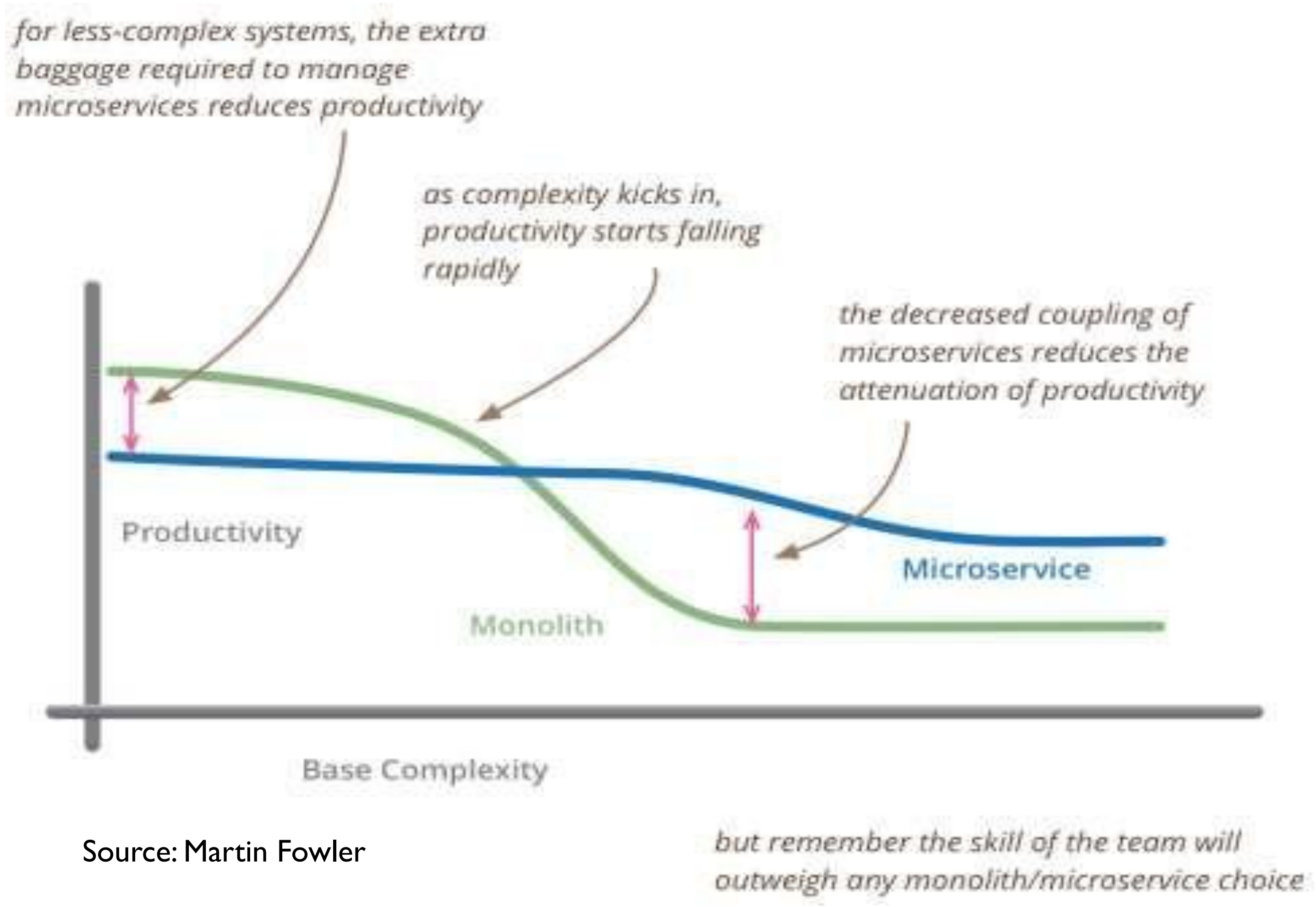
When to Choose a Monolithic Approach

- New business domain, lack of knowledge
- Proof of Concepts
- Lack of qualification
- Fast or Throw-away solutions
- Low budget

When to Choose a Microservices Approach

- Needs to scale
- You understand business domain
- Big budget
- Ready to invest into infrastructure and CI/CD processes
- Experienced and highly qualified team

