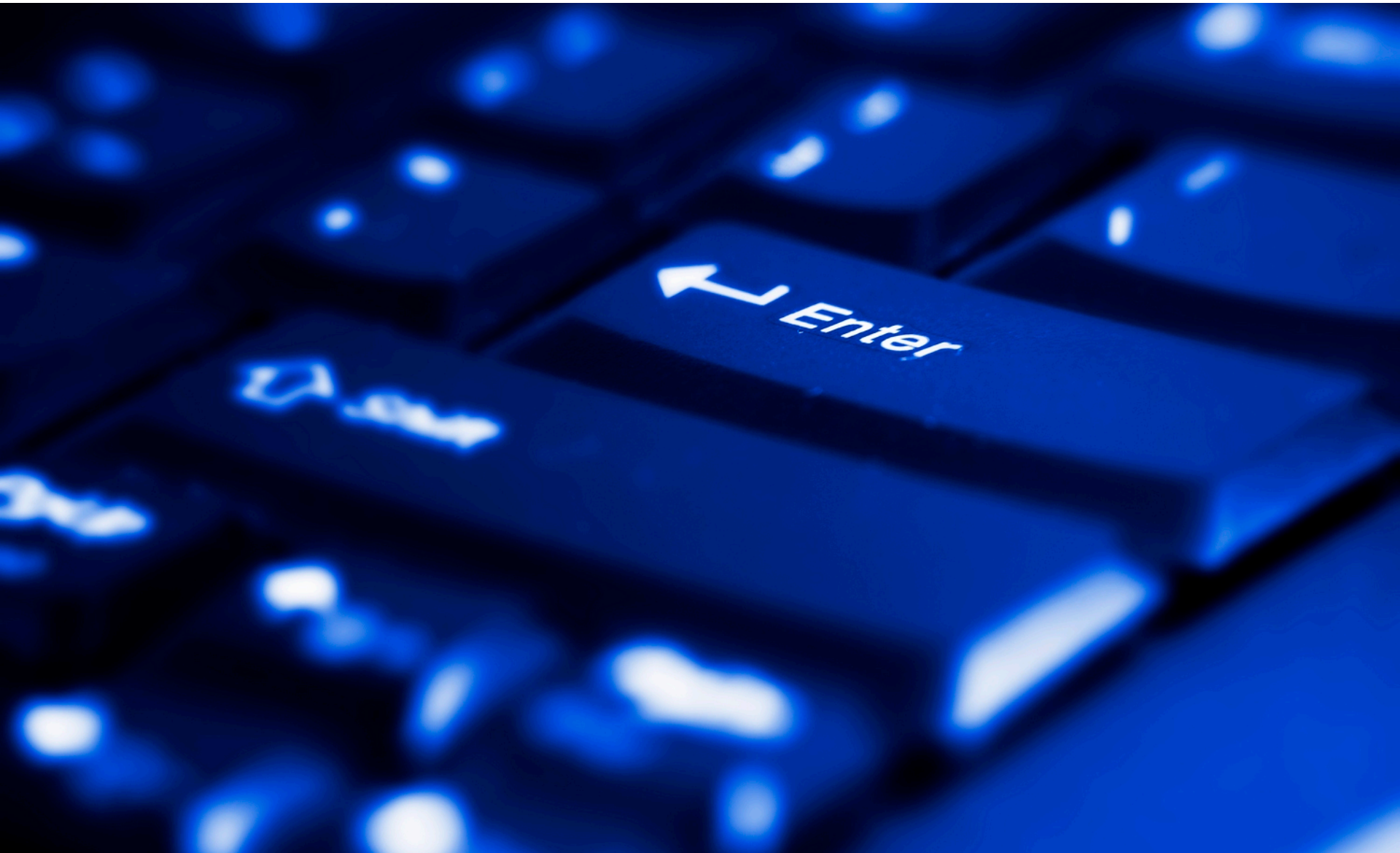
bns it

Introduction to Microservices

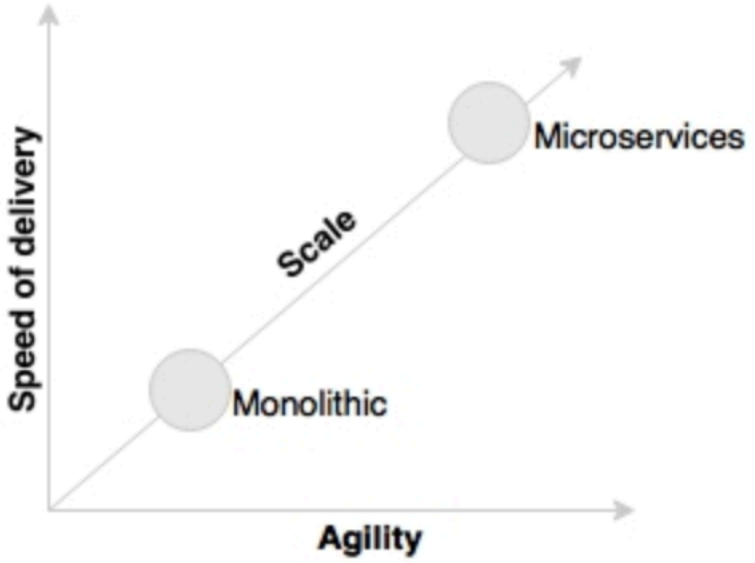
Modern Application Architectures



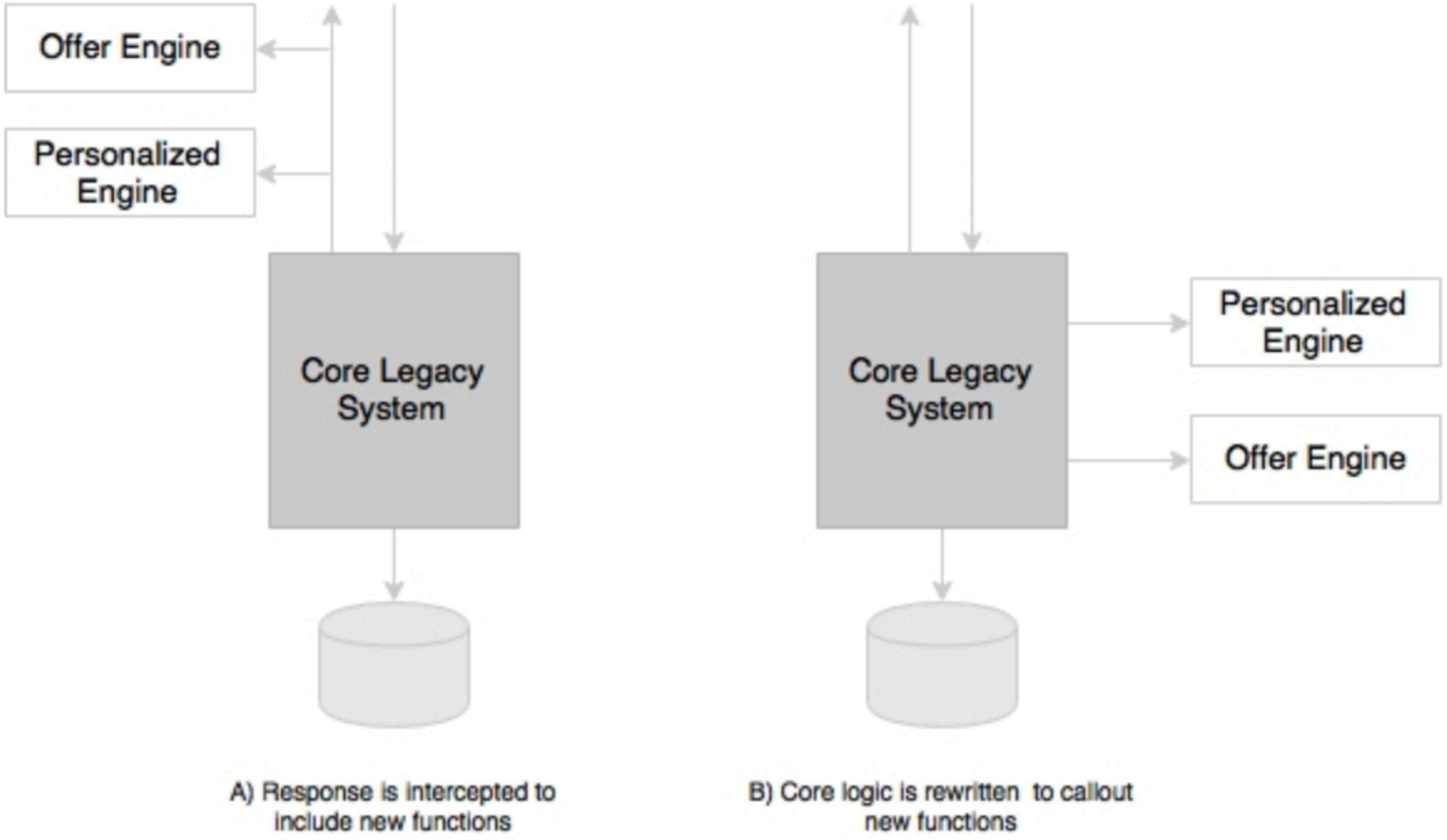
Definition

- Microservices is an architecture style and an approach for software development

