Service Architectural Patterns

Janal Kostičová

Outline

- Software architecture
- Major architectural patterns
  - Monolith
  - Service Oriented Architecture
    - SOA with ESB
    - Microservices
  - Event Driven Architecture & Event Sourcing
- Monolith to Microservices migration
- Current trends

Software architecture

= the organization or structure of a system, where the system represents a collection of components that accomplish a specific function or set of functions.

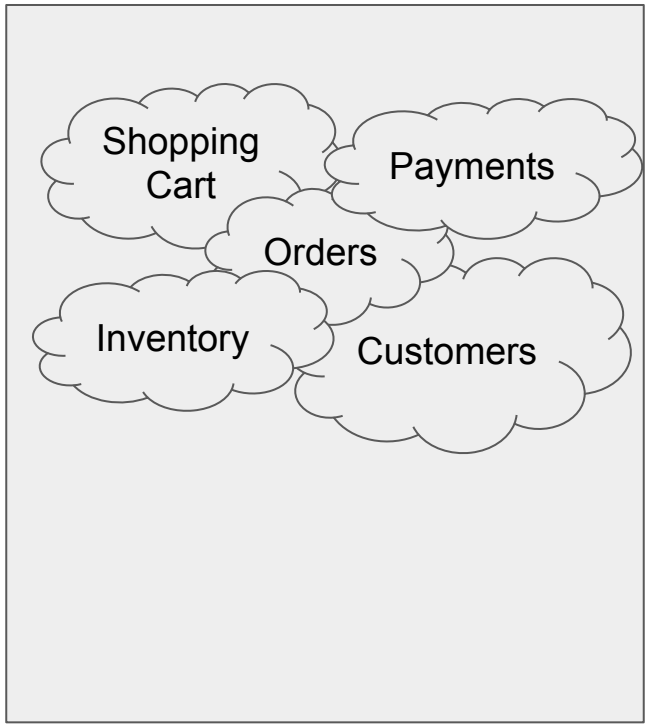
- A (software) component is a modular, cohesive unit of a software system that encapsulates related functionality
- Components serve as the building blocks for the structure of a system
- Components are connected via well-defined interfaces
- Components are typically specified in different views to show the relevant functional and non-functional properties of a software system

Example recap

- E-commerce site
  - Customer buys goods
  - Company employees manage the inventory and orders

Major Architectural Patterns and Styles

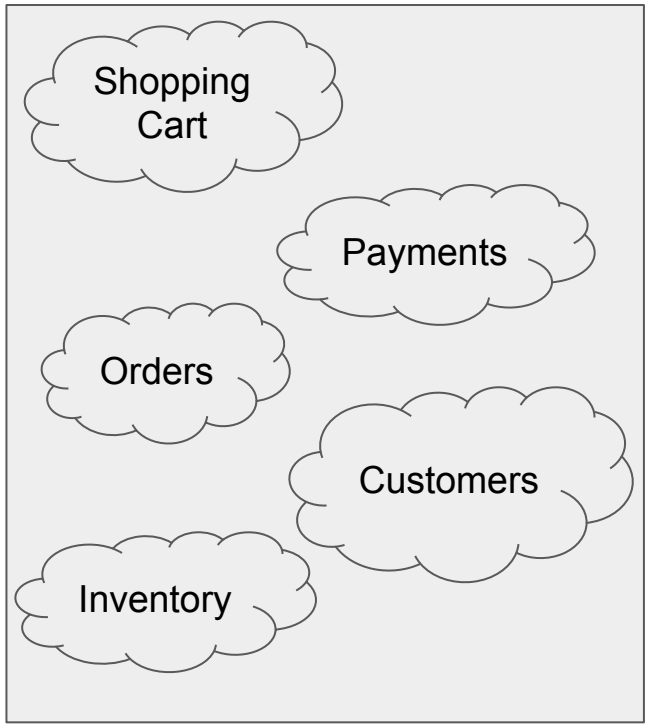
Monolith



Monolith

- Single unified software application, that is self-contained
- Single process
- Pros
  - Simplicity - everything in a single codebase
  - Efficiency - fast communication between sub-modules
  - Ease of development - running locally, debugging, …
- Cons
  - Maintainability
  - Scalability
  - Agility - adding new features can be complex
  - Single point of failure

