MICROSERVICES WITH PYTHON

INTRODUCTION

WHO AM I?

- Hi, I'm Elias Salam
- Software Engineer for 12 years
- Worked 10 years in Telco R&D
- Now working as Tech Lead in energy Management scale-up (Citron)

BEFORE WE BEGIN

https://forms.gle/Y9ztkrL6a4ECFhkZ8

I'LL BE HAPPY IF AFTER THIS COURSE YOU

- Understand why some companies have chosen microservices architecture
- Identify the tradeoffs of microservices architecture
- Recognize use cases where microservices make sense
- Can critically assess and explain the trade-offs behind architectural decisions
- Have gained experience in solving typical microservices architecture complexities
- Know the main tools used on microservices stack

PLAN

- Part 1: What are microservices
- Part 2: Designing microservices
- Part 3: Microservices communication
- Part 4: Microservices Techniques
- Part 5: Enabling technologies

PART 1 -
WHAT ARE MICROSERVICES

WHAT ARE MICROSERVICES: PLAN

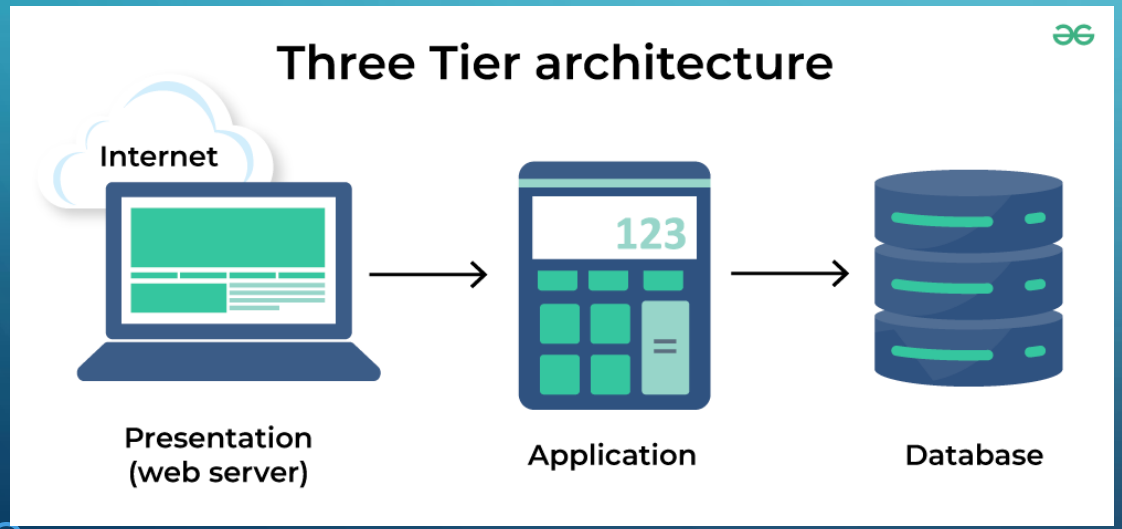
1. The monolith limitations
2. Microservices definition
3. Microservices trade-offs
4. Real life use cases
5. Workshop

WHAT ARE MICROSERVICES

1. THE MONOLITH

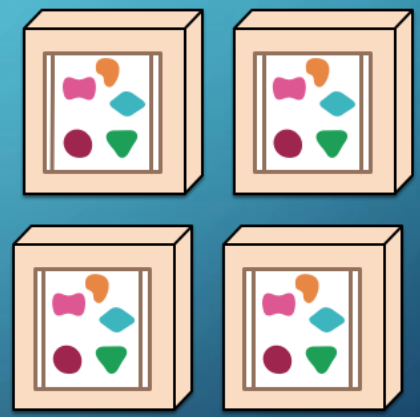
TRADITIONAL APPLICATION DEVELOPMENT : MONOLITH

- Self contained application
- Single codebase



A monolithic application puts all its functionality into a single process...

... and scales by replicating the monolith on multiple servers



MONOLITH LIMITATIONS

The bigger they are, the harder they fall

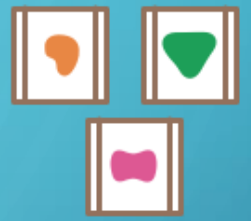
MONOLITH LIMITATIONS: BUILDING HUGE APPLICATION

- Build phase:
  - Understanding impact of a change to a feature is harder
  - Constant merge conflict
- Delivery phase: long deploy cycle (compilation, test, deploy)
- Run phase:
  - Big application requires a lot of resources
  - Fault-tolerance: If one feature crash the whole application crashes

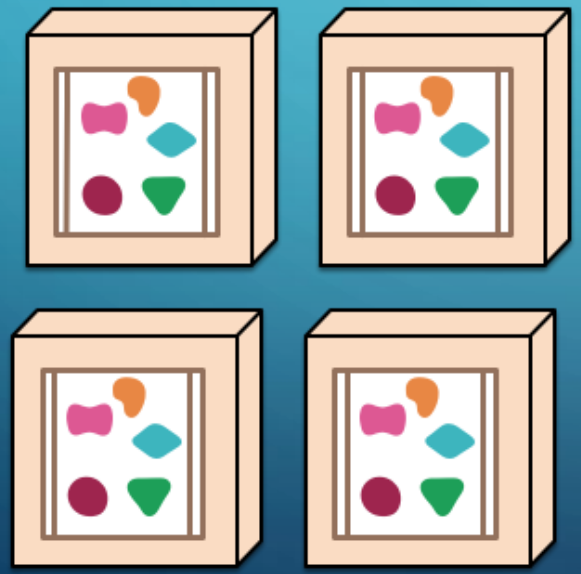
MONOLITH LIMITATIONS: SCALABILITY

A monolithic application puts all its functionality into a single process...

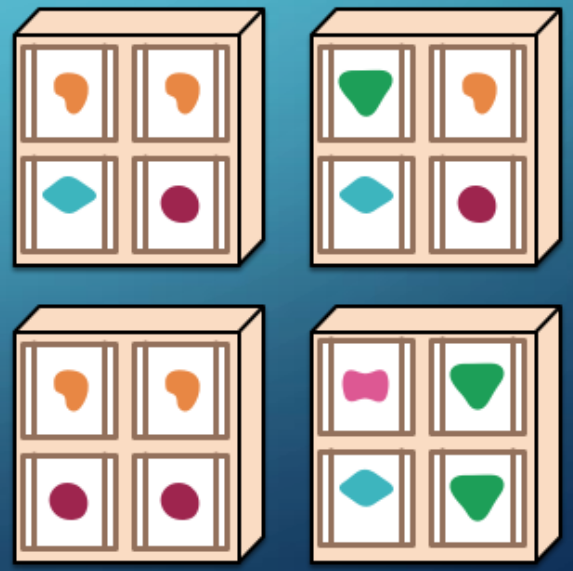
A microservices architecture puts each element of functionality into a separate service...



... and scales by replicating the monolith on multiple servers



... and scales by distributing these services across servers, replicating as needed.



Source: https://martinfowler.com/articles/microservices.html

MONOLITH LIMITATIONS: ORGANISATION

- Many developers contributing to the same codebase
  - Changes to one area can break seamingly unrelated areas
  - Growing complexity
- Coupled teams: teams have to coordinate even for the minor changes
- Big bang deployment: every deployment contains a lot of changes
- Team autonomy: architecture decisions should be shared among the whole company
- Result: Scaling teams does not scale the system

WHAT ARE MICROSERVICES

2. MICROSERVICES DEFINITION

WHAT ARE MICROSERVICES

- Independently releasable services
- Modeled around business domain
- Manage their own state
- Encapsulate functionalities and make them available via network
- Technology agnostic

INDEPENDENT DEPLOYABILITY

- The most important feature of microservice architecture
- We can deploy a single microservice without deploying any other service
- Services must be losely coupled: no shared database
- Requires well-defined and stable contracts
- Why?
  - Accelerate the time to market and remove cross team synchronization

MODELED AROUND BUSINESS DOMAIN

- A microservice should contain a business functionality not (only) a technical one
- Example: user notification service instead of email service

- Why?
  - Changing the requirement of a feature should be constrained to as few microservices (hence teams) as possible

MANAGE THEIR OWN STATE

- Each microservice is responsible for it's own data
- When a microservice requires data managed by another microservice it should ask that second microservice
- Data manager should have the business rules allowing each operation
- Why?
  - Independent deployability requires it
  - Information hiding: separate what can change easily and what is difficult to change

ENCAPSULATE FUNCTIONALITIES AND MAKE THEM AVAILABLE VIA NETWORK

- A microservice exposes API to read (query) its data
- It also exposes allowed commands on its domain
- All API should be only callable through network

- Why?
  - Any other service can access the service through network

TECHNOLOGY AGNOSTIC

- Since a microservice is only accessed through network the team managing it is free to choose the best tools
- Any programming language or framework or database
- Microservice should still follow cloud best practices (12 factor apps) and company (metrics, tracing…)
- Why?
  - Allow teams to pick best tools for the job

SIZE?