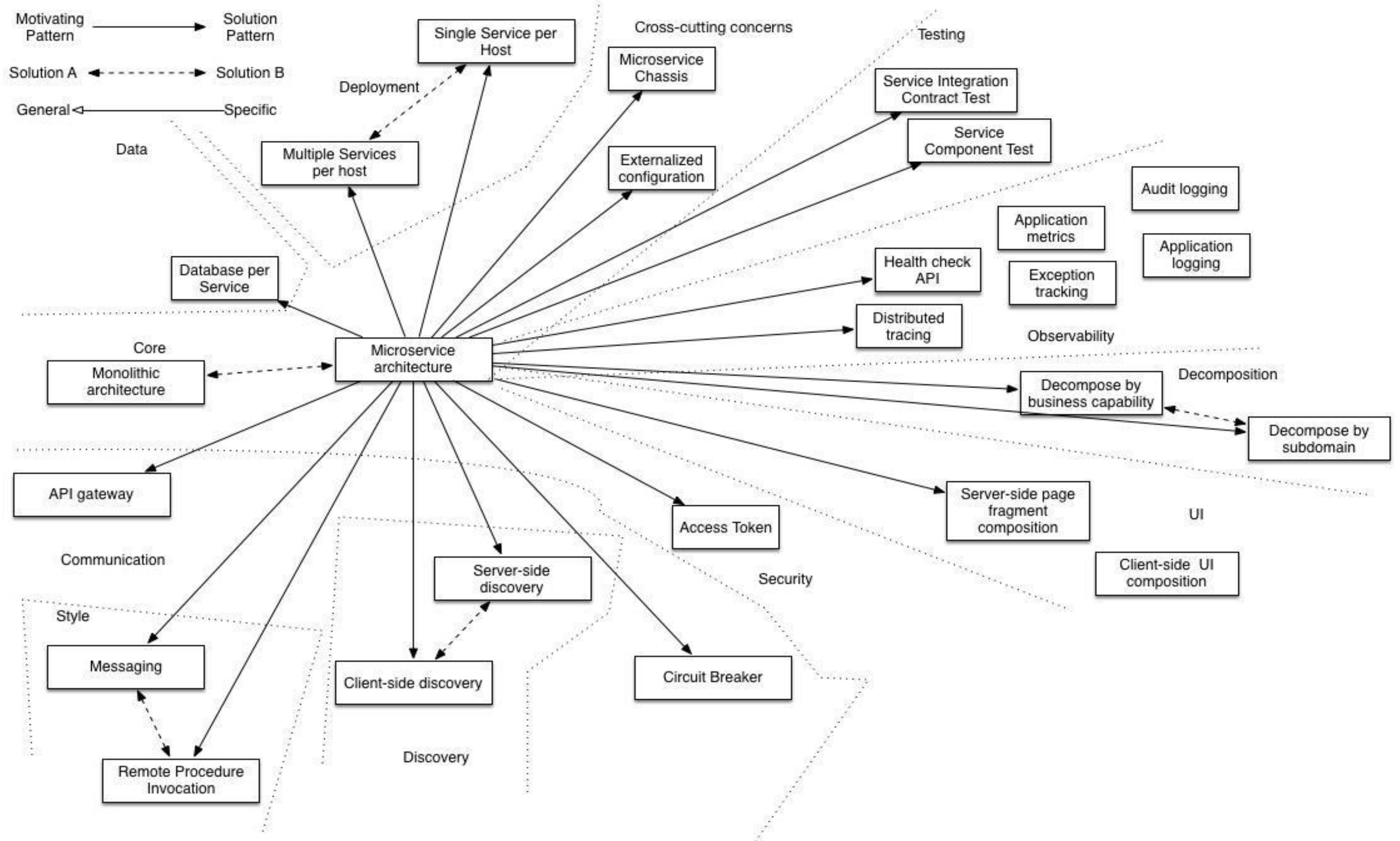
Monolith – Microservices Comparison



Source: Martin Fowler

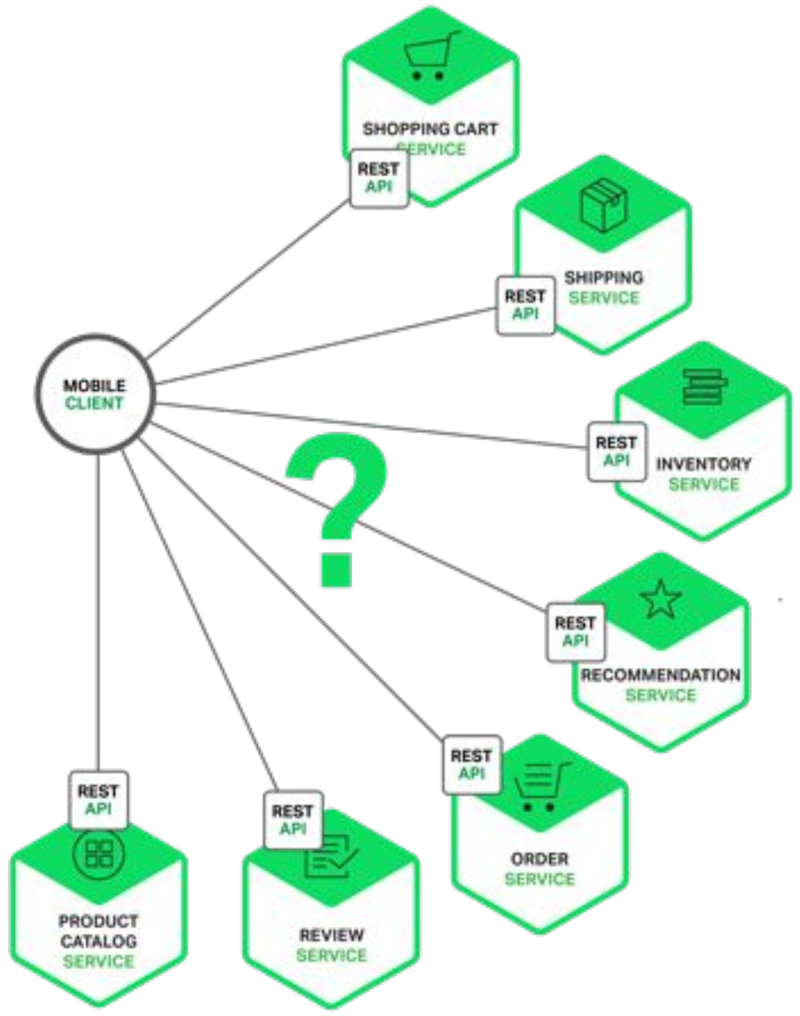
Code.Hub

Microservices Architectural Patterns