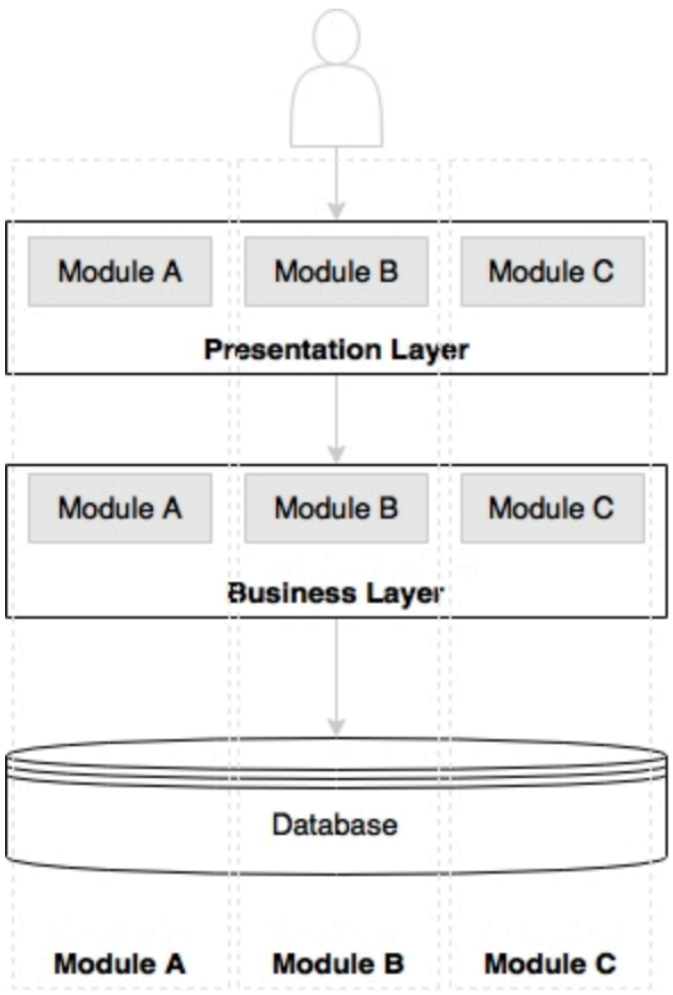
- Microservices promise
  - agility
  - speed of delivery
  - scale compared

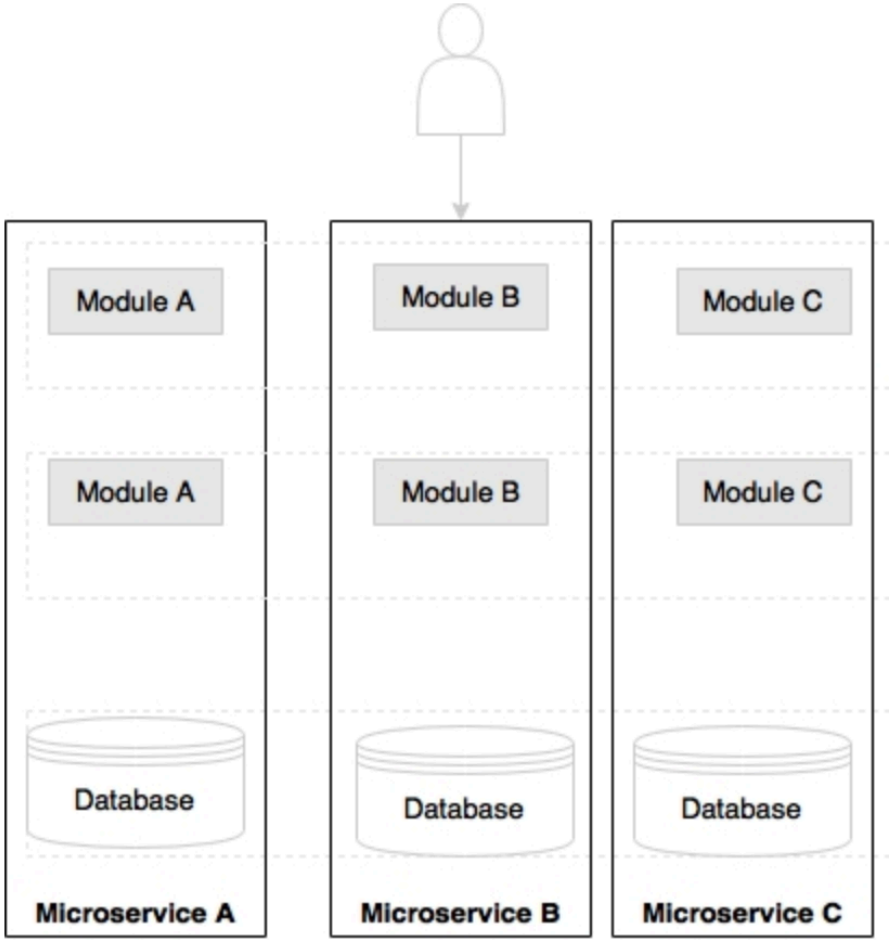


Microservices as a natural evolution

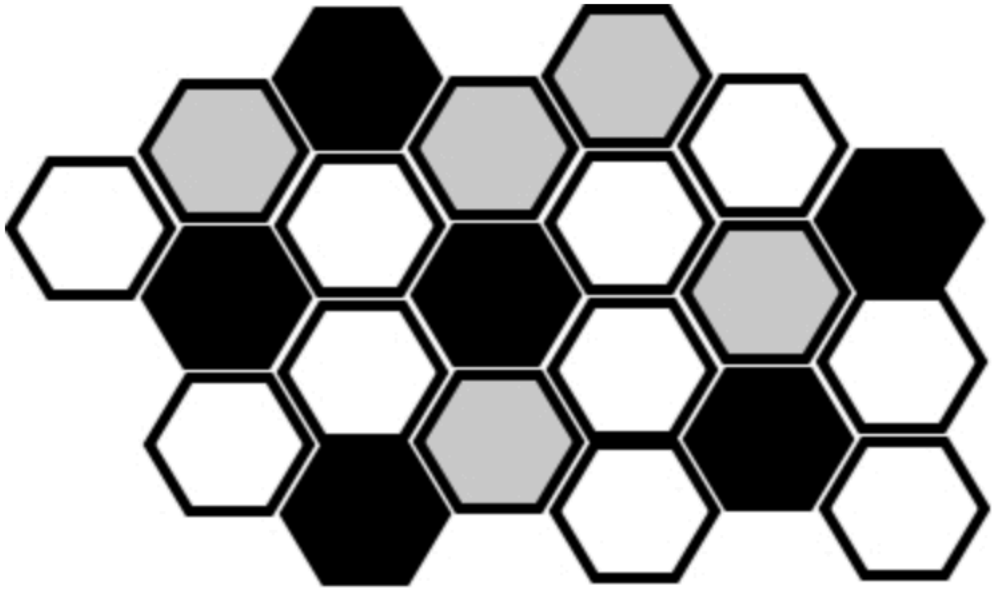


Classical vs Microservices Architecture (MSA)