- Microservices should provide a business feature
- The right size for a microservice is that it is easy to understand what it does
- It is a bad practice to focus on the size of a microservice just because there is micro in the name
- It is recommended to start bigger and then split if necessary

WHAT ARE MICROSERVICES

3. MICROSERVICES TRADE-OFFS

MICROSERVICES PERKS

- Scalability: We can scale services independently
- Resilience: If one service fails only a part of the app is down
- Team autonomy: Owned by a single team
- Faster deployment: A new feature can be deployed at any time

MICROSERVICES CHALLENGES

- Distributed system communication
- Data consistency between services
- Maintenance
- Compatibility
- Cost
- Team skills

DISTRIBUTED SYSTEM COMMUNICATION

- Network can and will fail
- Network adds latency

- Solutions:
  - Retries
  - Timeout
  - Circuit breaker

DATA CONSISTENCY

- Each service has its own database
- We cannot simply do SQL joins between tables
- ACID transactions are not possible
- Need to propagate lifecycle events
- Solutions
  - Eventual consistency
  - Saga
  - Cross domain search

MAINTENANCE

- Need to monitor each service
- Tracking a bug is hard since the failure can be in any microservice
- Need to maintain a CI/CD for each microservice
- Solutions
  - Centralized logging
  - Distributed tracing
  - Metrics & health checks

COMPATIBILITY

- independent deployability requires full backward compatibility

- Solutions
  - API contracts
  - Versionning

COST

- A lot of tooling is required to manage microservices in production
- Microservices duplicate data and processing
- Cost reduction from scaling different component does not always compensate the overhead

- Solutions:
  - Cost monitoring

TEAM SKILLS

- Each team should be able to manage its microservices
- Team members need to understand how each service is ran, deployed and monitored
- Team needs to add a lot of tools to manage system complexity and monitoring

- Solutions
  - DevOps culture

DISCUSSION

- Given these trade-offs, when would you recommend microservice architecture and when would you NOT recommend it?

MICROSERVICES DO NOT WORK WELL FOR

- Startups/new products:
  - Business changes too much to create stable contracts
- Small teams:
  - Supporting microservices tooling is expensive (microservice tax)
- On-premise software:
  - You cannot expect your customer to manage a complex infrastructure

MICROSERVICES WORK WELL FOR

- Scale-up trying to grow software rapidly
- Well established SaaS software
- Big companies to share functionalities

WHAT ARE MICROSERVICES

4. REAL LIFE USE CASES

WHY THEY BROKE THE MONOLITH

NETFLIX

- Scalability: Stream vs payment
- Velocity: 2 god classes: Movies, Customer
- Reliability: 2008 3 days outage due to data corruption

amazon

- Scalability: Uneven traffic
- Velocity: Double pizza team
- Reliability: You build it, you run it

Uber

- Scalability: moving from SF to the whole world
- Velocity: Every market is different (laws, cultures)
- Reliability: Frequent rollbacks

BEZOS MANDATE ~2002

- *1) All teams will henceforth expose their data and functionality through service interfaces.*
- *2) Teams must communicate with each other through these interfaces.*
- *3) There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.*
- *4) It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols -- doesn't matter. Bezos doesn't care.*
- *5) All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.*
- *6) Anyone who doesn't do this will be fired.*
- Source: https://news.ycombinator.com/item?id=18916406

CITRON CASE!

- Energy management company acquires Buildings management company
- Common services to share between products
- IOT applications are write heavy
- Killer feature: Representing real-estate as graphs

WORKSHOP #1 : GROUP DISCUSSION

- Pick an application and imagine how this application is split into microservices

PART 2 - DESIGNING MICROSERVICES

OBJECTIVES

- Microservices architecture should not be a goal. It is a solution to the problems identified with an existing monolith
- In this part our focus will be on how we should define microservice boundary and how to split a monolith

DESIGNING MICROSERVICES: PLAN

1. Domain Driven Design
2. How to define microservices boundaries
3. Monolith decomposition

DESIGNING MICROSERVICES

1. DOMAIN DRIVEN DESGIN

DOMAIN DRIVEN DESIGN

- Domain Driven Design is a way of building software that closely mirrors how a business or organization actually works. It emphasizes collaboration between developers and domain experts to create a model that's reflected in the code.

- The term was coined by Eric Evans in his book "Domain-Driven design: tackling complexity in the heart of software" 2003

DDD MAIN CONCEPTS DOMAINS

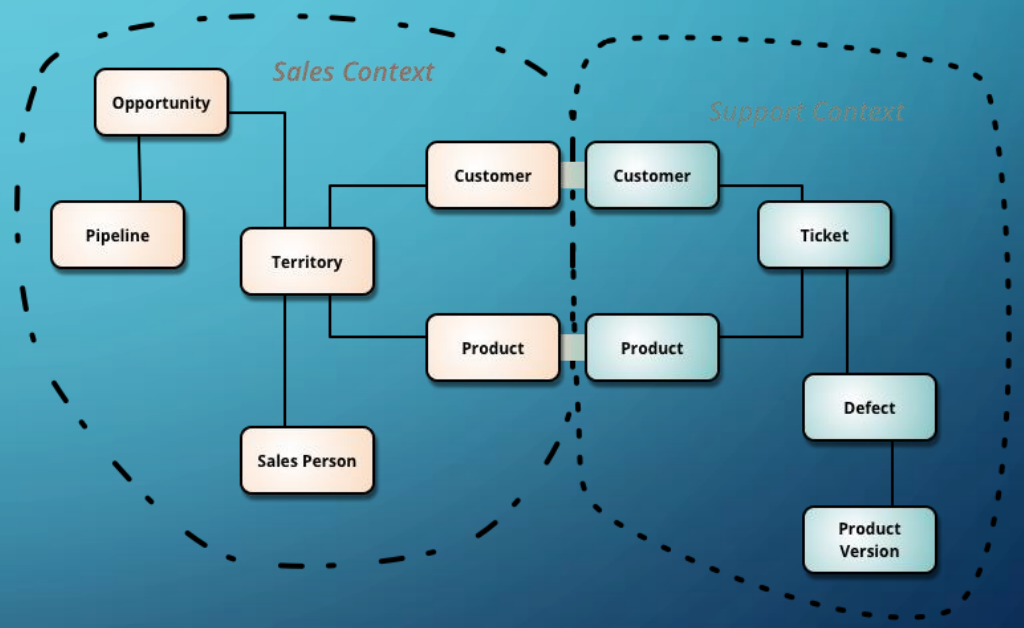
- Domain and subdomains: Each app operate in a domain and this domain can be split into subdomains
- Example: domain healthcare
  - Patient record
  - Appointment scheduling
  - Billing & insurance
  - Staff planning
  - User management
  - Notifications

DDD MAIN CONCEPTS
UBIQUITOUS LANGUAGE

- the practice of building up a common, rigorous language between developers and users
- Language should be based on the business model not the other way around
- Avoids misunderstandings and keep code aligned with the domain
- Allows domain experts to understand the test cases

DDD MAIN CONCEPTS BOUNDED CONTEXT

- A boundary where a particular domain model applies. Within this boundary, terms and logic are consistent and well-defined
- It breaks down complex systems into smaller manageable parts



DISCUSSION

- We are running an e-commerce application that sells books, cite a few bounded contexts? What data should each domain carry for a book?

DESIGNING MICROSERVICES

2. HOW TO DEFINE MICROSERVICES BOUNDARIES

BUSINESS DOMAIN BOUNDARY

- This should be the de facto option
- One microservice for each DDD bounded context
- Bounded context hide internal complexity and expose clear boundary
- If we want to change a functionality it is usually inside a single BC
- When a team works for a long time in a BC it develops empathy for the user and understanding of the business

VOLATILITY

- Identify the parts of the system going through more frequent changes and extract them into services
- Why?
  - High-changing components can cause instability across codebase
- Example: in e-commerce the user profile code changes rarely whereas promotions can change very frequently

DATA MANAGED

- The nature of some data might encourage its decomposition
- Why?
  - Some data might need specific privacy and security measures
- Example: Personally Identifiable Information (PII) to comform with GDPR
- Example: managing healthcare information requires specific security rules

TECHNOLOGY

- The need to use a specific technology can be a boundary factor
- Why?
  - Some component might require a specific technology
- Example: Graph database use case in Citron