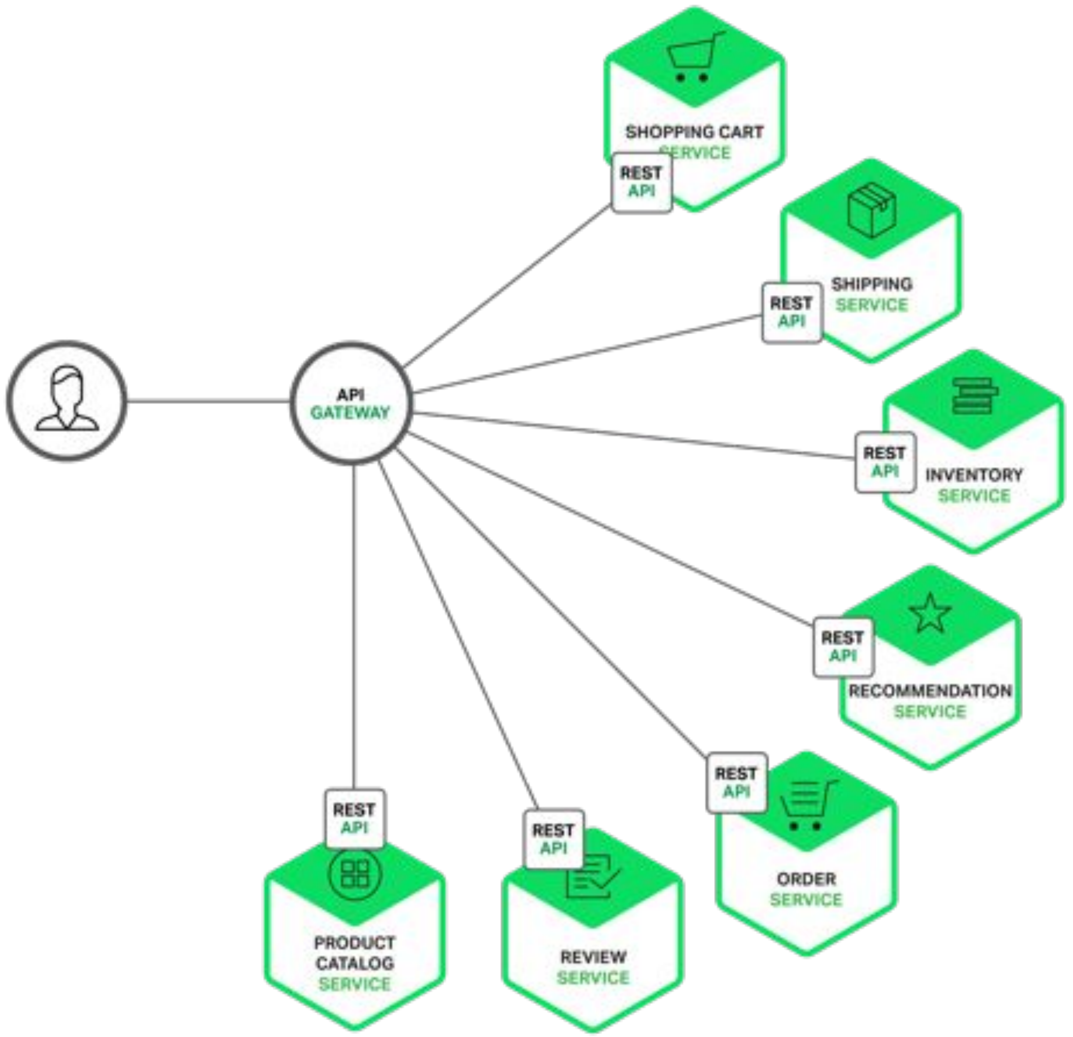


Code.Hub

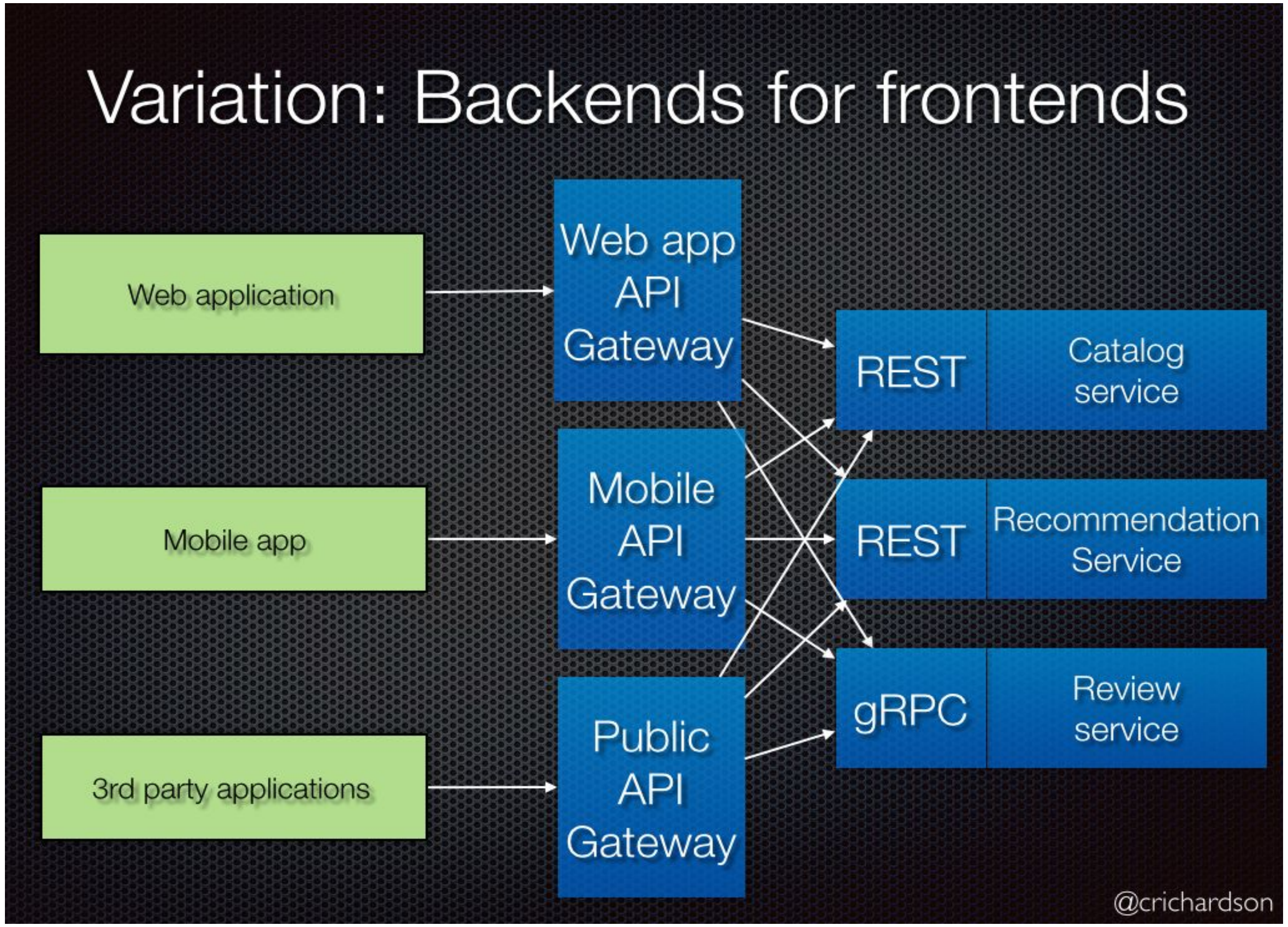
Microservices Client Configuration vs



Microservices API Gateway

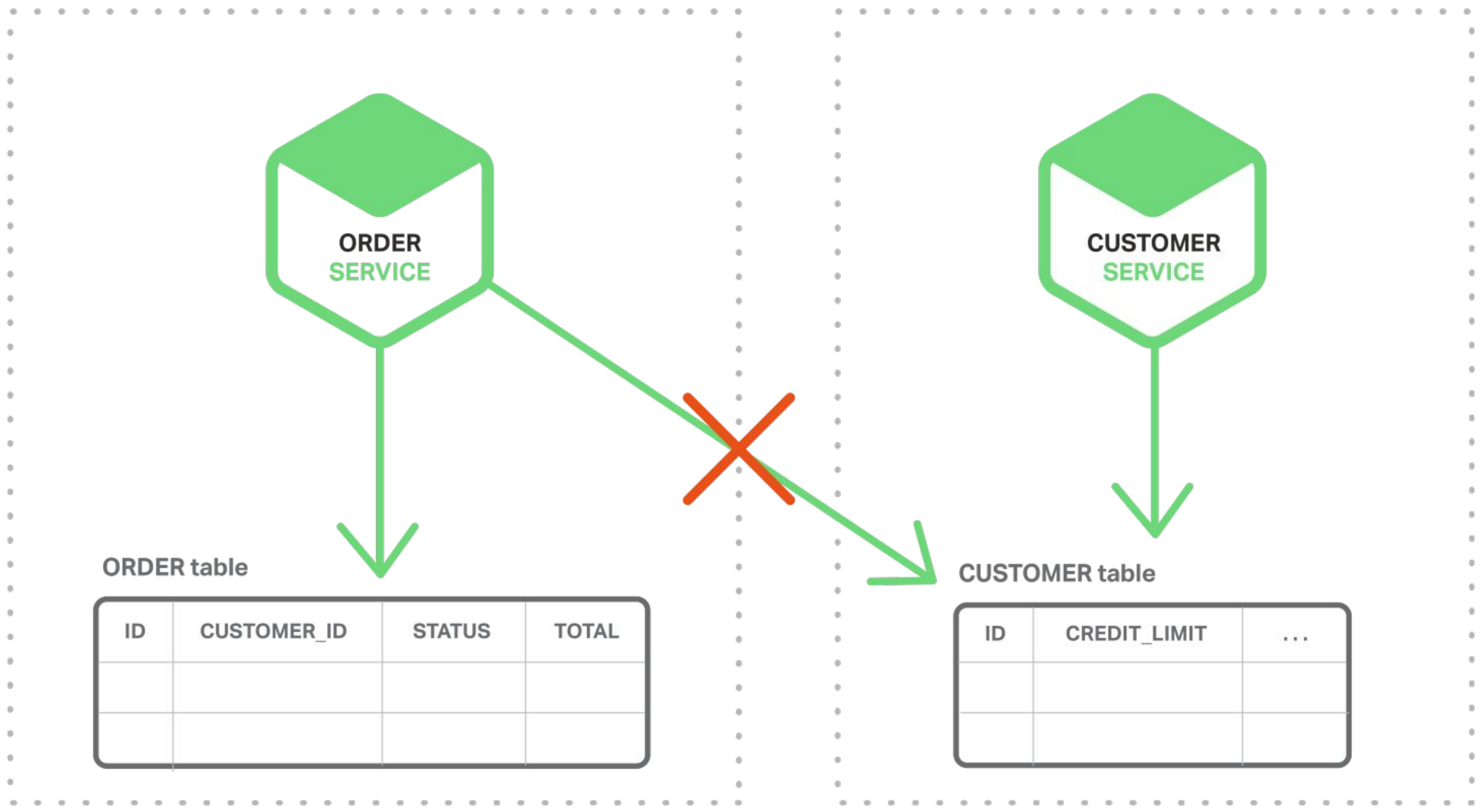