Honeycomb analogy

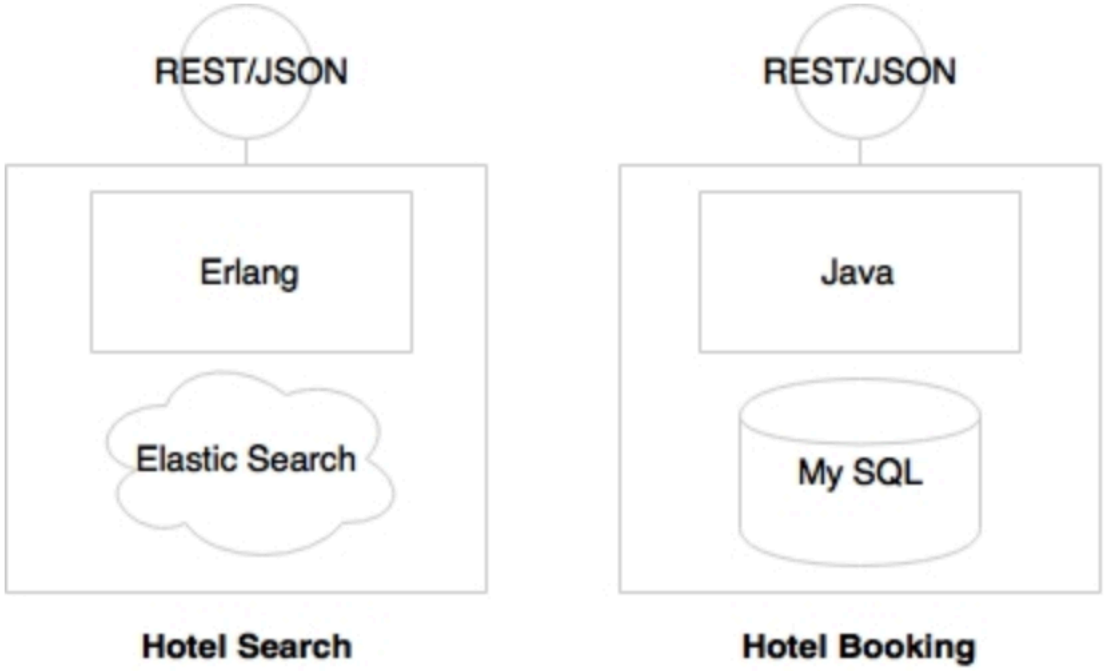


- They start small
- Construction based on what is available.
- Each cell independent but also integrated with other cells

Characteristics of MSA

- Single responsibility based on business capabilities
- Self-contained, independently deployable and autonomous
- Bundle all dependencies
- Lightweight
  - deployed in lightweight containers (Jetty, Docker)
- Enables polyglot architecture (eg. CQRS)

Different architectures (CQRS)



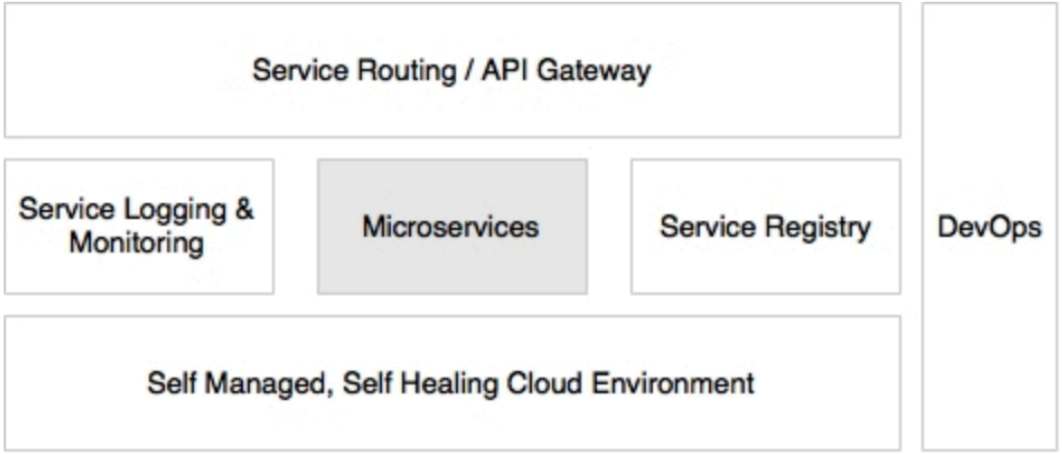
Read

Write

Characteristics of MSA

Strong automation required

Complexity in an ecosystem



Characteristics of MSA

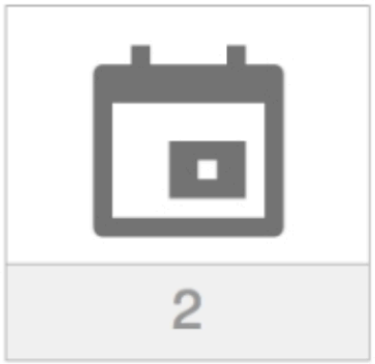
- Distributed systems
- Fail-fast and self-healing
  - focus on **Mean Time To Recover (MTTR)** not on Mean Time Between Failures (MTBF)