ORGANIZATIONAL

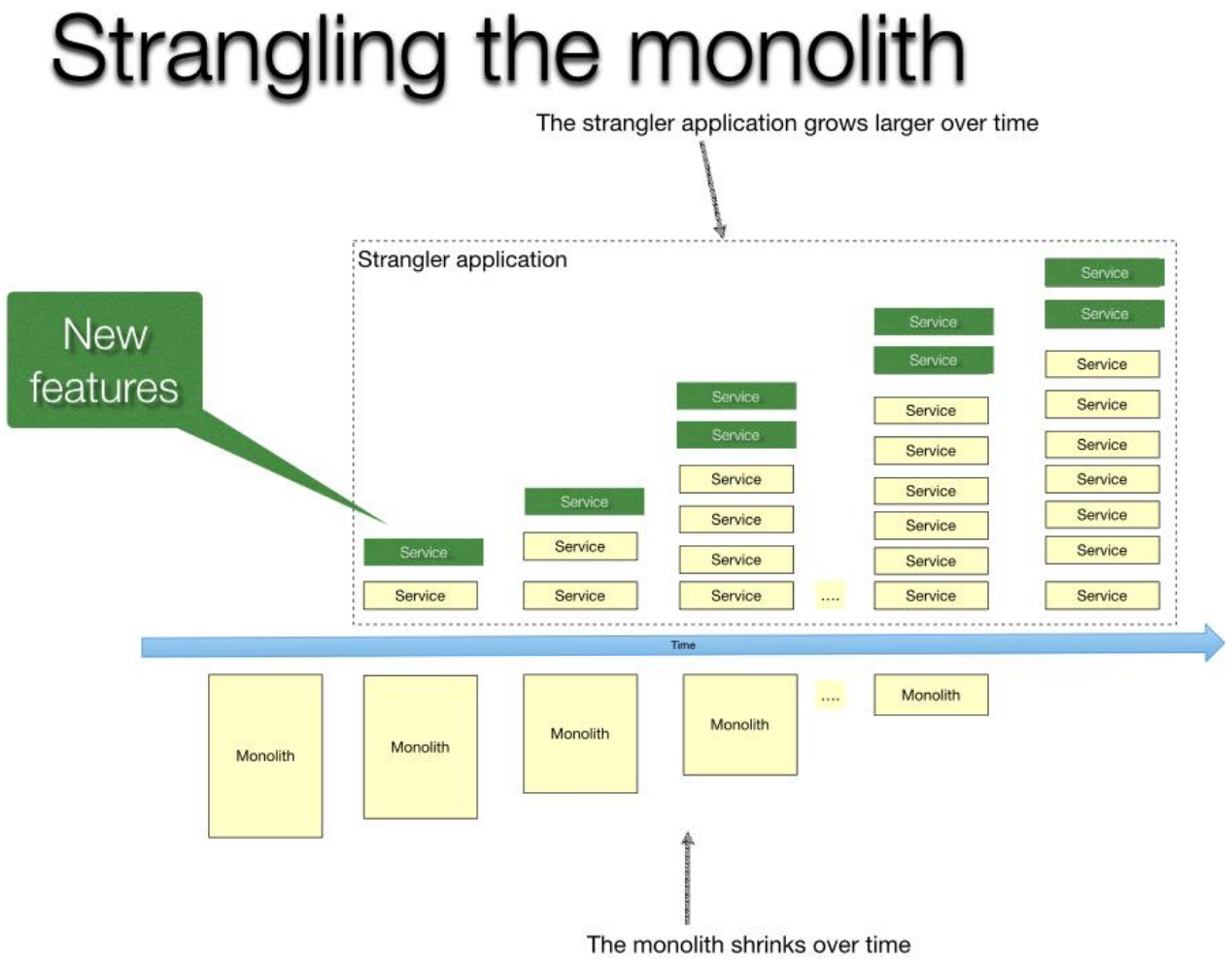
- Conway law: Oorganizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations
- It allows for clear ownership and fast development cycles
- Why?
  - Different teams might have different objectives and not collaborate together properly

DESIGNING MICROSERVICES

3. MONOLITH DECOMPOSITION

MONOLITH DECOMPOSITION

- Monolith should be split one part at a time
- First extract new features and most volatile domains
- “If you do a big-bang rewrite, the only thing you’re guaranteed of is a big bang” Martin Fowler

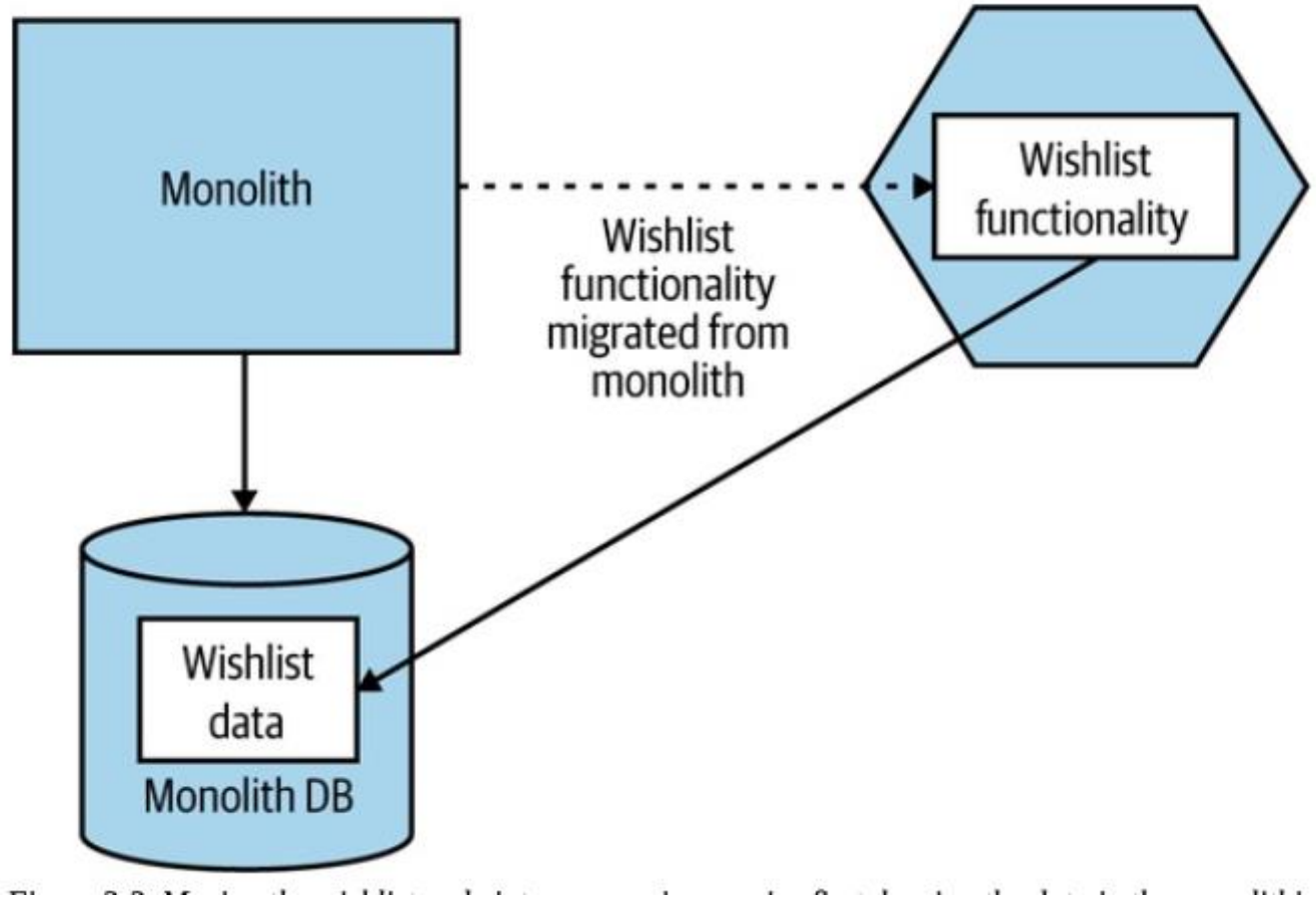


DECOMPOSITION TECHNIQUES

- Code first
- Data first
- Strangler fig pattern
- Parallel run
- Feature toggle

CODE FIRST

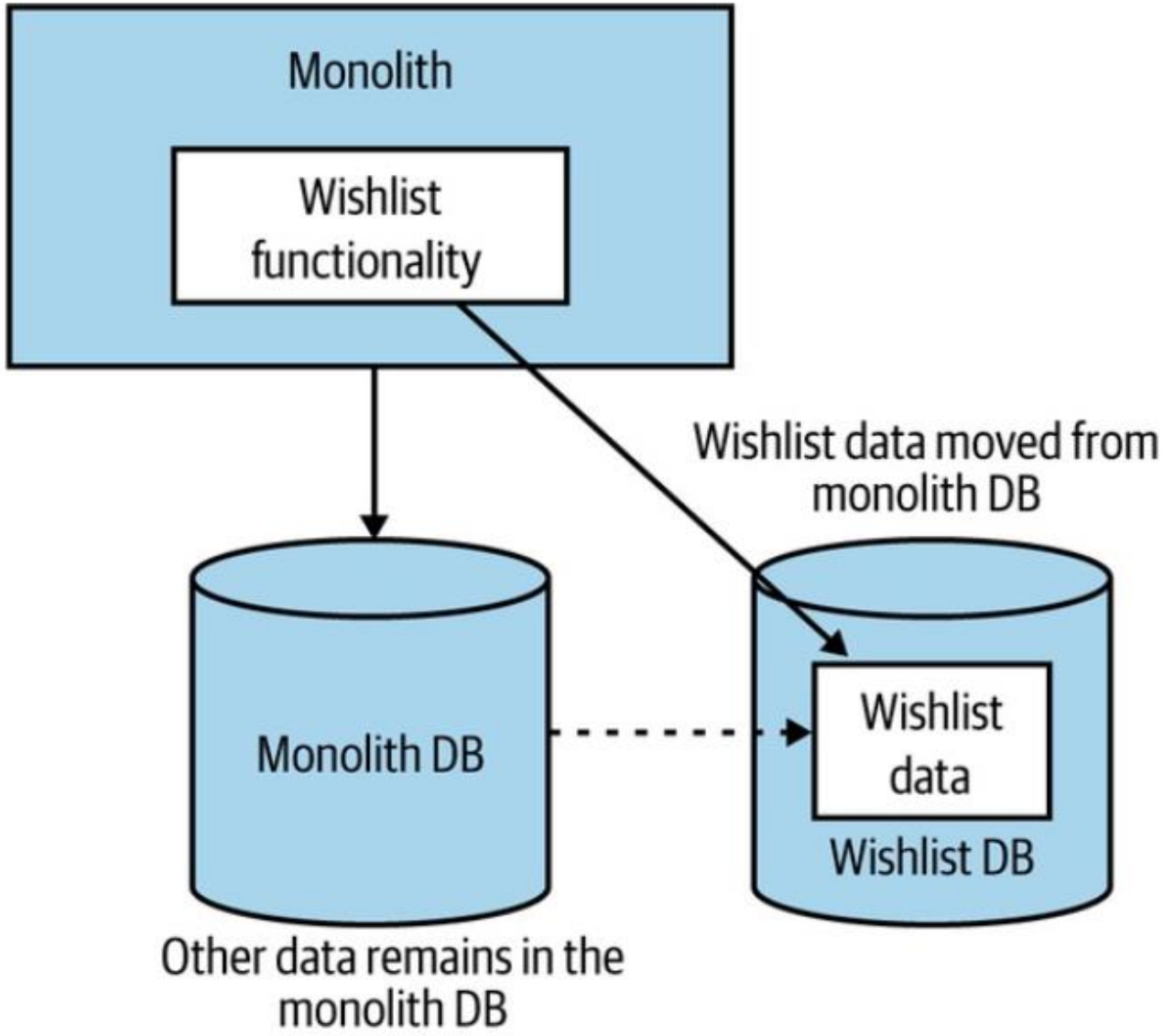
- Create an independent service connected to the monolith database
- Allows new team to take ownership of the functionality quickly