Modular monolith



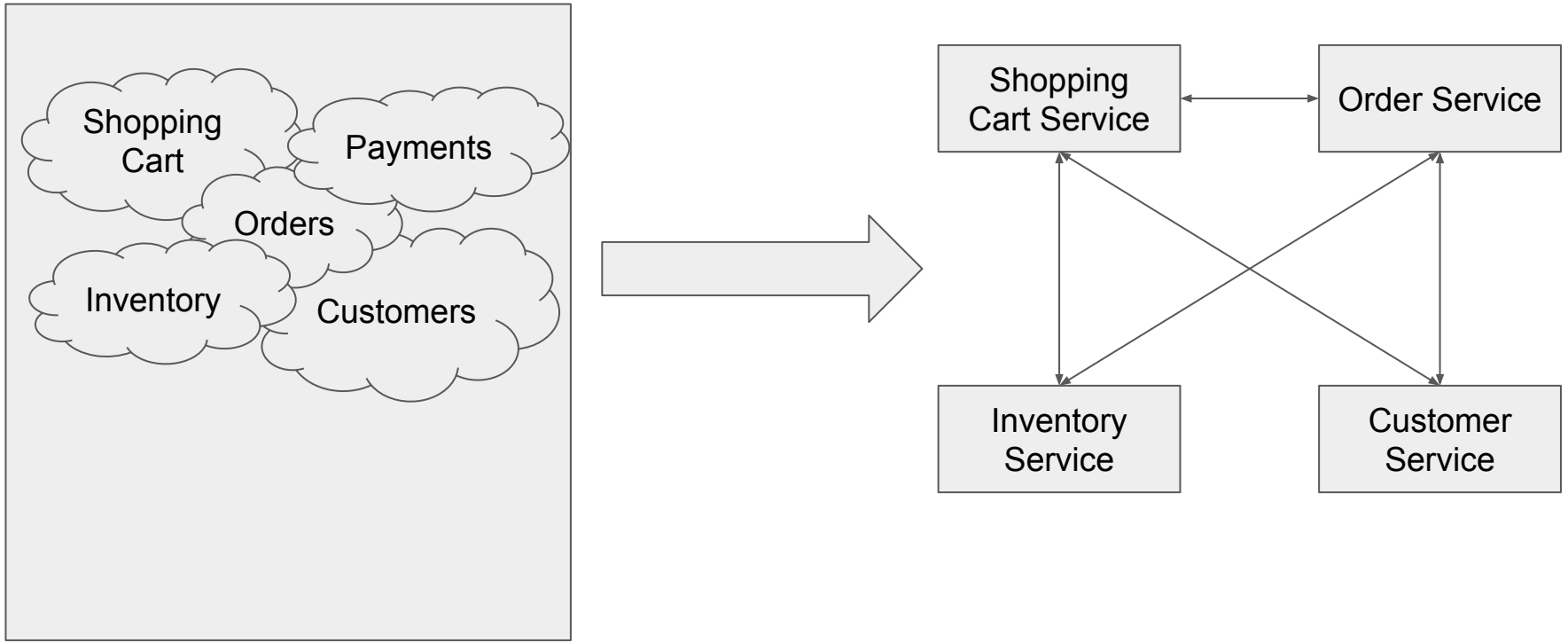
= monolith + good internal modular design

- Has all the defining monolithic characteristics
- In addition: code is organized into independent modules with well defined interfaces

➢ Aligns with component-based architecture

➢ Mitigates some drawbacks of traditional tightly coupled monoliths

Service Oriented Architecture



What is a service?

- Many definitions exist

In practice, a service is understood as one of the possible implementations of a software component:

- General characteristics of a software component hold
  - Encapsulates related business functionality
  - Provide well-defined interface
- In addition: a service mostly
  - runs as a separate process
  - makes its API accessible via network
  - is independently deployable

> Definition (Martin Fowler, [2]):
>
> Services are out-of-process components who communicate with a mechanism such as a web service request, or remote procedure call.

Often, services are designed to be discoverable and composable.