Examples

Travel System

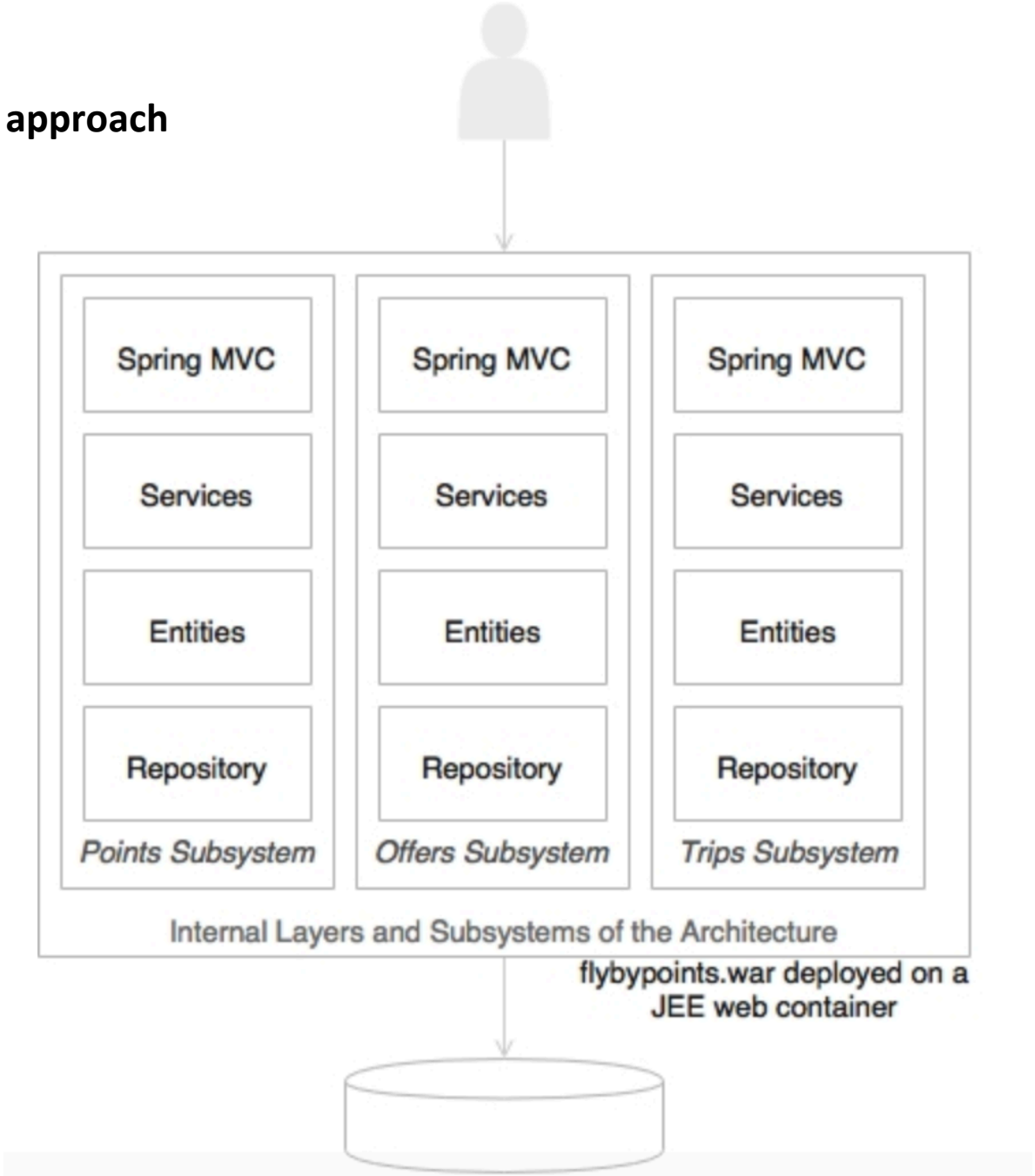
Points

Offers

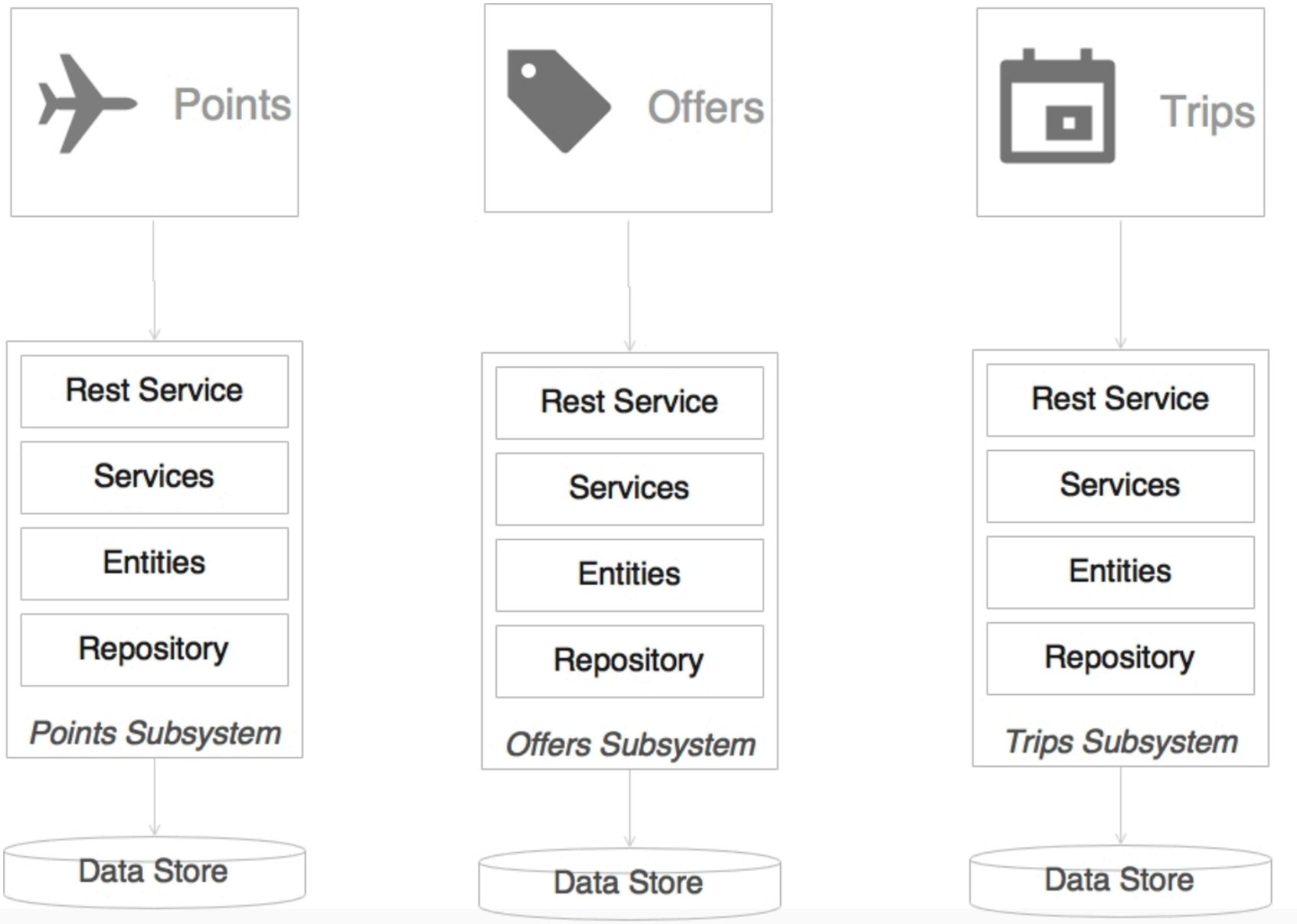


Trips

Classical approach

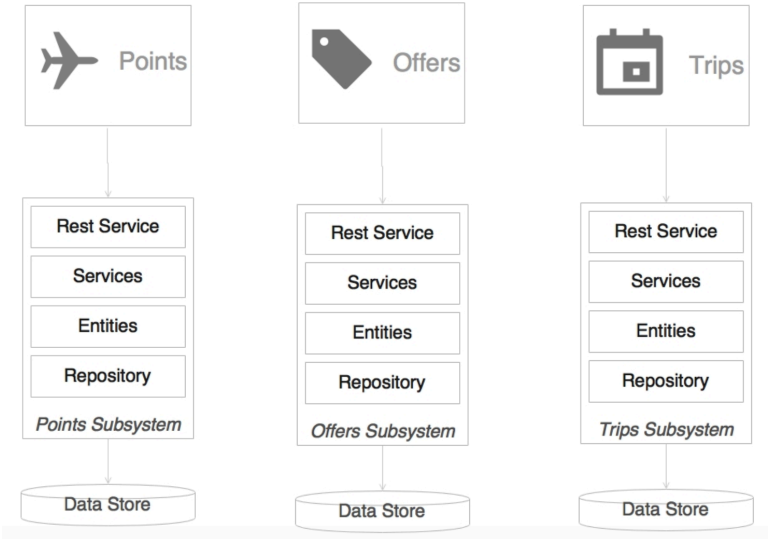


MSA approach



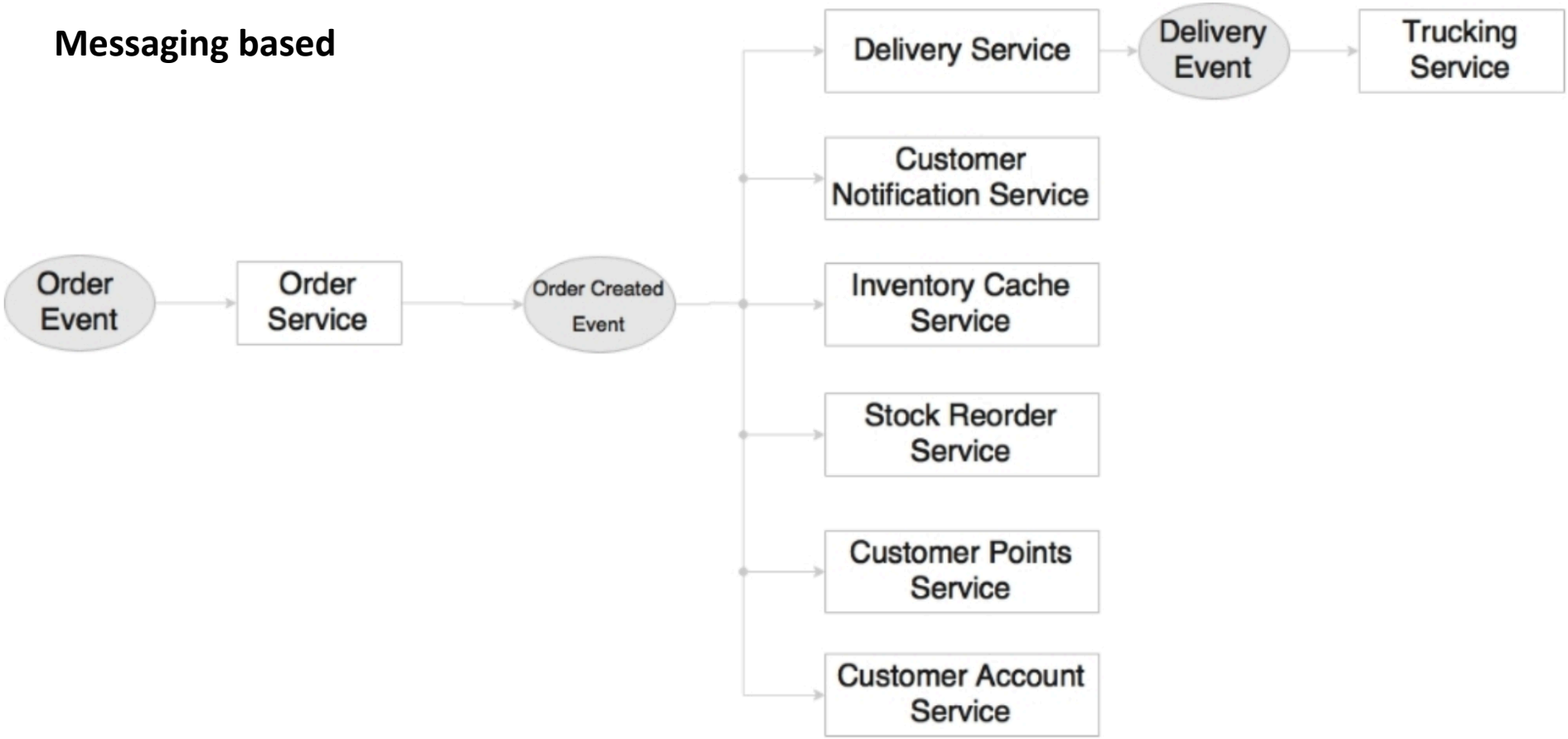
MSA approach

- Each subsystem is an independent system
- Three microservices (business functions)
  - **Trips**, **Offers** and **Points**
- Each has its own data storage
- Exposing API through REST

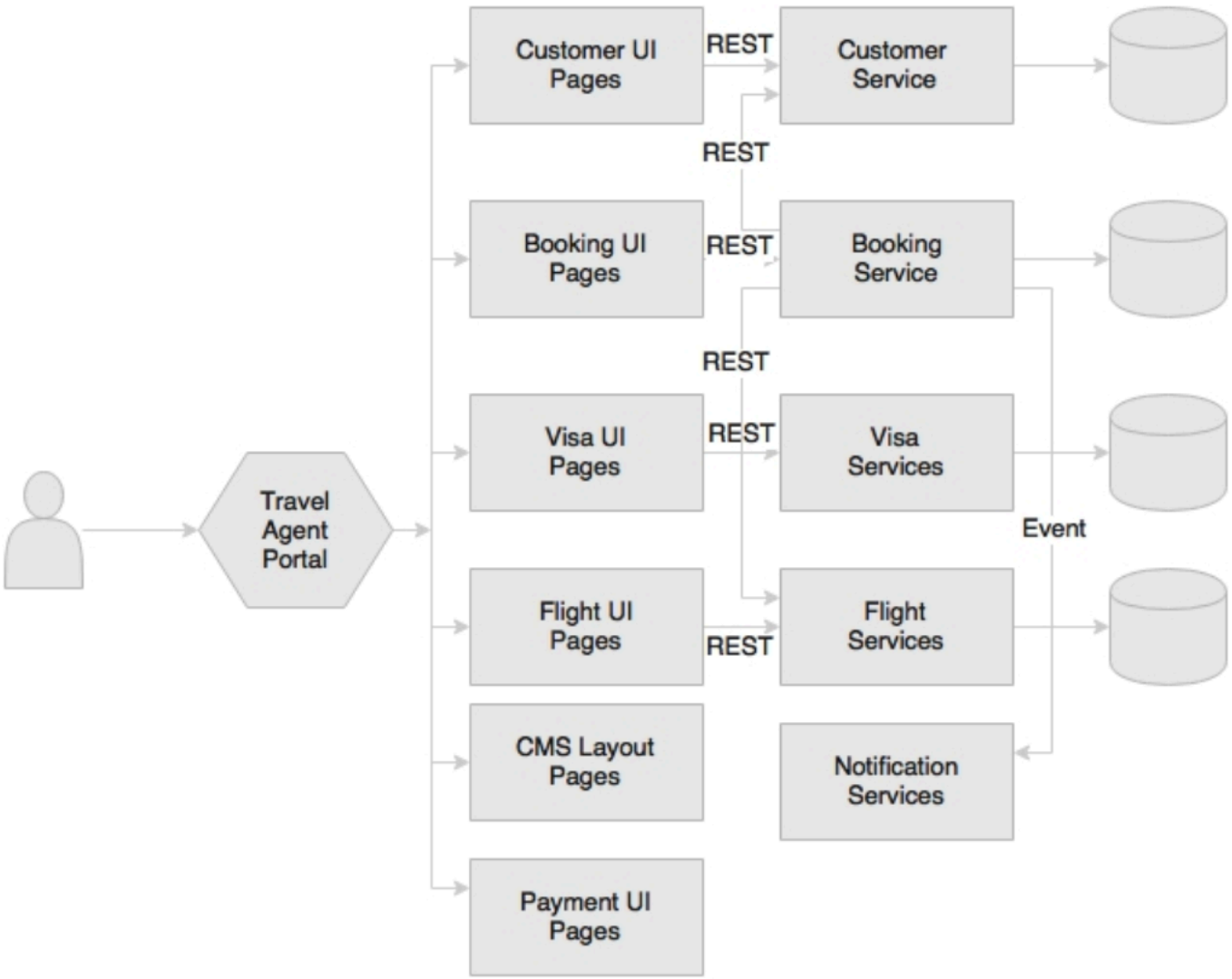


Online Retail Site

Messaging based



Travel Agent PORTAL



bns it

MSA and SOA

Relationship with SOA