Building microservices Designing fine-grained systems by Sam Newman

DATA FIRST

- Extract the data into a specific database and connect the monolith to both databases
- This approach is used to prove microservice extraction is possible
- Forces to deal ASAP with loss of data integrity and transactions

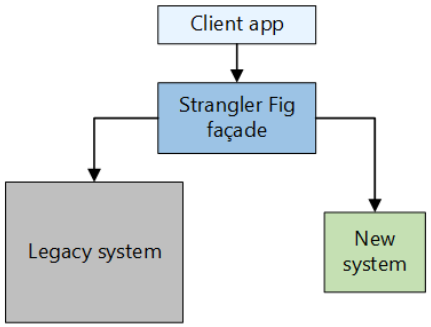


Building microservices Designing fine-grained systems by Sam Newman

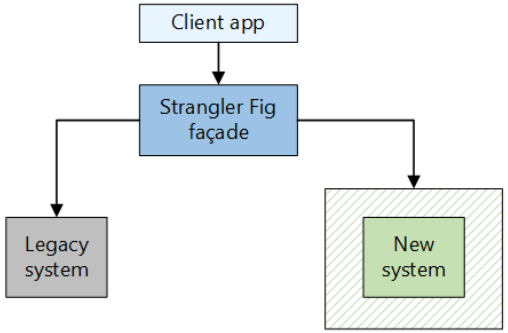
STRANGLER FIG PATTERN

- Common refactoring technique
- Avoids big bang refactoring
- Allows to validate new system in production before full rollout

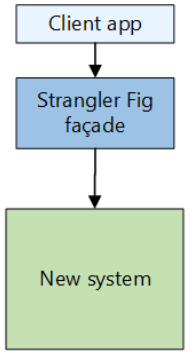
1 A façade routes client requests between the legacy system and the new system



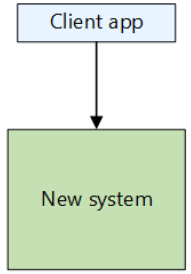
2 Incremental decomposition shifts functionality from the legacy system to the new system



3 The legacy system is fully decommissioned and has no dependencies



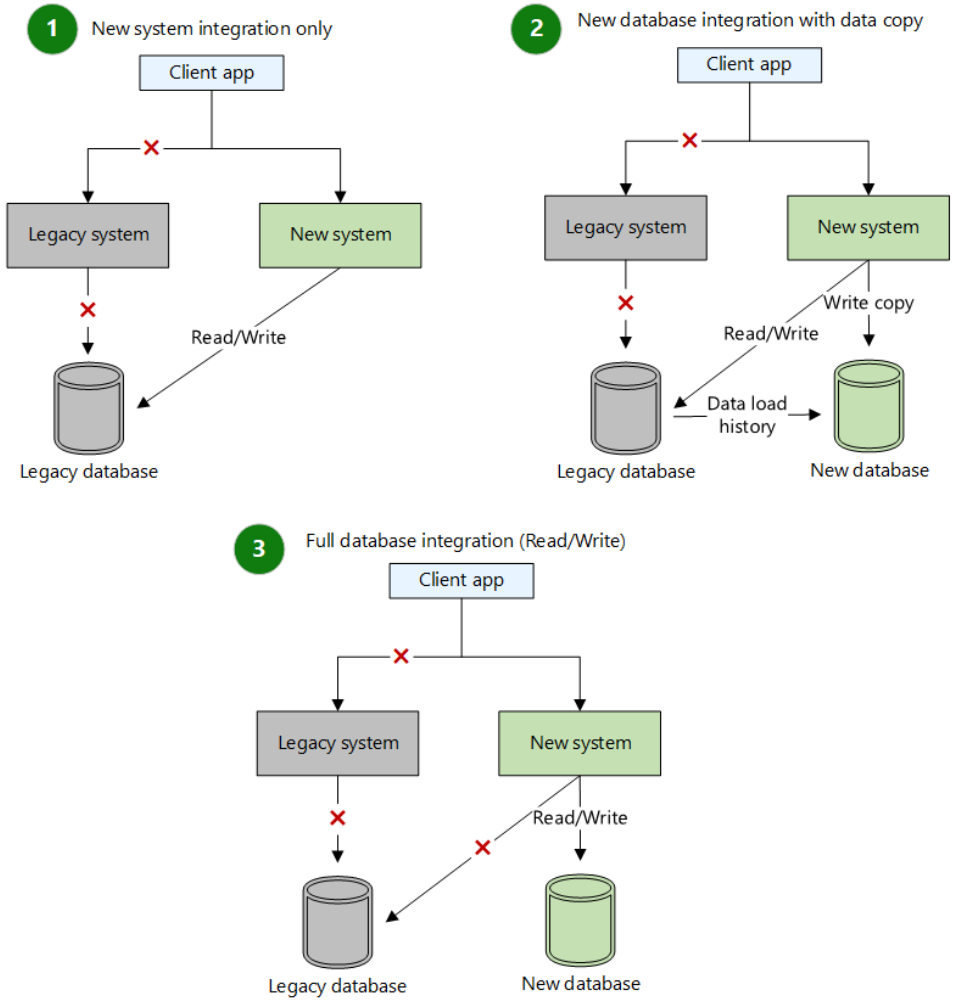
4 The façade is removed, and the client interacts directly with the new system



Microsoft Azure

STRANGLER FIG PATTERN

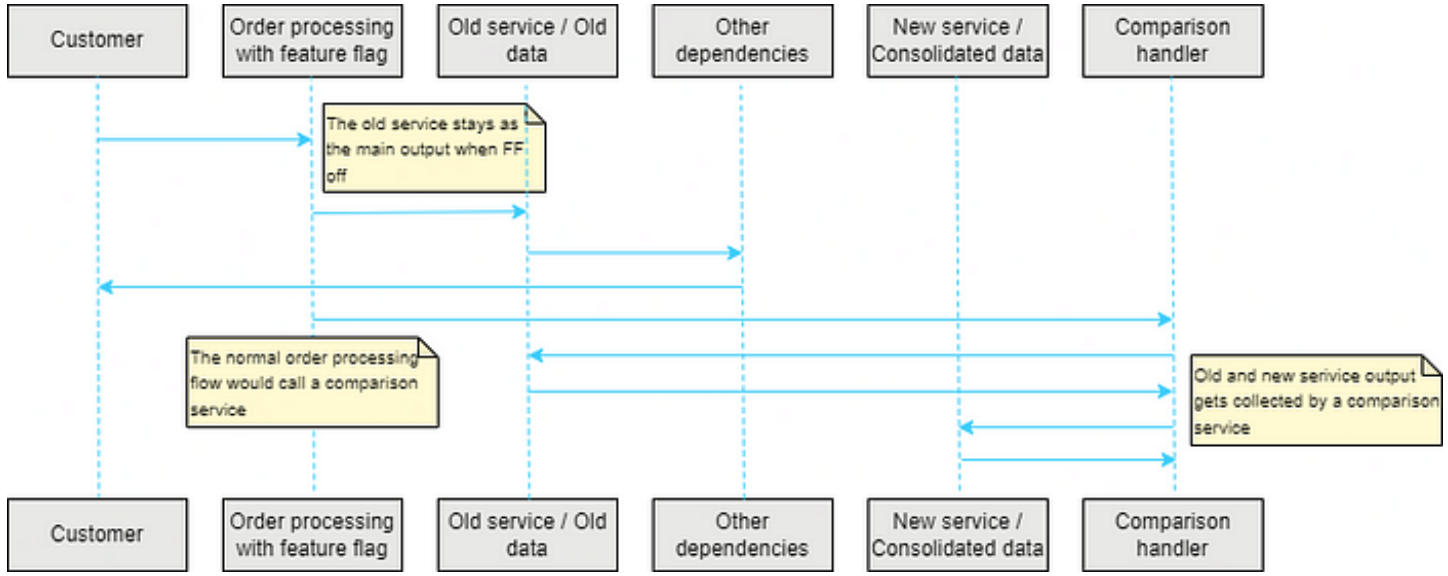
- Example of database migration



Microsoft Azure

PARALLEL RUN

- For critical migrations we could run the two versions of the service in paralllel
- Must decide which is the source of truth
- Should automate comparison



Example flow for a parallel run deployment

FEATURE TOGGLE

- Switch in the code to enable or disable a feature at runtime
- Enable safe testing in production
- Support canary releases with easy rollback
- Can be very simple (switch) or more complex (ratio, header based…)

PART 2: RECAP

- The right time to move to microservices is when the monolith starts hurting
- We use DDD to identify microservices boundaries
- The first microservice to extract is the one changing more frequently
- We use strangler fig, parallel run and feature toggles to migrate smoothly

PROJECT PITCH

“The Fit Company” is a stratup building cutting-edge AI-powered fitness coach.