Service Oriented Architecture

= a method of software development that uses software components called services to create business applications

(.. again many definitions exist)

“SOA is this very broad term and microservices is a subset of its usage” (Martin Fowler [3])

- Pros
  - Reusability
  - Scalability
  - Agility
  - Loose Coupling
  - Platform/Technology Independence
- Cons
  - Complexity
  - Network overhead
  - Ease of development - distributed debugging, difficult to run locally

Monolith vs SOA

Logical components in **monolithic architecture**

- Implementation: Typically as libraries or internal modules
- Communication: Function/method calls within the same process

Logical components in **SOA** (distributed environment)

- Implementation: Exposed as services
- Communication: Over network protocols (REST, SOAP, gRPC, events, …)

⇨ SOA = a specialization of component-based architecture where **software components** are **services**

We will talk about two SOA variants: SOA with a bus and microservices.

History

1970s–1990s
Monolithic systems, often tightly coupled

Late 1990s–2000s
SOA (enterprises) communication via a bus / SOAP

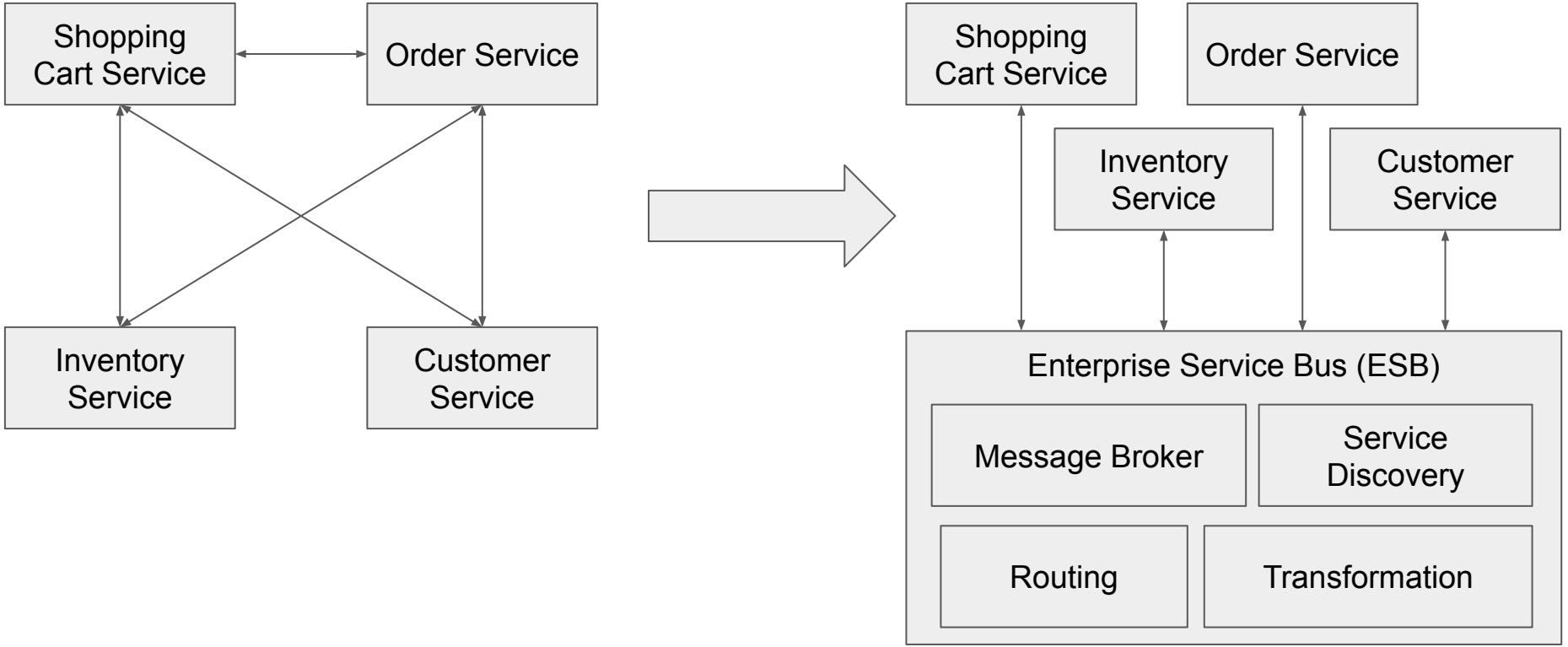
2005–2010
Early microservices-like systems (Amazon, Netflix,..)