Open Group Definition:

- **Service-Oriented Architecture** (SOA) is an architectural style that supports service orientation.
- Service orientation is a way of thinking in terms of services and service-based development and the outcomes of services.

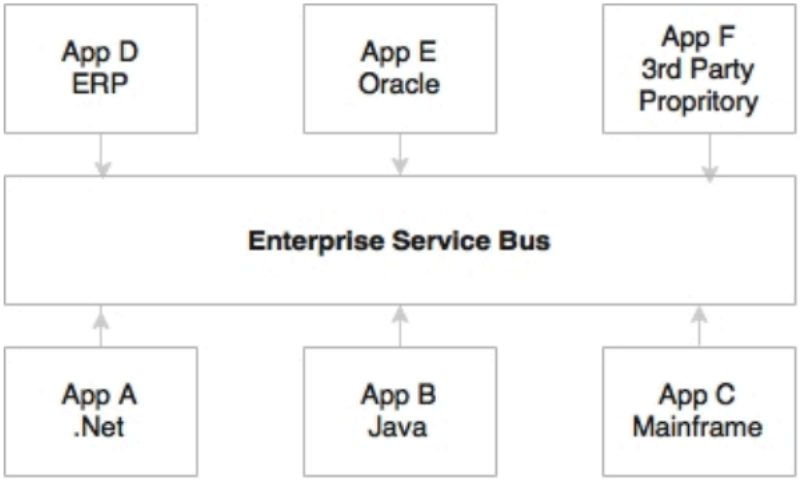
Relationship with SOA

SOA Service

- A logical representation of a repeatable business activity that has a specified outcome (e.g., check customer credit, provide weather data, consolidate drilling reports)
  - It is self-contained.
  - It may be composed of other services.
  - It is a "black box" to consumers of the service.