The start-up is growing very fast and is having trouble scaling the team and its technology.

As part of the architecture team, your mission is to break down this application into microservices that can scale, evolve independently, and deliver a great user experience.

The AI Fitness Coach will allow users to:

- Set their fitness goals (e.g., weight loss, muscle gain, endurance)
- Receive personalized workout plans based on those goals
- Get dietary guidance tailored to their profile
- Manage their profile, physical metrics, and preferences
- Upgrade to premium plans and handle payments securely

WORKSHOP #2: IDENTIFY PROJECT “FIT” MICROSERVICES

PART 3 - MICROSERVICES COMMUNICATION

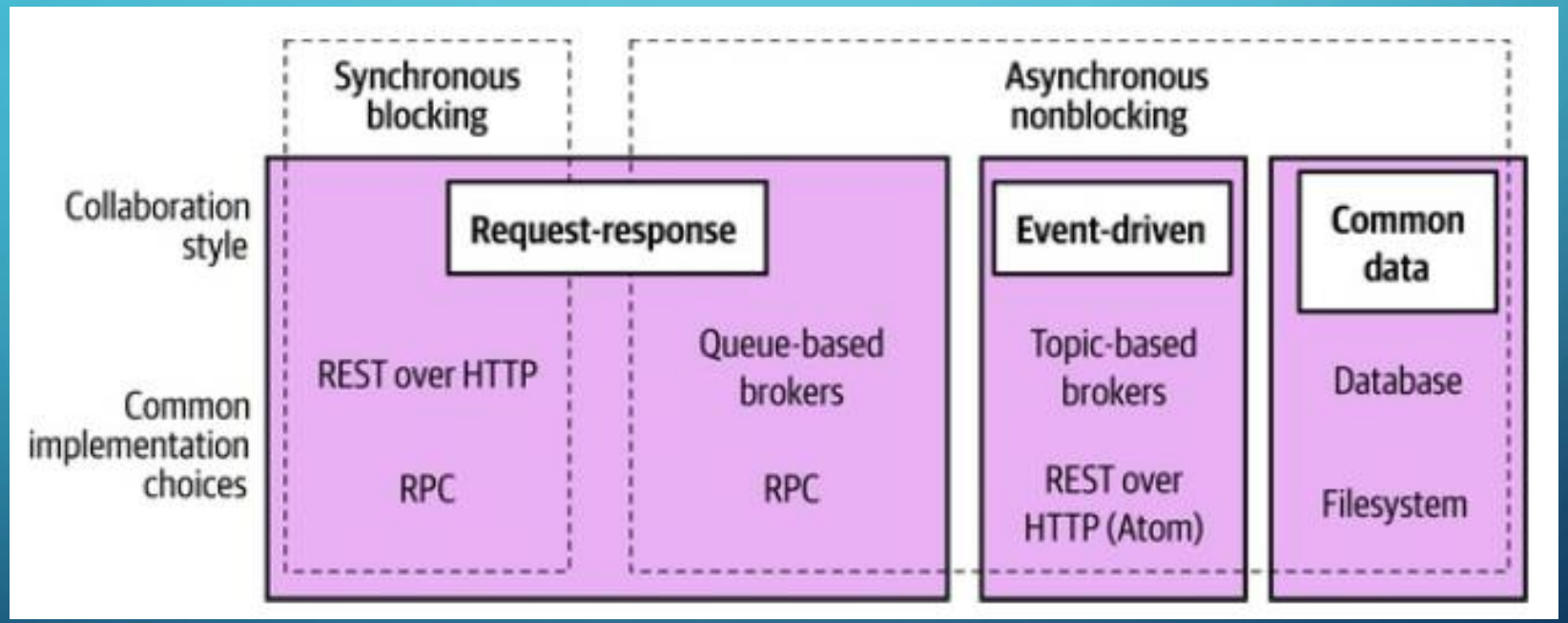
MICROSERVICES COMMUNICATION

- In a monolith, communication is just method calls
- In microservices: services must communicate over the network
- Since network is not reliable we need to design our system to tolerate failures

MICROSERVICES COMMUNICATION: PLAN

1. Synchronous blocking
2. Queue based communication
3. Event driven communication
4. Common data

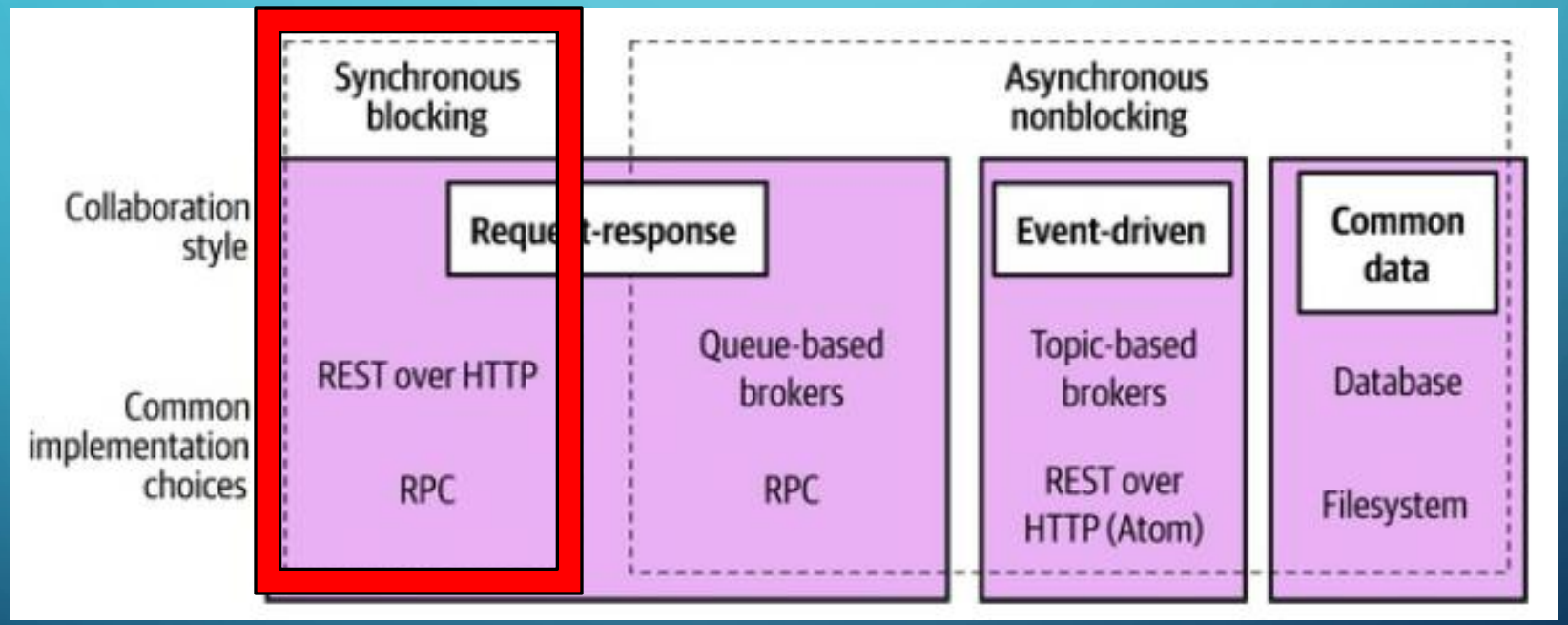
STYLES OF MICROSERVICE COMMUNICATION



MICROSERVICES COMMUNICATION

1. SYNCHRONOUS BLOCKING

SYNCHRONOUS BLOCKING



SYNCHRONOUS BLOCKING

- Service A sends a request to service B and waits for service B to return a response before continuing the processing
- Used when we need to receive the response or make sure the process was done
- Typical example:

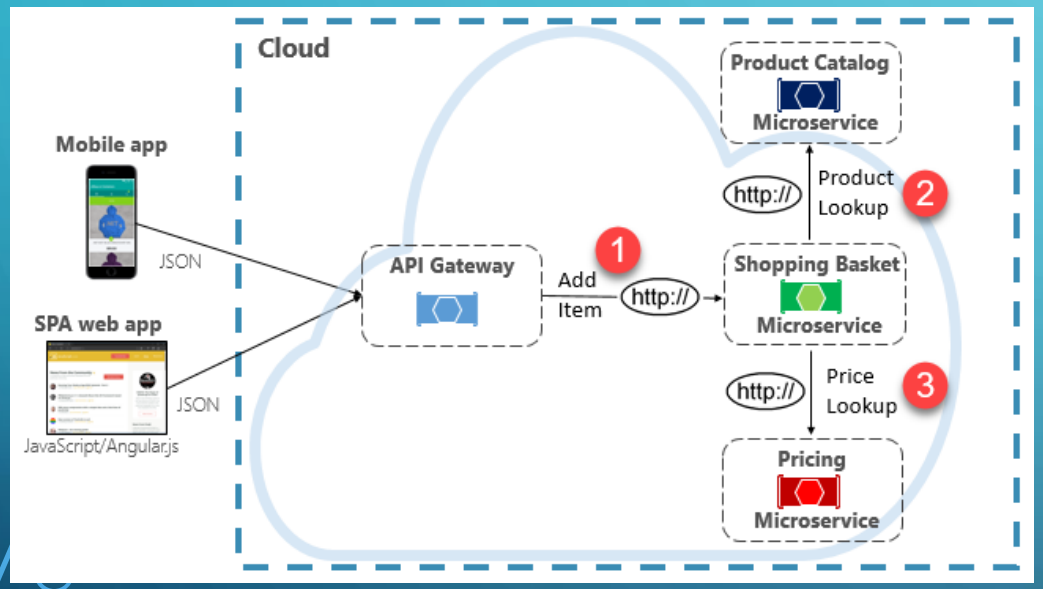
Service A → HTTP call → Service B → response → Service A continues