2011–2014
Microservices widely recognized

2017
Microservices in the mainstream

2023-now
Shift to modular monoliths / hybrid architectures

SOA with Enterprise Service Bus (ESB)



SOA with Enterprise Service Bus (ESB)

- Simplify communication and service discovery in settings with high number of services
- Enterprise Service Bus
  - Message routing
  - Data and protocol transformation
  - Security enforcement
  - Monitoring and management
- Pros
  - Central integration point
  - Loose coupling
  - Standardization
- Cons
  - Increased complexity
  - Performance overhead
  - Increased cost and vendor lock-in
  - ESB is still a single point of failure

Example

SOA ESB Restaurant reservation system

https://github.com/medjb10/soa-esb-restaurant-reservation-system

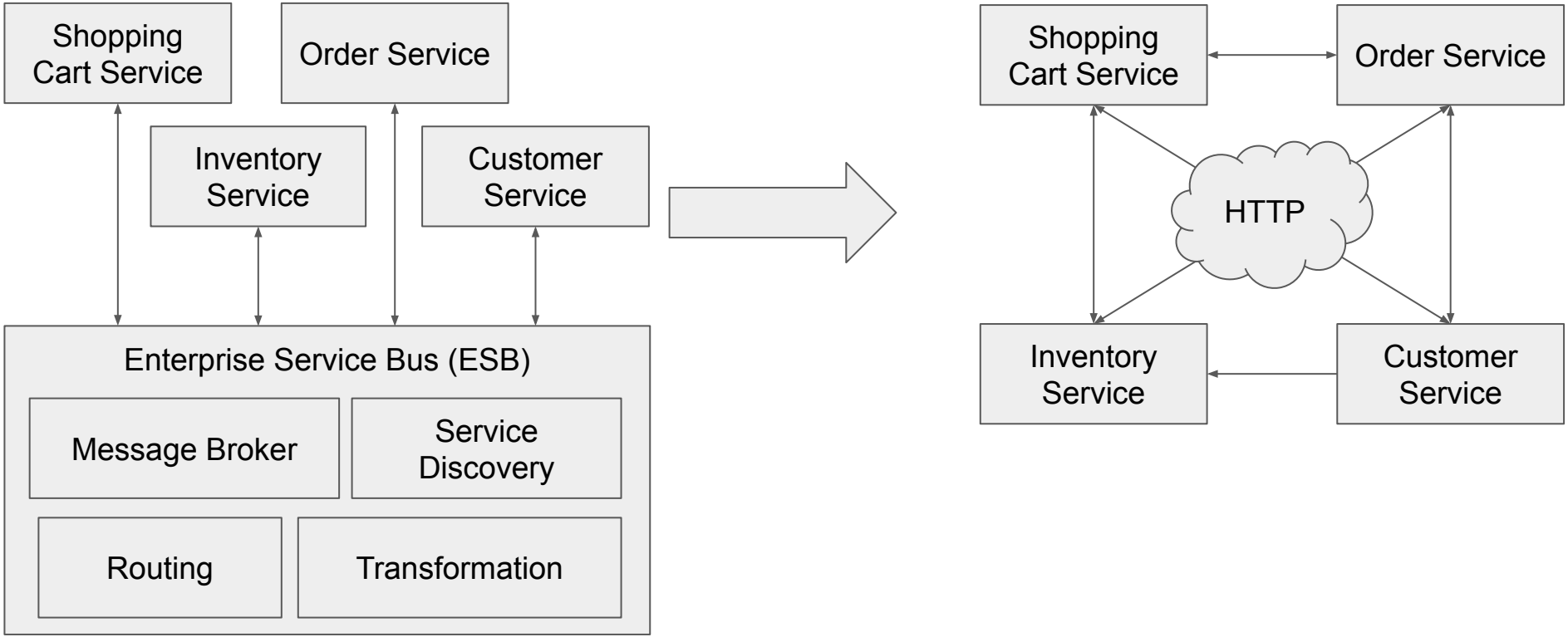
The project uses an existing ESB platform (Mule ESB Community Edition)

- Message routing, Data transformation, Protocol Bridging, Orchestration

What has to be implemented

- Services (business logic)
- Integration logic inside Mule
  - Flows (routing rules)
  - Transformations (XML/JSON mappings)
  - Orchestration (call multiple services)

Microservices



Microservices

- Microservices are an architectural approach to software development where software is composed of small independent services that communicate over well-defined APIs
- Pros
  - Scalability
  - Agility
  - Fault tolerance
  - Improved maintainability
  - Technology independence
- Cons
  - Increased complexity
  - Distributed debugging
  - Communication overhead
  - Deployment complexity

Microservices - how to split?