Classical SOA implementation

- Strongly relies on heavyweight ESB
- Packed with a set of related products such as:
  - rules engines
  - business process management engines etc.
- Heavy orchestration logic in the ESB layer
- Business logic in ESB



Twelve-factor Apps

Twelve-factor Apps

- Described by Heroku
- Specifies the characteristics expected from modern cloud-ready applications
- Applicable to microservices

Twelve-factor Apps

Single code base

- one codebase for dev, tests and production
- each Microcervice has its codebase

Bundle dependencies

- bundle all dependencies with the application

Externalize configuration

- externalize all configuration parameters from the code, often centralized

Twelve-factor Apps

Addressable services

- all backing services should be accessible through an addressable URL
- mostly HTTP endpoints

Isolation between build, release and run

- **build** == compile and produce binaries including all the assets
- **release** == combine binaries with enviroment specific configuration parameters
- **run** == run application on specific environment
- pipeline is unidirectional: build -> release -> run

Twelve-factor Apps

Stateless and shared nothing processes

- services are stateless in terms of conversional state

Exposing services through port bindings

- services musn’t rely on an external servers
- self-contained

Twelve-factor Apps

Concurrency to scale out

- promote functional programming for easy scaling

Disposability with minimal overhead

- keep minimal startup and shutdown time
- consider lazy loading

Twelve-factor Apps

Development and production parity

- keep development and production environments identical
- embrace lightweight containers and cloud

Externalize logs

- use centralized logging framework (Splunk, Logstash etc.)
- enables reasoning in Microservices environment

Package admin processes

- Admin code should also be packaged with the application code

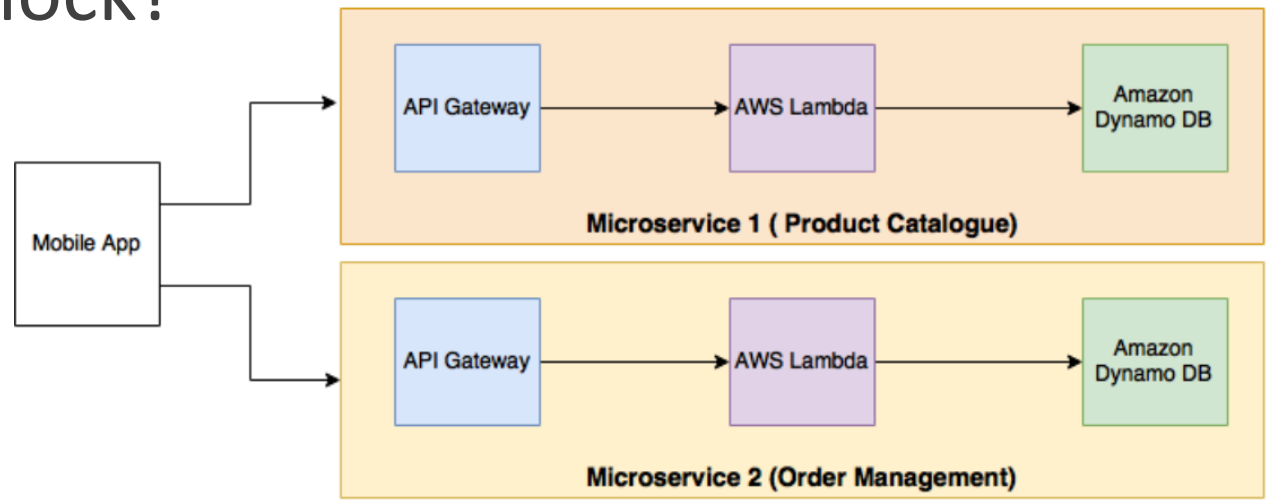
Other trends

Serverless

- aka **Functions as a Service (FaaS)**
- aka **NoOps**
- No need to worry about:
  - application servers
  - virtual machines
  - containers
  - infrastructure
  - scalability
  - other quality of services
- AWS Lambda, IBM OpenWhisk, Azure Functions, Google Cloud Functions

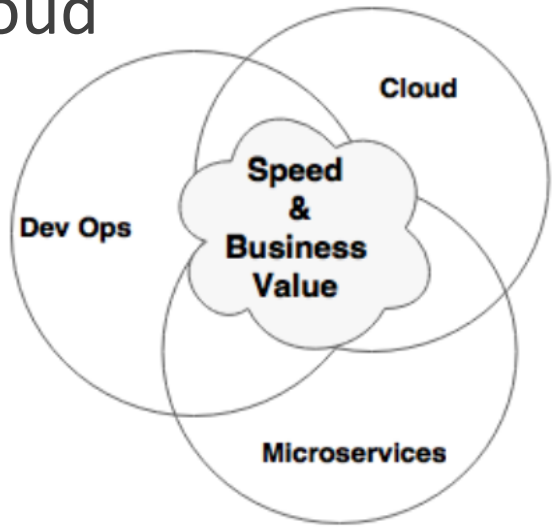
Serverless characteristics

- Functions generally perform one task
- Isolated by nature
- Communicate through either event-based or HTTP API
- Vendor lock?



DevOps, Cloud and Containers

- **Cloud** (more specifically, Containers), **Microservices** and **DevOps** is a current trend
- Complement each other
- Often a path for organization
  - DevOps -> Microservices -> Cloud



Microservices Early Adopters

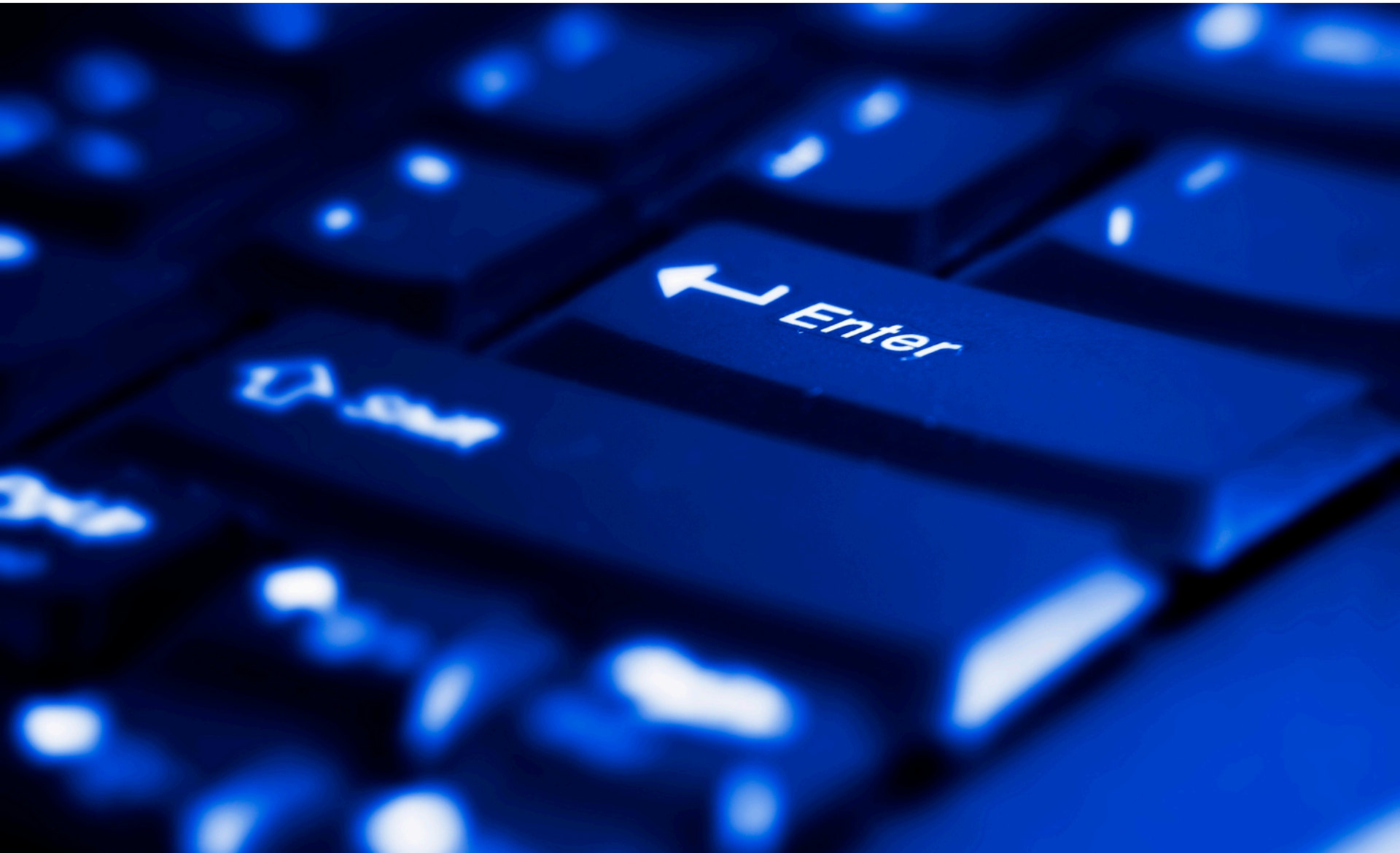
Microservices Heroes

- Netflix (www.netflix.com)
- Uber (www.uber.com)
- Airbnb (www.airbnb.com)
- Orbitz (www.orbitz.com)
- eBay (www.ebay.com)
- Amazon (www.amazon.com)
- Twitter (www.twitter.com)