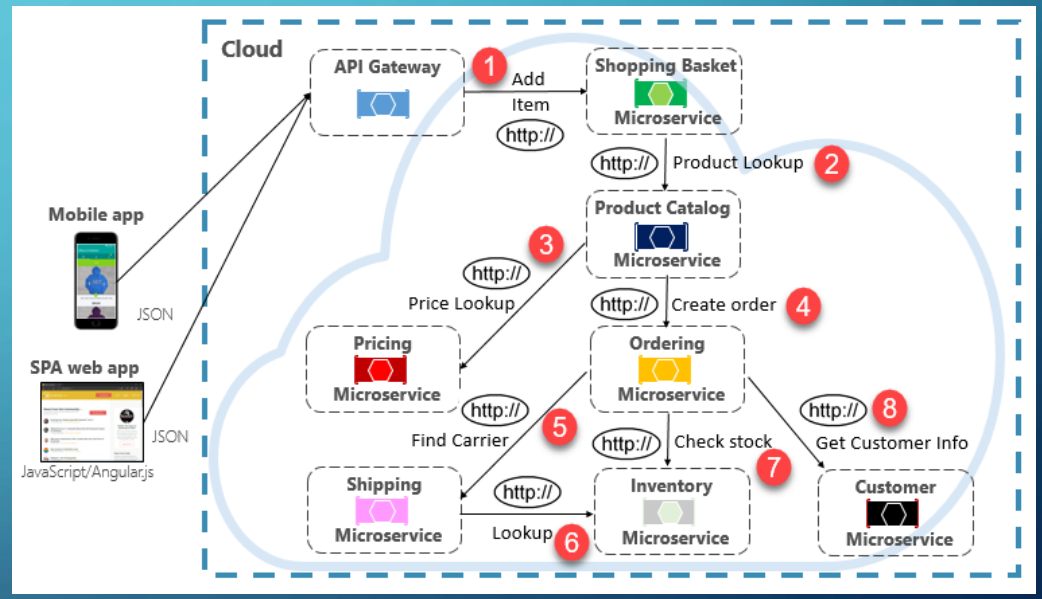
COMMON USE CASES

- Authentication
- Payment processing
- Inventory check

- Any time the client **needs** an immediate answer

SYNCHRONOUS CALL CHAIN





SYNCHRONOUS CALLS TRADE-OFFS

 Advantages

- Simple to implement
- Common tooling
- Easy to trace and debug
- Developers are familiar with it

 Challenges

- Chained latency
- Cascading failures
- Temporal coupling
- Instance coupling

BEST PRACTICES

- Avoid too many synchronous calls in the critical path
- Use retries to reduce failures rate
- Add timeouts to each call + global timeout
- Circuit breakers: stop calling failing service
- Fallbacks: keep service but degraded

SYNCHRONOUS COMMUNICATION PROTOCOLS REST

- Representational State Transfer is an architectural style inspired by the web
- Defines resources and actions on this resources e.g Users, Books, Orders
- Standard Http verb to interact with those resources
  - POST to create a resource
  - GET to retrive resource
  - PUT to update resource
  - DELETE to delete resource
- Typically uses HTTP with Json body
- De facto API standard with a lot of tooling
- Documentation using OpenAPI

SYNCHRONOUS COMMUNICATION PROTOCOLS REST TRADE-OFFS

 Advantages

- Simple and widely adopted
- Human-readable
- Lots of compatible tools

 Challenges

- No built-in contract (OpenAPI widely used)
- JSON string serialization is expensive
- Not everything is a resource => leads to bad designs

SYNCHRONOUS COMMUNICATION PROTOCOLS GRPC

- Framework created by Google for high performance RPC
- Uses HTTP/2 natively
- Binary data serialization using Protocol Buffers
- Supports bi-directional streaming
- Widely used in big tech and cloud open source technologies (Docker, Kubernetes)

SYNCHRONOUS COMMUNICATION PROTOCOLS GRPC TRADE-OFFS

✅ Advantages

- Low latency
- Low network load
- Strong type safety
- Tools for backward compatibility
- Code generation supported by Google

⚠️ Challenges

- Harder to debug: not human-readable
- Requires tooling for code generation
- Not compatible with browsers

GRPC CONTRACTS

```
service TodoListService {
  rpc CreateTask (CreateTaskRequest) returns (CreateTaskResponse) {}
  rpc GetTask (GetTaskRequest) returns (GetTaskResponse) {}
  rpc ListTasks (ListTasksRequest) returns (ListTasksResponse) {}
}

enum Status {
  STATUS_UNSPECIFIED = 0;
  STATUS_TODO = 1;
  STATUS_DONE = 2;
  STATUS_ON_GOING = 3;
}
message Task {
  string name = 2;
  google.type.DateTime created_at = 4;
  optional google.type.DateTime finished_at = 5;
  Status status = 6;
  uint32 uid = 7;
}

message CreateTaskRequest {
  string name = 1;
}
message CreateTaskResponse {
  Task task = 1;
}
```