Microservices Backend for Frontend



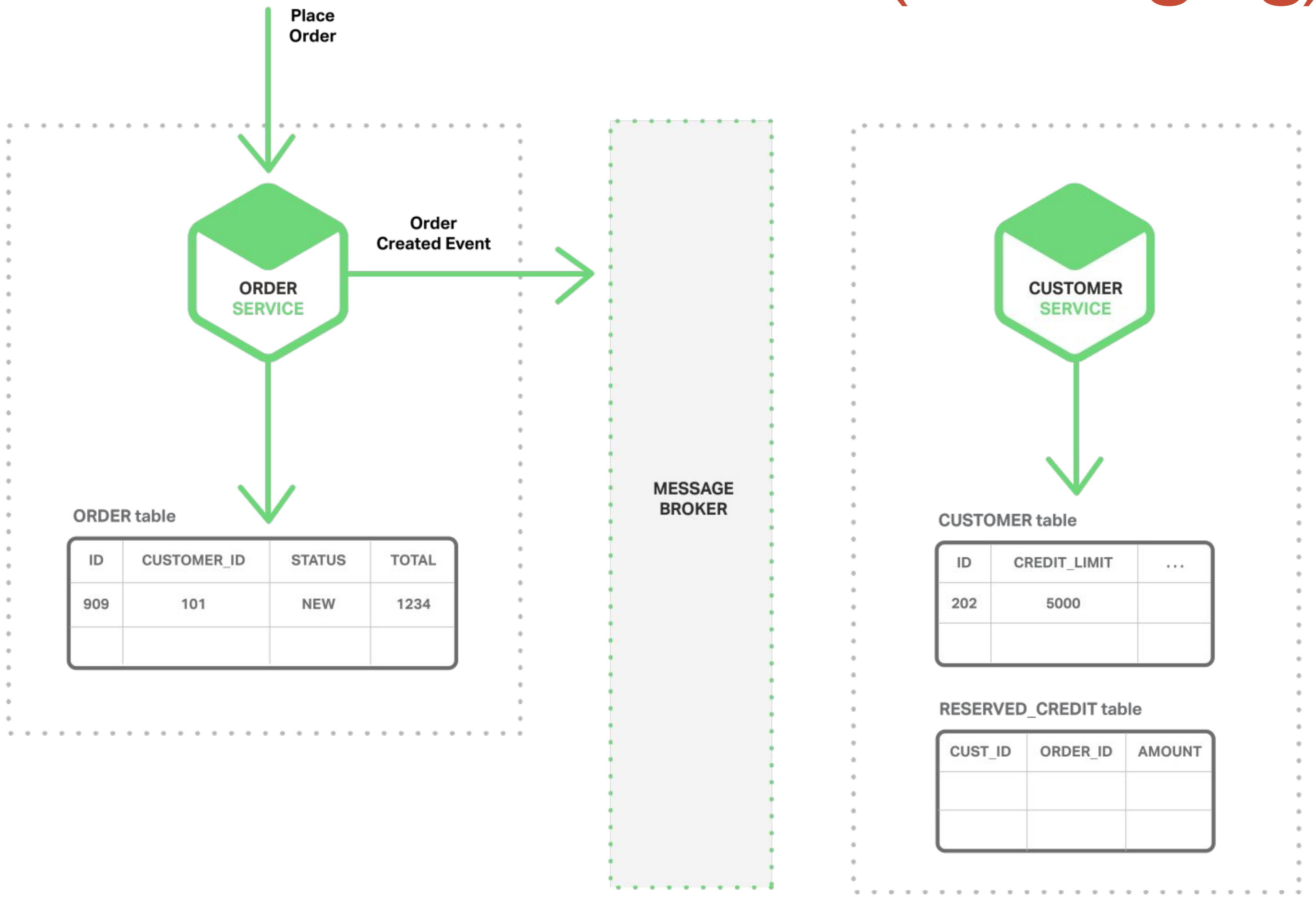
Code.Hub

Communication between Services



Code.Hub

Event-Driven Communication (Messaging)