bns it

Microservices design

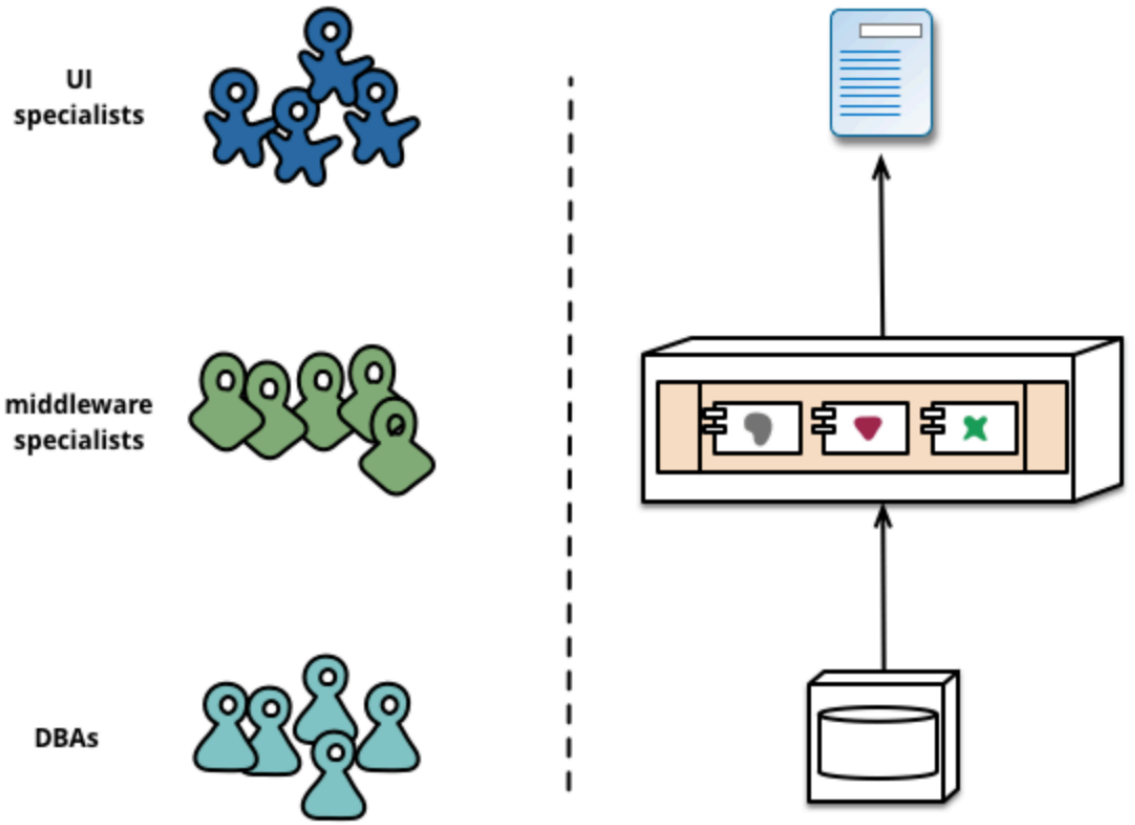
Building microservices using Spring Boot



Conway Law

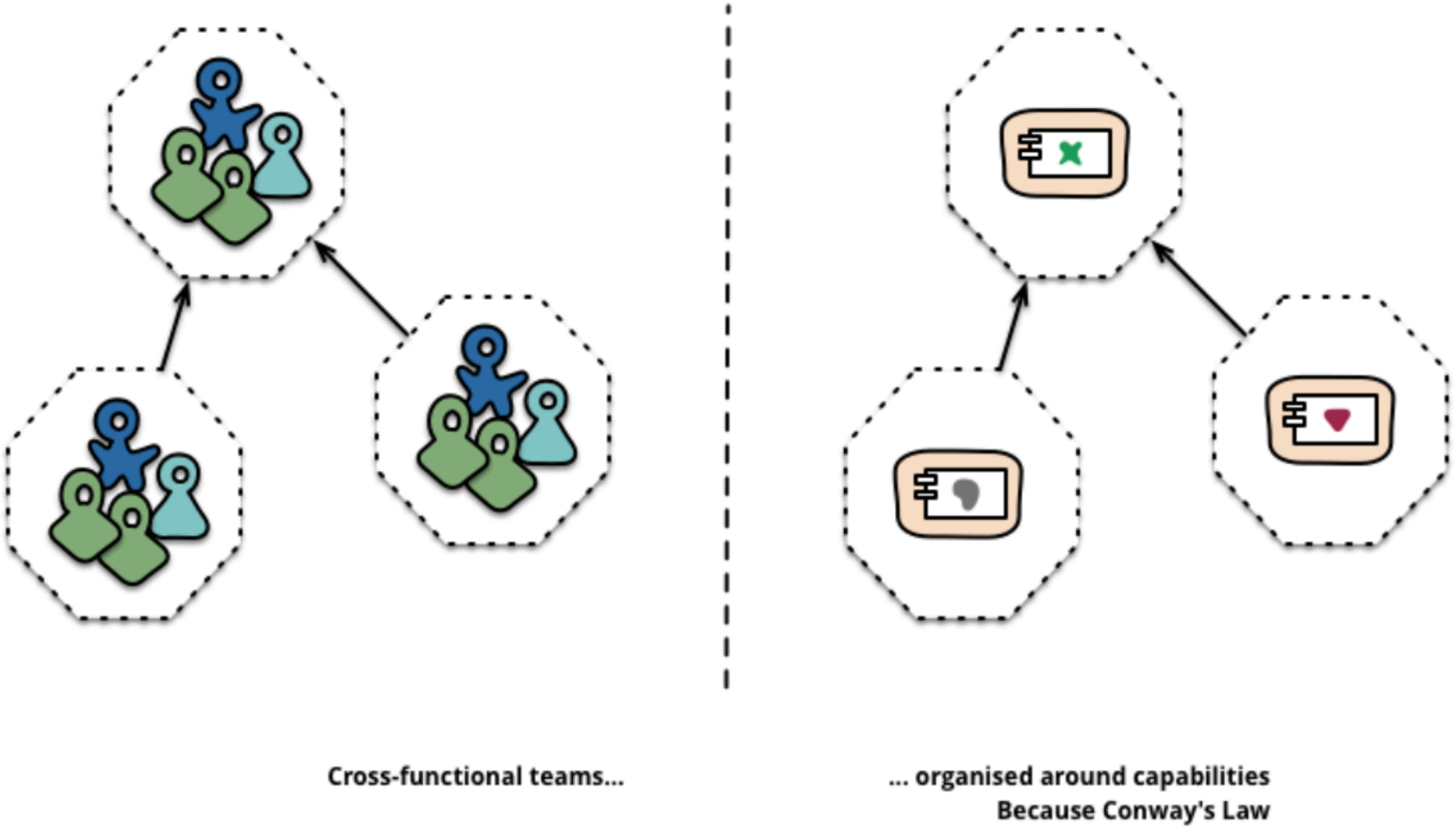
Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

-- Melvyn Conway, 1967



https://martinfowler.com/microservices/

Conway Law



https://martinfowler.com/microservices/

But...

Microservices provide benefits...

- Strong Module Boundaries: Microservices reinforce modular structure, which is particularly important for larger teams.

- Independent Deployment: Simple services are easier to deploy, and since they are autonomous, are less likely to cause system failures when they go wrong.

- Technology Diversity: With microservices you can mix multiple languages, development frameworks and data-storage technologies.