WORKSHOP #3: SYNCHRONOUS CONTRACTS

1

List which API should be synchronous blocking on your microservices

2

Write an OpenAPI contract for one service

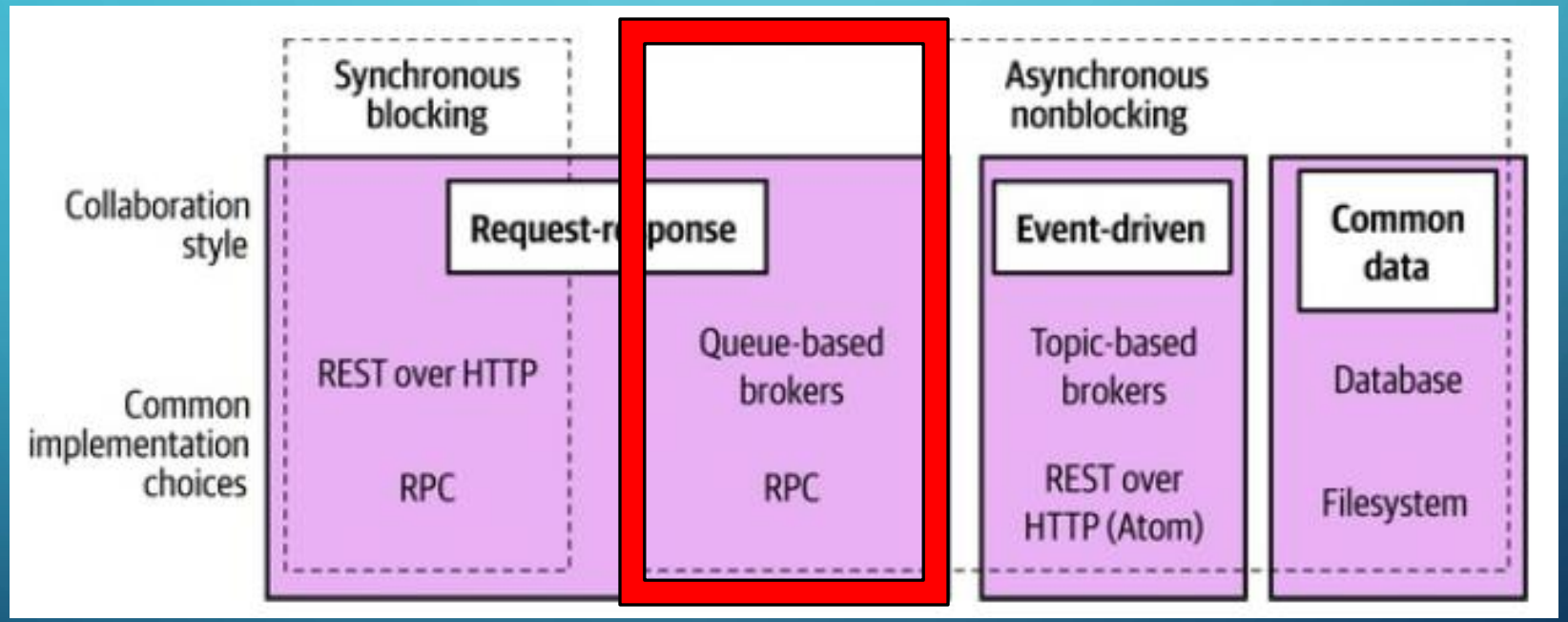
3

Write a protobuf contract for a second service
https://www.protobufpal.com/

MICROSERVICES COMMUNICATION

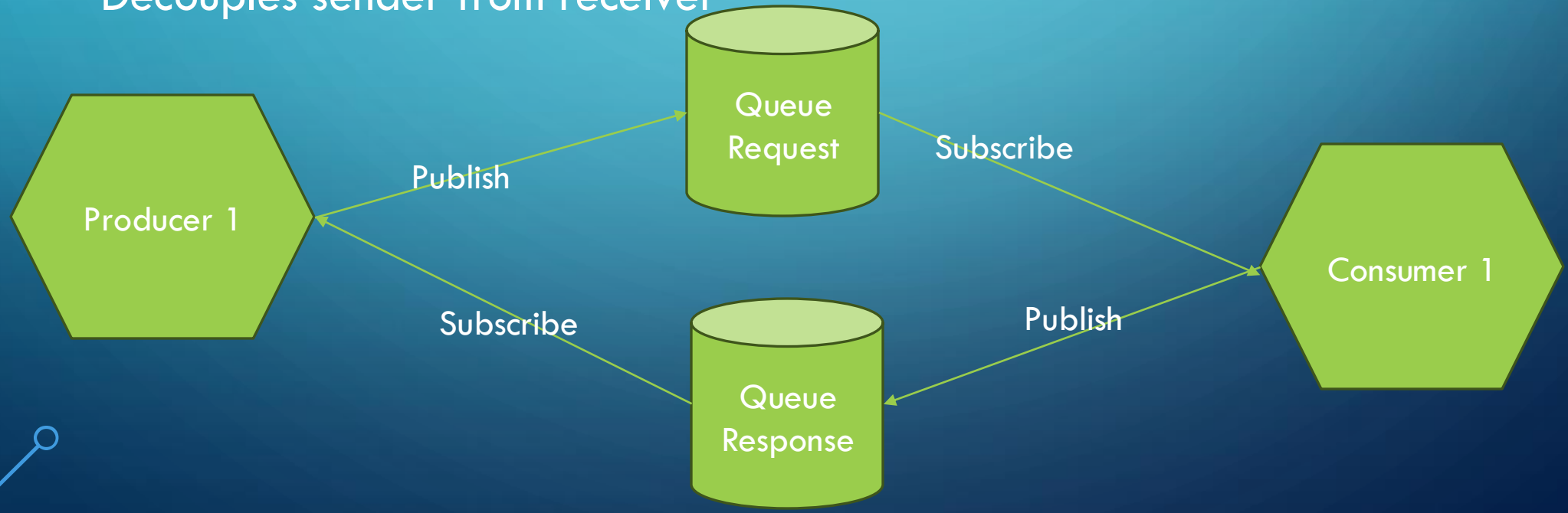
2. QUEUE-BASED COMMUNICATION

QUEUE BASED COMMUNICATION



QUEUE BASED COMMUNICATION

- Communication pattern where services exchange messages using a message queue (broker)
- Decouples sender from receiver



COMMON USE CASES

- Order processing
- Notification service
- Task scheduling
- IOT data ingestion
- Batching LLM calls

QUEUE BASED COMMUNICATION TRADE-OFFS

 Advantages

- Loose coupling between services
- Fault tolerance
- Allow load based scalability
- Async retries and dead letter queues

 Challenges

- Debugging is harder
- Message may be duplicated or out of order
- Broker is SPOF
- Increase infrastructure complexity

BEST PRACTICES

- Use idempotent consumers to safely reprocess messages
- Keep consumers short-running
- Implement DLQ as first class citizen
- Use messaging contract and versioning (AsyncAPI)
- Monitor DLQ, processing time, queue length, completed per minute…
- Set timeout and retry policy
- Scale based on the load of the queue

POPULAR BROKER TECHNOLOGIES

- RabbitMQ
- Apache Kafka
- NATS
- Redis Streams
- Cloud providers queues

WORKSHOP #4: QUEUE CONTRACTS

1

List which API should use queuing on your microservices

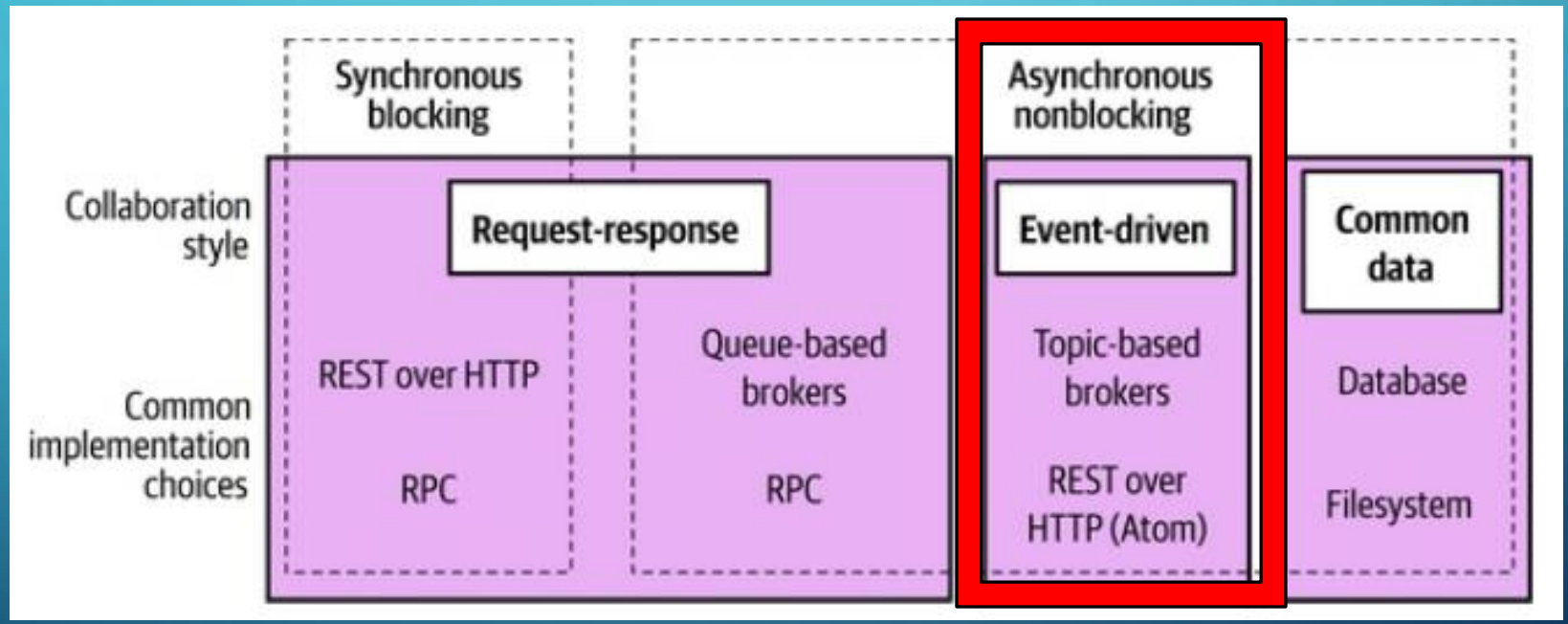
2

Write an AsyncAPI contract for one service

MICROSERVICES COMMUNICATION

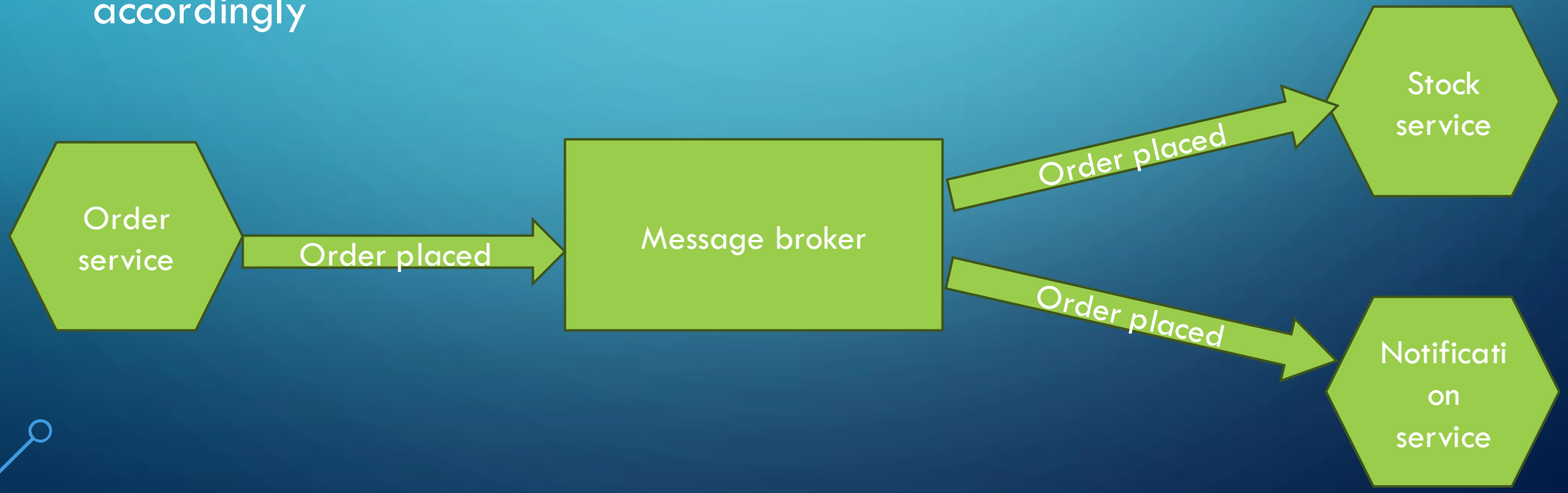
3. EVENT DRIVEN

EVENT DRIVEN



EVENT DRIVEN

- An event is a statement about something that has occurred.
- A microservice broadcast events assuming interested parties will react accordingly



COMMON USE CASES

- Order processing workflows
- User onboarding/offboarding
- Data synchronization across services
- Notifications

EVENT DRIVEN COMMUNICATION TRADE-OFFS

 Advantages

- No more domain coupling between services
- Increase domain & team autonomy
- Scalability and resilience under load
- Improved extensibility