ORDER table

ID	CUSTOMER_ID	STATUS	TOTAL
909	101	NEW	1234

CUSTOMER table

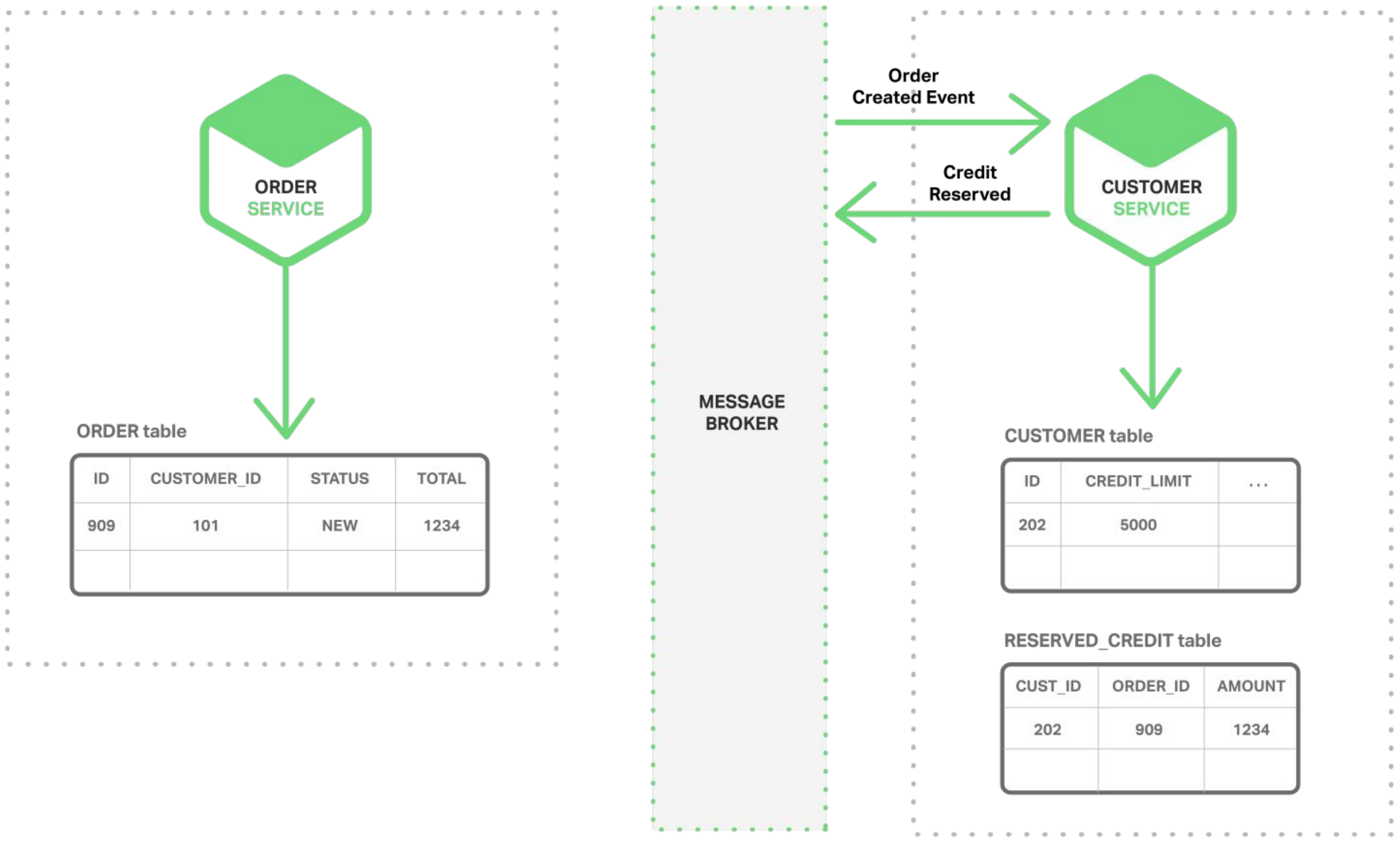
ID	CREDIT_LIMIT	. . .
202	5000	

RESERVED_CREDIT table

CUST_ID	ORDER_ID	AMOUNT

Code.Hub

Event-Driven Communication (Messaging)



ORDER table

ID	CUSTOMER_ID	STATUS	TOTAL
909	101	NEW	1234

CUSTOMER table

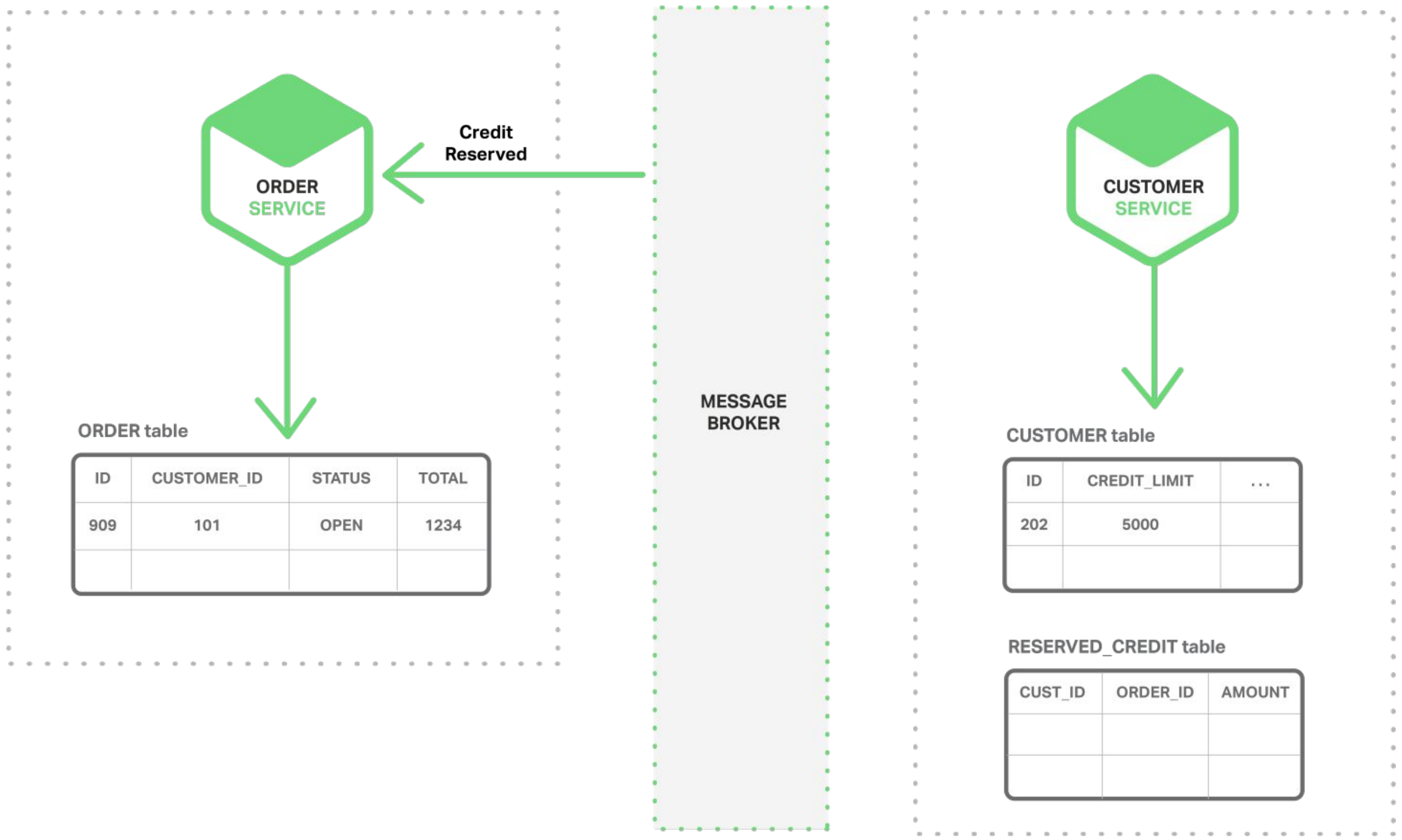
ID	CREDIT_LIMIT	...
202	5000	

RESERVED_CREDIT table

CUST_ID	ORDER_ID	AMOUNT
202	909	1234

Code.Hub

Event-Driven Communication (Messaging)



ORDER table

ID	CUSTOMER_ID	STATUS	TOTAL
909	101	OPEN	1234

CUSTOMER table

ID	CREDIT_LIMIT	...
202	5000	

RESERVED_CREDIT table

CUST_ID	ORDER_ID	AMOUNT

Event-Driven Communication (Messaging)