...but come with costs

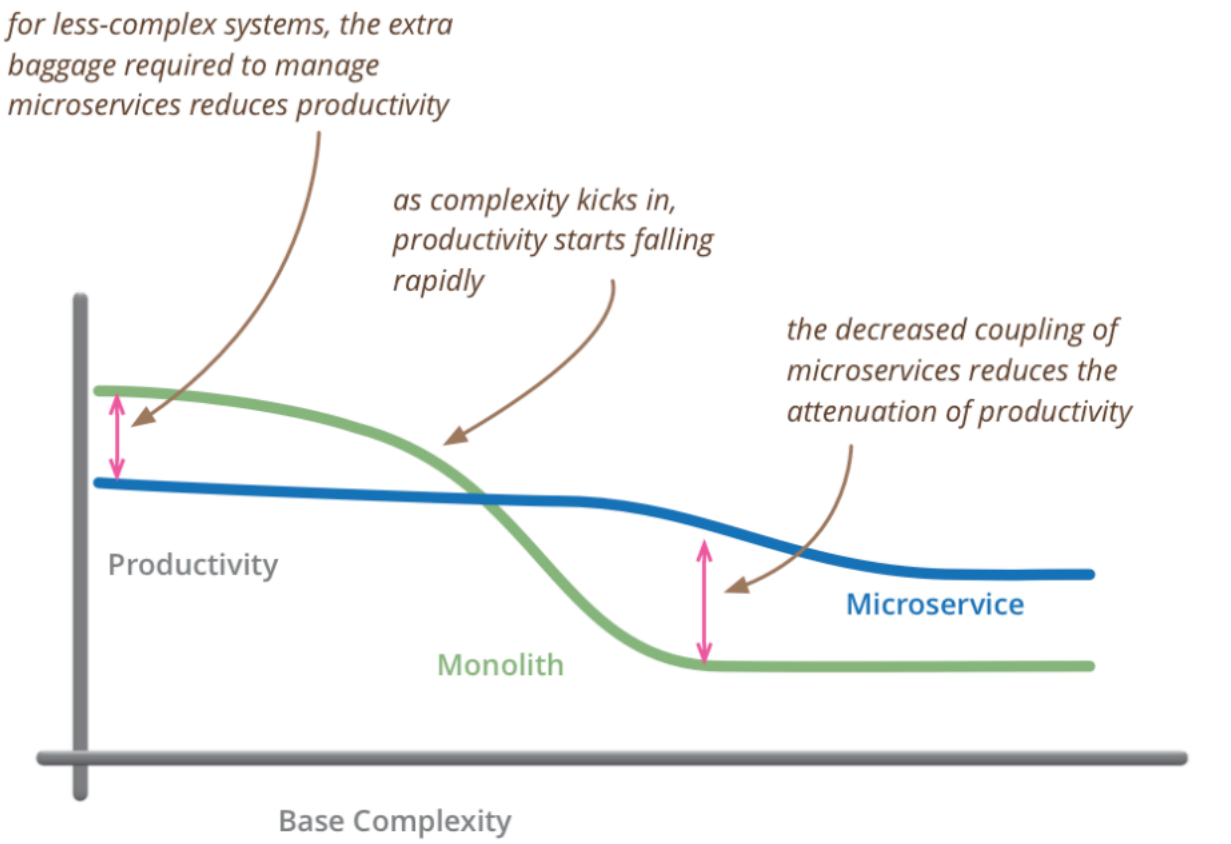
- Distribution: Distributed systems are harder to program, since remote calls are slow and are always at risk of failure.

- Eventual Consistency: Maintaining strong consistency is extremely difficult for a distributed system, which means everyone has to manage eventual consistency.

- Operational Complexity: You need a mature operations team to manage lots of services, which are being redeployed regularly.

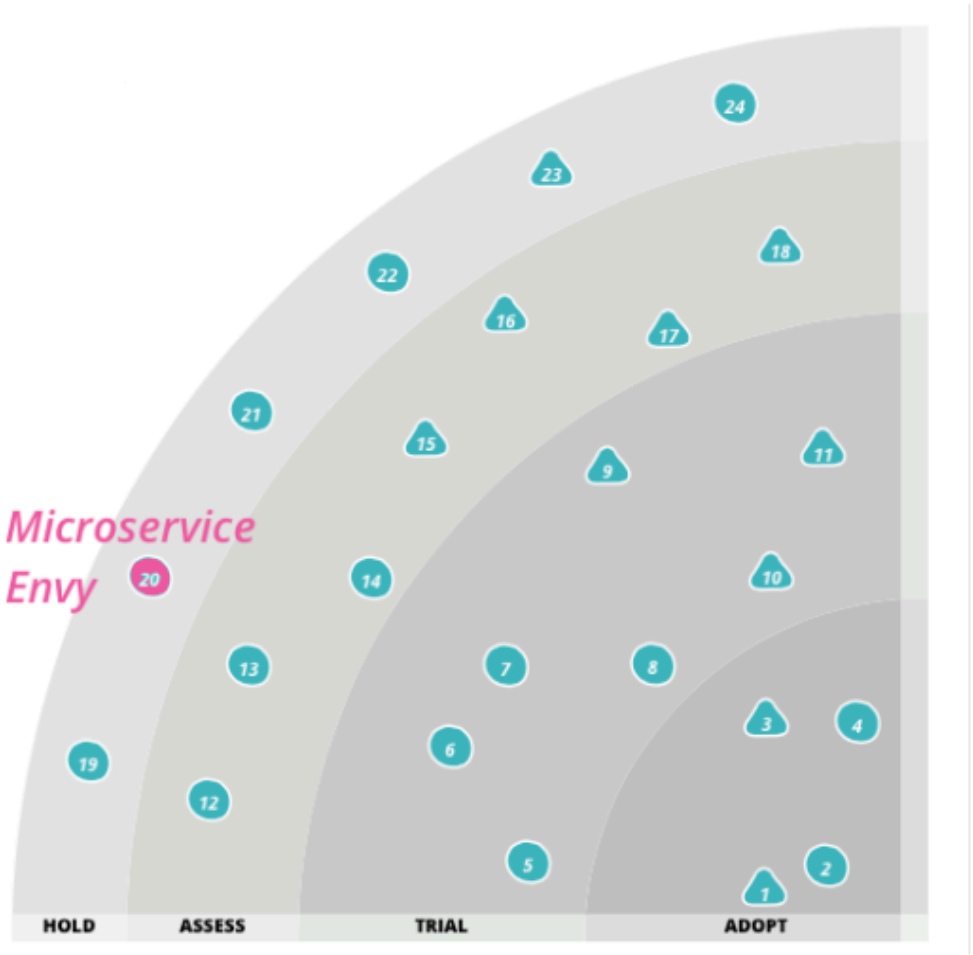
https://martinfowler.com/microservices/

Costs vs benefits



but remember the skill of the team will outweigh any monolith/microservice choice

Thoughtworks technology radar ... warning



Continuous Delivery Maturity Model

Practice	Build management and continuous integration	Environments and deployment	Release management and compliance	Testing	Data management
Level 3 - Optimizing: Focus on process improvement	Teams regularly meet to discuss integration problems and resolve them with automation, faster feedback, and better visibility.	All environments managed effectively. Provisioning fully automated. Virtualization used if applicable.	Operations and delivery teams regularly collaborate to manage risks and reduce cycle time.	Production rollbacks rare. Defects found and fixed immediately.	Release to release feedback loop of database performance and deployment process.
Level 2 - Quantitatively managed: Process measured and controlled	Build metrics gathered, made visible, and acted on. Builds are not left broken.	Orchestrated deployments managed. Release and rollback processes tested.	Environment and application health monitored and proactively managed. Cycle time monitored.	Quality metrics and trends tracked. Non functional requirements defined and measured.	Database upgrades and rollbacks tested with every deployment. Database performance monitored and optimized.
Level 1 - Consistent: Automated processes applied across whole application lifecycle	Automated build and test cycle every time a change is committed. Dependencies managed. Re-use of scripts and tools.	Fully automated, self-service push-button process for deploying software. Same process to deploy to every environment.	Change management and approvals processes defined and enforced. Regulatory and compliance conditions met.	Automated unit and acceptance tests, the latter written with testers. Testing part of development process.	Database changes performed automatically as part of deployment process.
Level 0 – Repeatable: Process documented and partly automated	Regular automated build and testing. Any build can be re-created from source control using automated process.	Automated deployment to some environments. Creation of new environments is cheap. All configuration externalized / versioned	Painful and infrequent, but reliable, releases. Limited traceability from requirements to release.	Automated tests written as part of story development.	Changes to databases done with automated scripts versioned with application.
Level -1 – Regressive: processes unrepeatable, poorly controlled, and reactive	Manual processes for building software. No management of artifacts and reports.	Manual process for deploying software. Environment-specific binaries. Environments provisioned manually.	Infrequent and unreliable releases.	Manual testing after development.	Data migrations unversioned and performed manually.