- Business capability - models organization
- Subdomain
- Self contained service
- Team (Conway’s law [7])

How big?

“micro” in “microservices” - misleading, it often causes over-splitting

- **Amazon:** Two-pizza team [5]
- **James Lewis:** A micro-service should fit in my head. [6]

Microservices - database

- Each service has its own database
- Services communicate only via APIs / events
  - → independence
  - A microservice should not access DB of another microservice directly (shared DB is antipattern)
- Separate vs shared infrastructure

Issue - CONSISTENCY

- Mostly eventual consistency is used
  - Failures handled explicitly via compensation (SAGA pattern)
- What about distributed transactions (2PC) ?
  - Complex, slow, fragile.. rarely used

Example: Overselling

- Scenario: product stock = 1, users A and B try to buy at the same time

Order Service + Inventory Service + separate databases + async communication (events)
(eventual consistency, saga pattern)

Step 1: Order Service
- A → OrderCreated(A)
- B → OrderCreated(B)

Step 2: Inventory Service
- OrderCreated(A) → StockReserved(A)
- OrderCreated(B) → StockFailed(B)

Step 3: Order Service
- StockReserved(A) → continue processing A
- StockFailed(B) → cancel order B (saga compensation)

API composer pattern

- Builds out a facade and unifies multiple services
- No business logic, no state change

Request: GET /order/123

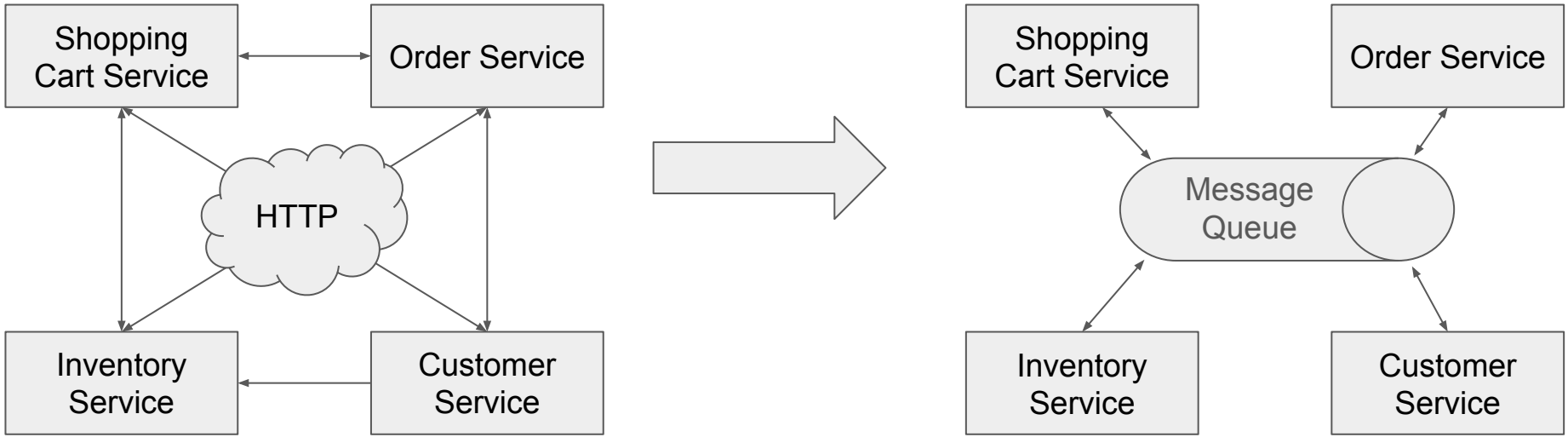
call Order Service → get order
call User Service → get customer
call Product Service → get product info
combine:

```
{
  "orderId": 123,
  "customerName": "Alice",
  "products": [
    { "name": "Laptop", "price": 1000 }
  ]
}
```

Microservices Ecosystem

- Composable Architecture
  - API first design
  - Focus on reusability
  - API gateway for publishing APIs, see this figure
- Service Mesh
  - Communication via proxy, see this figure
  - Goals - handles networking concerns for services
    - Observability
    - Discovery
    - Load balancing
    - Scalability
    - Security
    - Retries and timeouts