- Benefits:
  - Loose coupling since it decouples client from services
  - **Improved availability since the message broker buffers messages until the consumer can process them**
  - Can scale well
  - Lots of market solutions
- Drawbacks:
  - Additional complexity of message broker, which must be highly available
  - Requires qualification
  - Deployment and support complexity(monitoring and tools)

Designing for Microservices

Code.Hub

Common Characteristics

- Componentization via Services
- Organized around Business Capabilities
- Products not Projects
- Smart endpoints and dumb pipes
- Decentralized governance
- Decentralized data management
- Infrastructure automation
- Design for failure
- Evolutionary design

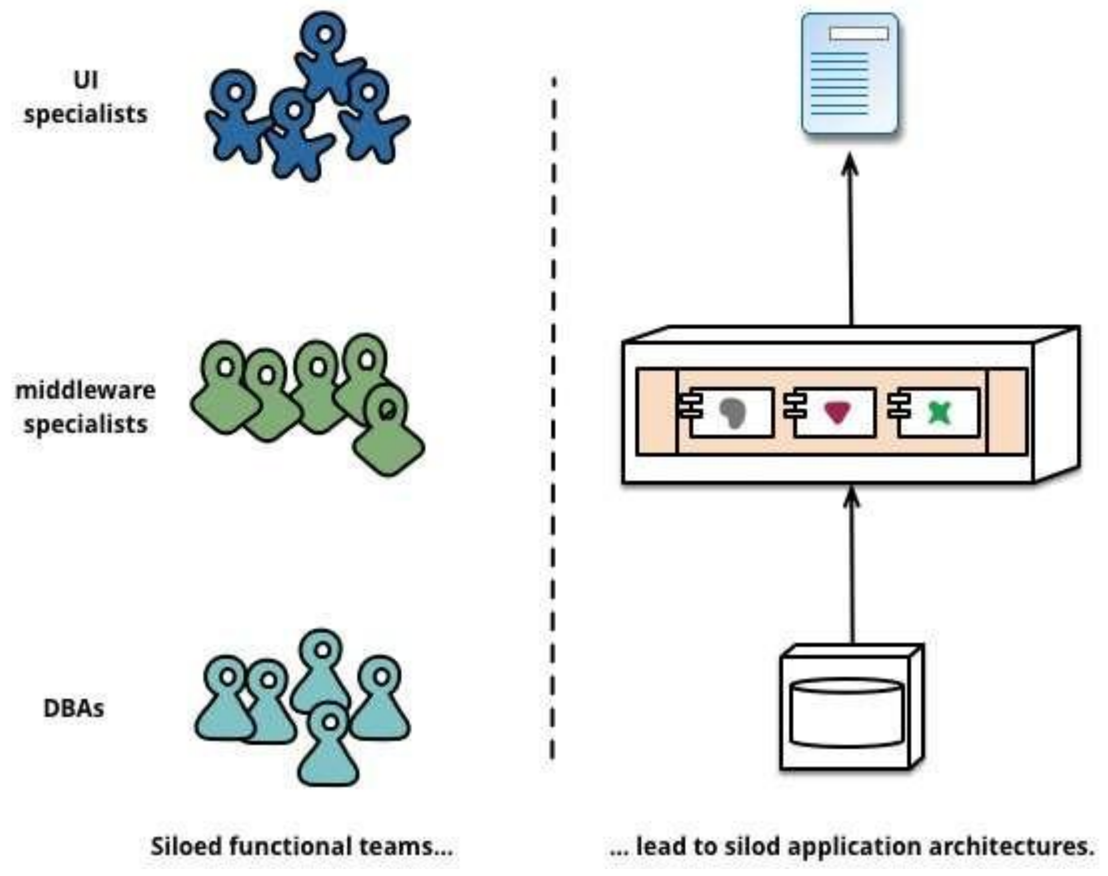
Componentization via Services

- Services as components rather than libraries
- Services avoid tight coupling by using explicit remote call mechanisms
- Services are independently deployable and scalable
- Each service also provides a firm module boundary
  - business or technical
  - even allowing for different services to be written in different programming languages
  - they can also be managed by different teams
- A Microservice may consist of multiple processes
  - that will always be developed and deployed together
  - Ex : an application process and a database that's only used by that service

Organized around Business Capabilities

Any organization that designs a system will produce a design whose structure is a copy of the organization's communication structure.

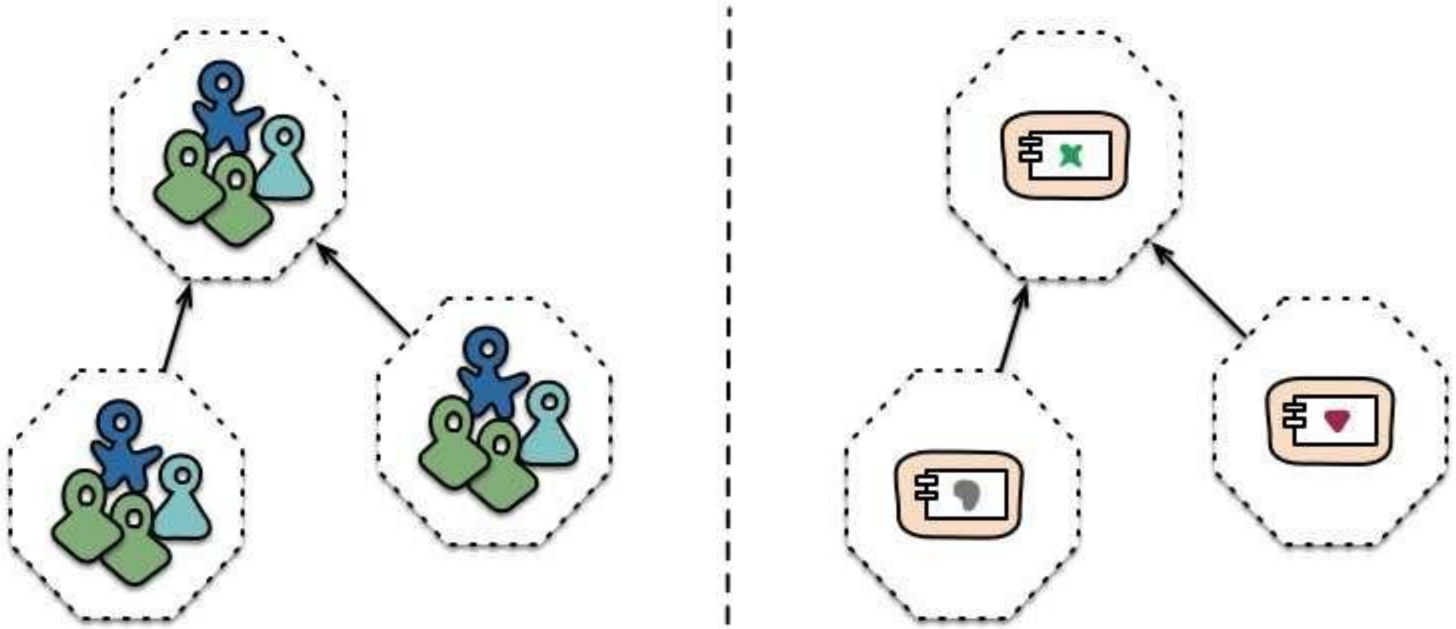
Melvyn Conway, 1967


UI specialists
middleware specialists
DBAs
Siloed functional teams...
... lead to silod application architectures.

Code.Hub

Organized around Business Capabilities

Microservices to solve Conway's anti-pattern



Cross-functional teams… …organized around capabilities

Products not Projects

- Standard project model:
  - deliver pieces of software which are then considered to be completed,
  - hand over to a maintenance organization and disband the project team
- The Microservices style
  - a team should own a product over its full lifetime
- Amazon :You build => You run it

Smart endpoints and dumb pipes

- Be as decoupled and as cohesive as possible
  - own domain logic,
  - act more as filters in the classical Unix sense
  - using simple RESTish protocols and lightweight messaging
- Smarts live in the services, not in-between the endpoints
  - No central tool / bus that includes sophisticated routing, transformations, process, business rules
- Pre-requisite : turn the chatty in-process communication of the monolith into coarser-grained network messaging

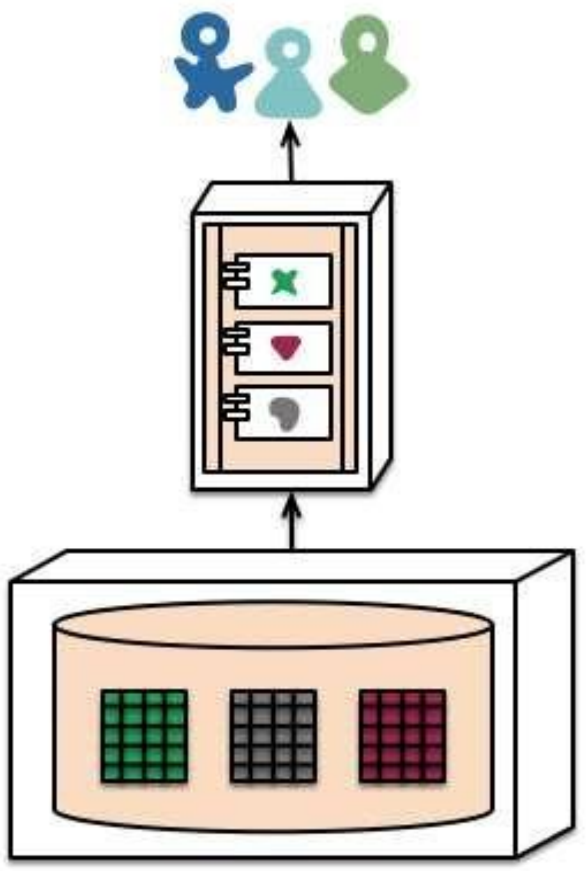
Decentralized Governance

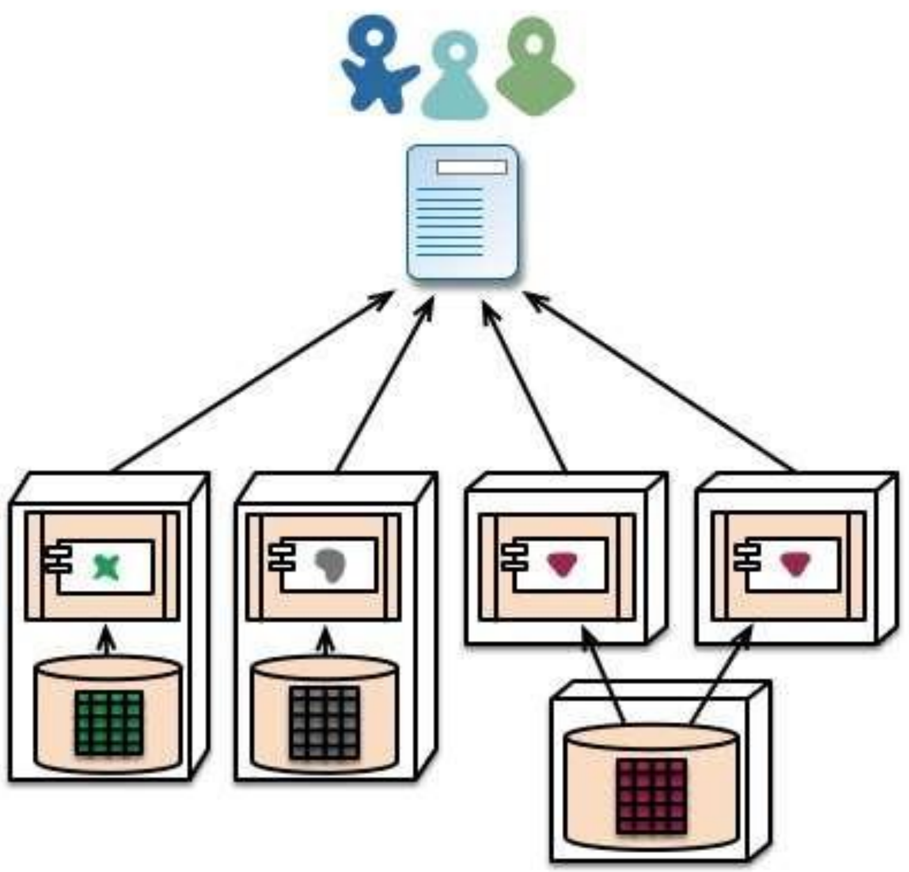
- The common microservices practice is to choose the best tool for each job
- Tools are shared inside and outside the organization
- Focus on common problems: data storage, IPC, infrastructure automation
- E.g. : Netflix opensource libraries
- Favor independent evolution
  - Consumer-driven Service contracts
  - Tolerant Reader Pattern
- Minimal over-heads create an opportunity to have teams responsible for all aspects
- Build their microservices and operate with 24/7 SLAs

Decentralized Data Management

- No unique data model approach
  - the conceptual model differs between microservices
  - Reinforce the separation of concerns
- Decentralized data storage
  - Polyglot persistence is frequent in microservices architectures
- Manage inconsistencies via the business practices in place throughout the organization
- Common design : reversal processes versus 2PC distributed transactions

Decentralized Data Management





Infrastructure Automation

- CI/CD, Real-time monitoring, global system and fine-grained resources dashboards,
- Has become a standard practice
- thanks to public cloud providers,
- but also opensource tools
- a QA stake for monoliths
- a pre-requisite for microservices

Design for Failure

- Any service call can fail
  - Circuit Breaker pattern,
  - Async consumption
    - Best practice : 1 to ZERO sync call
  - E.g.: Netflix Stack
    - Hystrix / Eureka / Ribbon
  - /!\ PRovide interop & implementations for various languages
- Infrastructure prerequisites
  - Monitor and restore at scale
  - Dedicated tooling for simulation
  - Netflix Simian Army (Chaos Monkey)

Evolutionary Design

- More granular release planning
- Goal: change tolerance
  - Identify change impacts quickly (via automation, service contracts, automated dependency maps)
  - Fix rather than revert
  - In case of mandatory breaking change, use versioning (a last resort option in the microservices world)

How Big Should my Microservice Be?