Event Driven Architecture



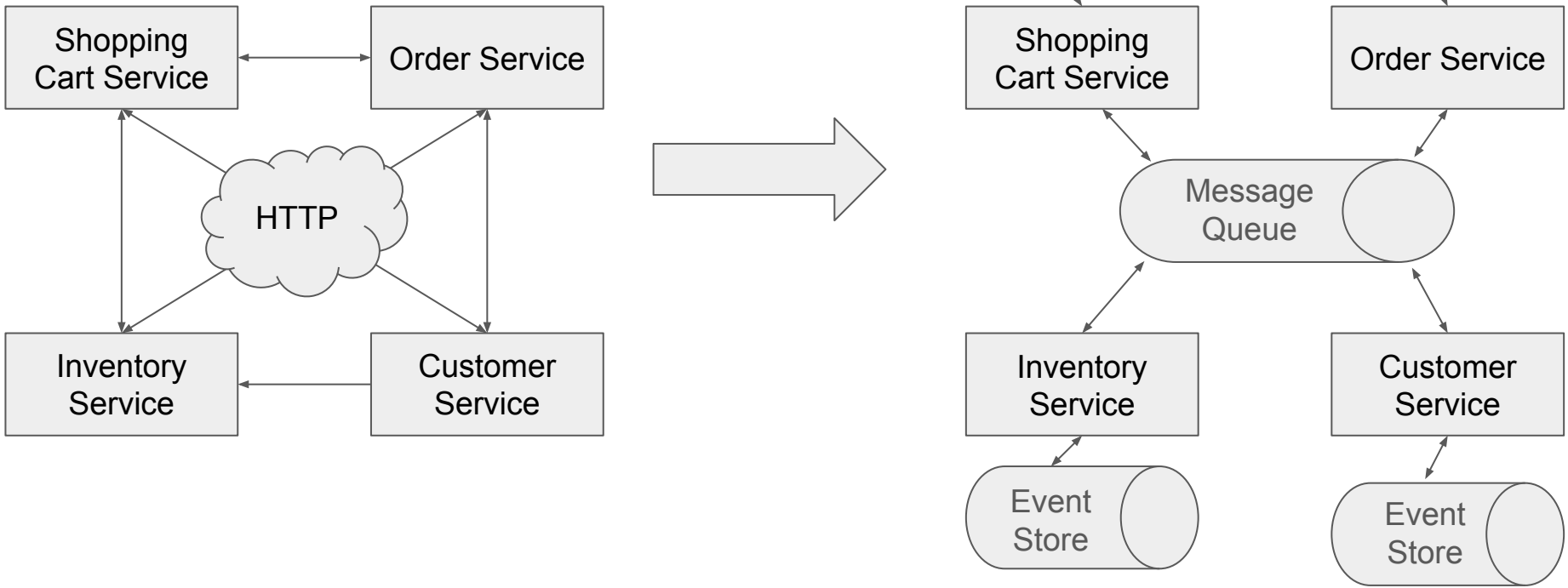
Event Driven Architecture

= an architectural pattern that focuses on service communication

- No direct API calls
- Services communicate by sending and reacting to events (asynchronous)

Message brokers (e.g., Apache Kafka)

Event Driven Architecture
+ Event Sourcing



Event Sourcing

= an architectural pattern that focuses on state storage

- All state changes are stored as a sequence of events
- The current state is derived by replaying the events instead of storing it direclty
- Events are immutable

Example: Traditional DB vs event-sourcing

```
Order {
  id: 123,
  status: "Shipped"
}
```

Status is overwritten each time

```
OrderCreated
PaymentAuthorized
PaymentCaptured
OrderShipped
```

To get current state: replay events, compute “Shipped”

Event Sourcing

- Internally, append-only event store
- Optional projections (e.g., cached views)