- Microservices ownership implies that each team is responsible for the entire lifecycle
  - functional vs divisional organizations
  - product management, development, QA, documentation, support
- Sizing depends on the system you're building
  - Amazon 2PT principle - Two Pizza Teams
  - 6 to 10 people to build/deploy/maintain a microservice
  - an average microservice consists of 2000 lines of code, 5 source files, and is run on 3 instances in production

RESTish Microservices?

- REST
  - Web lingua franca, 100% interoperable
  - Development cost is generally higher
  - Best practices: provide client SDKs, (e.g. generated from Swagger/RAML or other API description languages)
  - Performance issues if not well-designed (chattiness)
  - Best practices: experience based and coarser grained APIs
- RPC
  - Optimized communication via binary formats
  - Automated generation from IDL, polyglot by default
  - Integrated support multiples scenarios : request/response, streaming, bi-directional streaming

RESTish Microservices?

- RPC vs REST style depends on the system you’re building and teams existing skills set
- Whatever the style, your microservices architecture **must** provide
  - Services Discovery,
  - Reliable Communications,
  - Operational insights (logs, monitoring, alerts, real time analysis)

Got It, But Isn't It SOA?

- SOA so what ?
  - Enterprise SOA
  - Event-driven architecture (Pub/Sub)
  - Streaming Services (real-time time series, bidirectional)
  - Container-Services (ala Docker)
  - Nanoservices (ala AWS Lambda)
- Simply stated : Microservices are a SOA style for systems whose first goal is to scale
- **<u>In details, let's see how microservices differ from…</u>**

Microservices vs Enterprise SOA

- Enterprise SOA is often seen as
  - multi-year initiatives, costs millions
  - complex protocols with productivity and interoperability challenges
  - central governance model that inhibits change
- Enterprise SOA is more about integrating siloed monoliths
  - Generally, via a smart and centralized service bus
- Microservices is scalable SOA
  - an architectural style to design, develop and deploy a large and complex system, so that it is easier to scale and evolve

Microservices vs Event Driven Architecture

- EDA fits well with Document oriented systems and information flows
- Communication between microservices can be a mix of RPC (i.e., P2P calls) and EDA calls
- See EDA as a communication pattern for your microservices
- Can address choreography, orchestration, pipeline requirements

Microservices vs Container Services

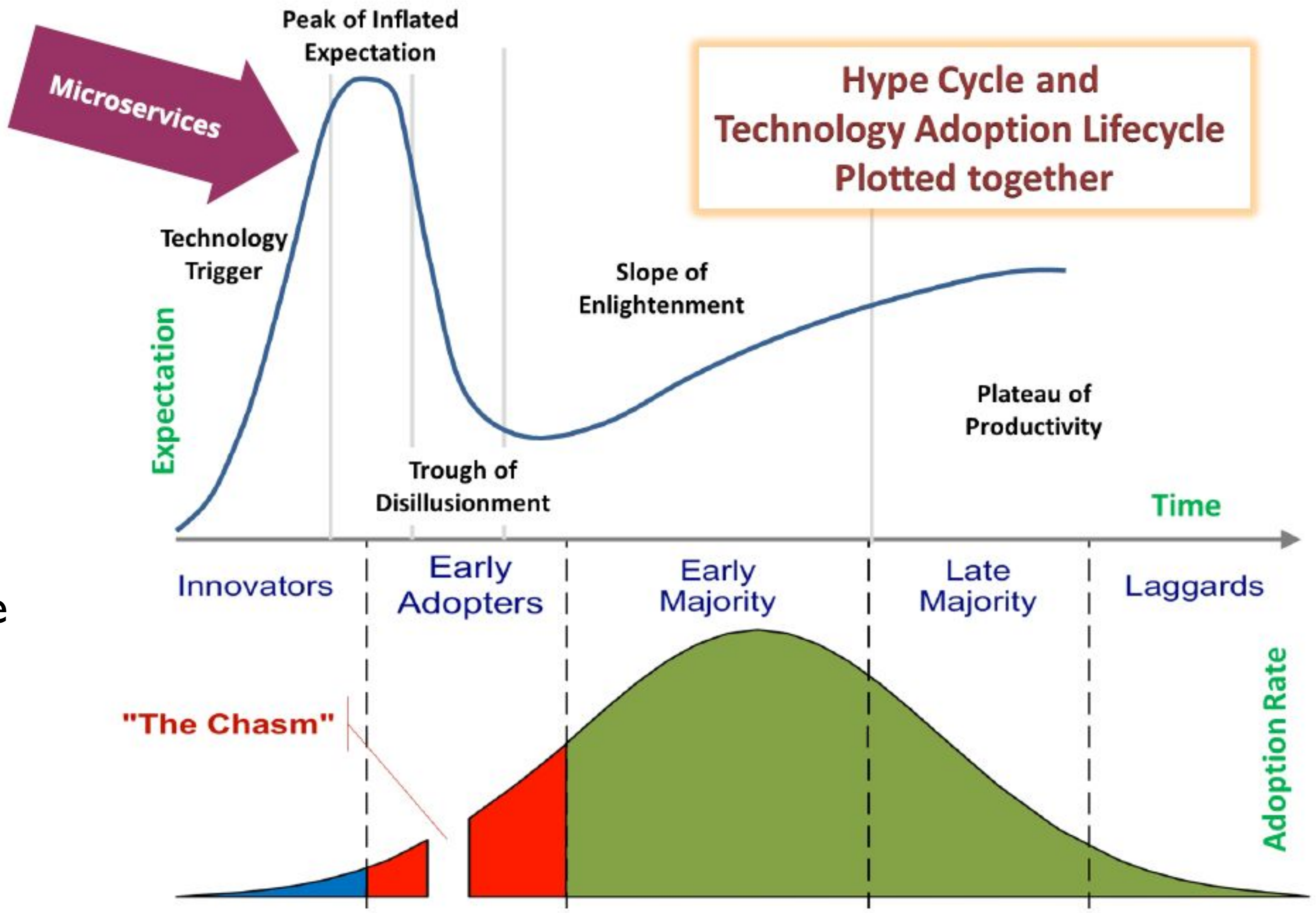
- Containers provide the infrastructure to deploy your microservices independently one from another
- See Container Services as a building block of your global microservices architecture

Microservices vs Nanoservices

- Nanoservices are small pieces of code (functions)
- E.g.: AWS Lambda, Auth0 Webtasks
- A microservice may leverage 1+ nanoservices

Gartner Hype Cycle

- Gartner Hype Cycles provide a graphic representation of the maturity and adoption of technologies and applications, and how they are potentially relevant to solving real business problems and exploiting new opportunities.
- Gartner Hype Cycle methodology gives you a view of how a technology or application will evolve over time, providing a sound source of insight to manage its deployment within the context of your specific business goals.



Code.Hub

Thank you!

Code.Hub