Pros

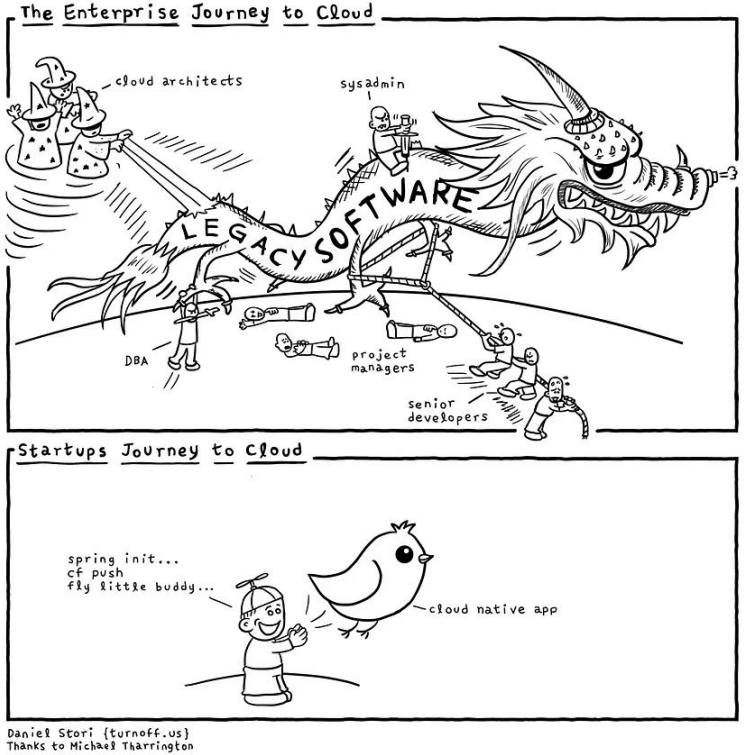
- Full audit history
- Time-travel debugging
- Better integration with microservices (-> event-driven architecture)

Cons

- More complexity (events, versioning, replaying logic)
- Harder mental model for developers
- Harder debugging

Monolith to Microservices Migration

- Big bang
- Feature-driven extraction
  - Extract a service when there is a strong reason to change or add the underlying functionality
- Incremental migration (strangler pattern)
  - API gateway / proxy layer - It decides:
    - send to monolith (legacy path)
    - send to a new microservice
  - Extract one functionality at a time
  - Incrementally replace more functionality
  - Retire the monolith
- “Distributed monolith” (antipattern)



https://turnoff.us/geek/enterprise-vs-startup-journey-to-cloud/

Monolith vs microservices today

“Extreme” microservices phase has taught us a few lessons:

- Very small, independent services are hard to maintain, test, deploy, and debug
- Operational overhead (orchestration, monitoring, networking) is significant
- Latency chain grews
- Many applications don’t need microservices immediately

Some companies (Amazon, Netflix,..) have already consolidated their architecture [9]

- Merge microservices to larger bounded services (sometines internally modular monoliths)
- Microservice architecture might still be present

Designing a new system

Modular monolith / hybrid architecture - a practical alternative to jumping straight into microservices

Recommended approach (Martin Fowler):

1. Start with a simple design (modular monolith)
2. As complexity grows or requirements change, refactor, extract services, and adapt

-> aligns with moderns principles

- Avoid Big Design Up Front (BDUF)
- Build to change instead of build to last

So my primary guideline would be **don't even consider microservices unless you have a system that's too complex to manage as a monolith**. The majority of software systems should be built as a single monolithic application. Do pay attention to good modularity within that monolith, but don't try to separate it into separate services.[3]

Summary

- Metrics to evaluate your architecture
  - Simplicity/Complexity
  - Agility
  - Scalability
  - Maintainability
  - Developer experience (local development, debugging)
  - Fault tolerance
  - Technology independence
  - Communication overhead
- Service Oriented Architecture, Microservices, Composable architecture reference similar concepts - do not get too hung on the details!
- Beware of the marketing lingo - focus on the underlying principles!
- Complexity kills - apply just enough architecture!

References & further reading

[1] Service Oriented Architecture : What Is SOA?, The Open Group SOA Working Group.

[2] Thomas Erl: Service-Oriented Architecture: Analysis & Design for Services and Microservices (Second Edition)

[3] Martin Fowler: Microservices, Microservice Premium

[4] Microservices • Martin Fowler • GOTO 2014 (YouTube video)

[5] Martin Fowler: Two Pizza Team

[6] James Lewis: HOW BIG SHOULD A MICRO-SERVICE BE?, originally published 2013

[7] Martin Fowler: Conway's law

[8] Chris Richardson: Pattern: Saga

[9] Joab Jackson: Year-in-Review: 2023 Was a Turning Point for Microservices

[10] Michel Murabito: Composable Architectures vs. Microservices: Which Is Best?

[11] Michal Kostič: Service-oriented architectural patterns