 Challenges

- Debugging is harder
- Requires mind rewiring
- Hard to know what payload is required by each service
- Eventual consistency requires careful design
- Need to handle missed events

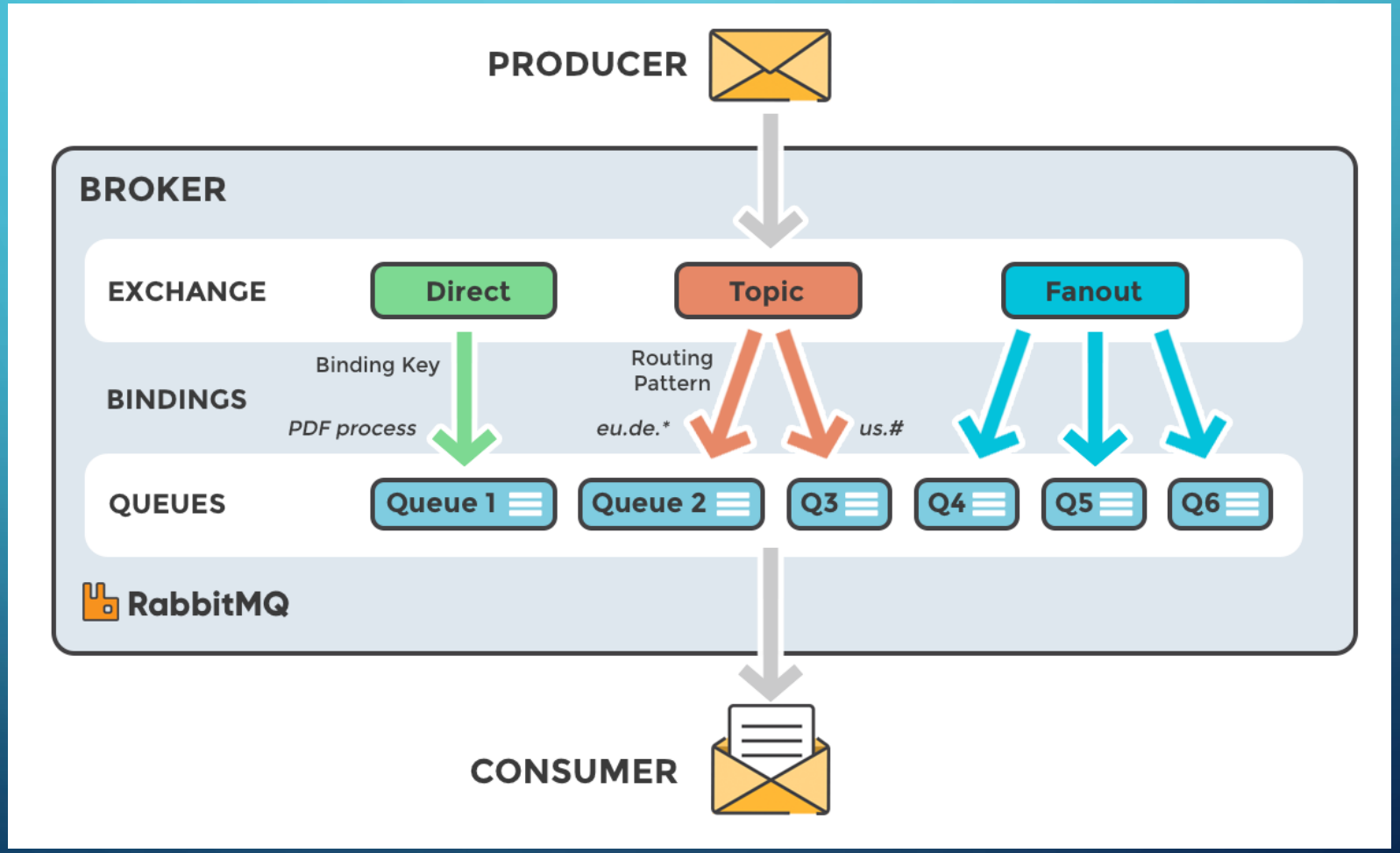
EVENT DRIVEN COMMUNICATION BEST PRACTICES

- Use event names should represent business fact with past tense: UserDeleted, invoicePaid
- Include event versioning and contract (AsyncApi)
- Use idempotent event handlers
- Implement dead letter queues
- Log all events for auditing and replayability

POPULAR BROKER TECHNOLOGIES

- RabbitMQ
- Apache Kafka
- NATS
- Cloud providers broker

RABBITMQ EXCHANGE TYPES

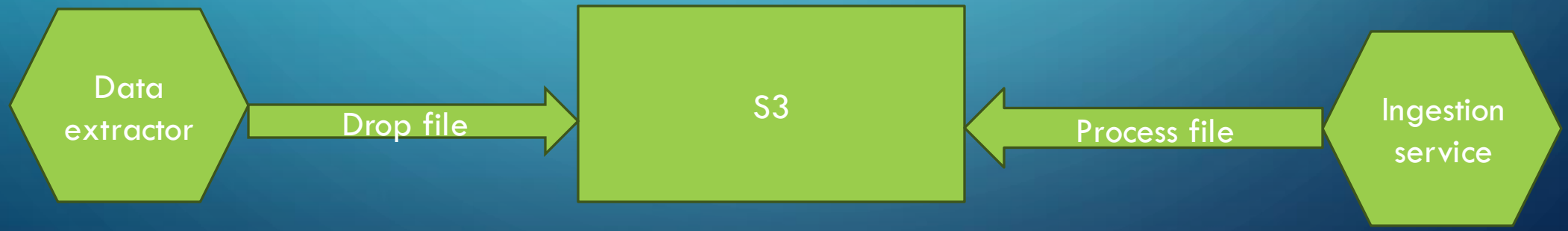


MICROSERVICES COMMUNICATION

4. COMMON DATA

COMMON DATA

- One microservice puts data into a defined location for other microservices to read
- Communication is indirect and usually asynchronous
- There is no direct API between services



COMMON USE CASES

- IOT data ingestion
- Data lake and data warehouse
- Batch processing

COMMON DATA TRADE-OFFS

 Advantages

- Remove temporal coupling
- Scalability and resilience under load

 Challenges

- No contracts between parties
- Difficult to enforce ownership
- Can lead to tight coupling

PART 4 - MICROSERVICES TECHNIQUES

MICROSERVICES TECHNIQUES: PLAN

1. Sagas
2. Handling breaking changes
3. Cross domain search
4. Reporting database

PART 4 - MICROSERVICES TECHNIQUES

1 - SAGA

SAGAS
THE PROBLEM – DISTRIBUTED TRANSACTIONS IN MICROSERVICES

- ACID transactions work well inside a single database
  - **Atomicity:** All operations succeed or no operations succeed.
  - **Consistency:** Data transitions from one valid state to another valid state.
  - **Isolation:** Concurrent transactions yield the same results as sequential transactions.
  - **Durability:** Changes persist after they're committed, even when failures occur.
- In microservices:
  - Each service owns its own database
  - No cross-database transcation guarateed
  - We lose atomicity when a workflow spans multiple services

SAGAS
THE PROBLEM – DISTRIBUTED TRANSACTIONS IN MICROSERVICES

Example: placing an order:

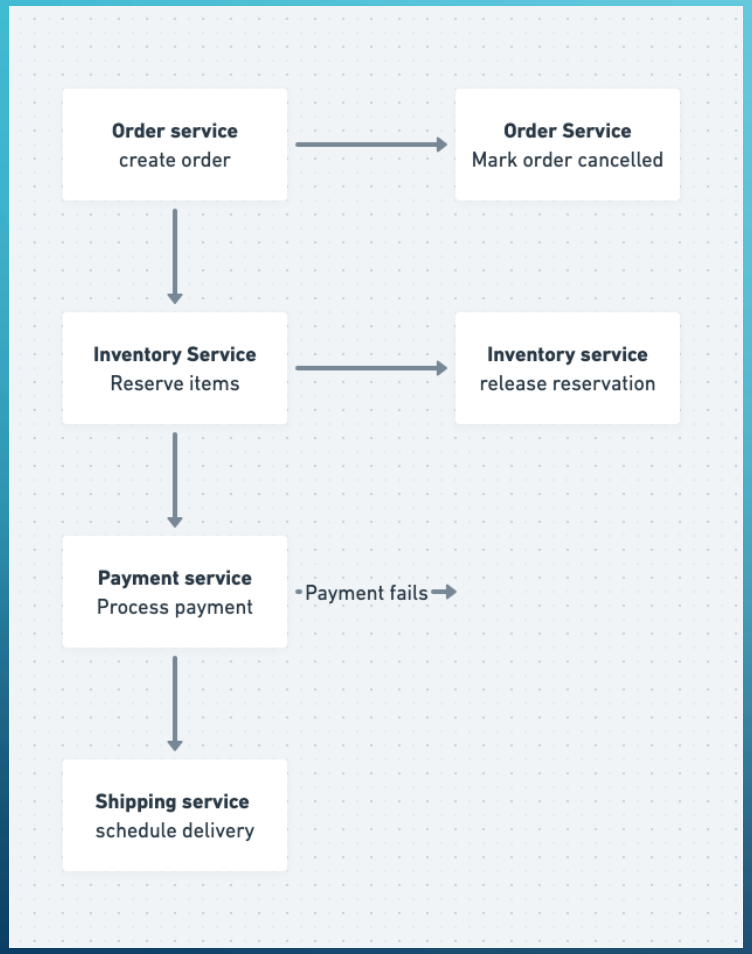
Customer was charged but we can't ship

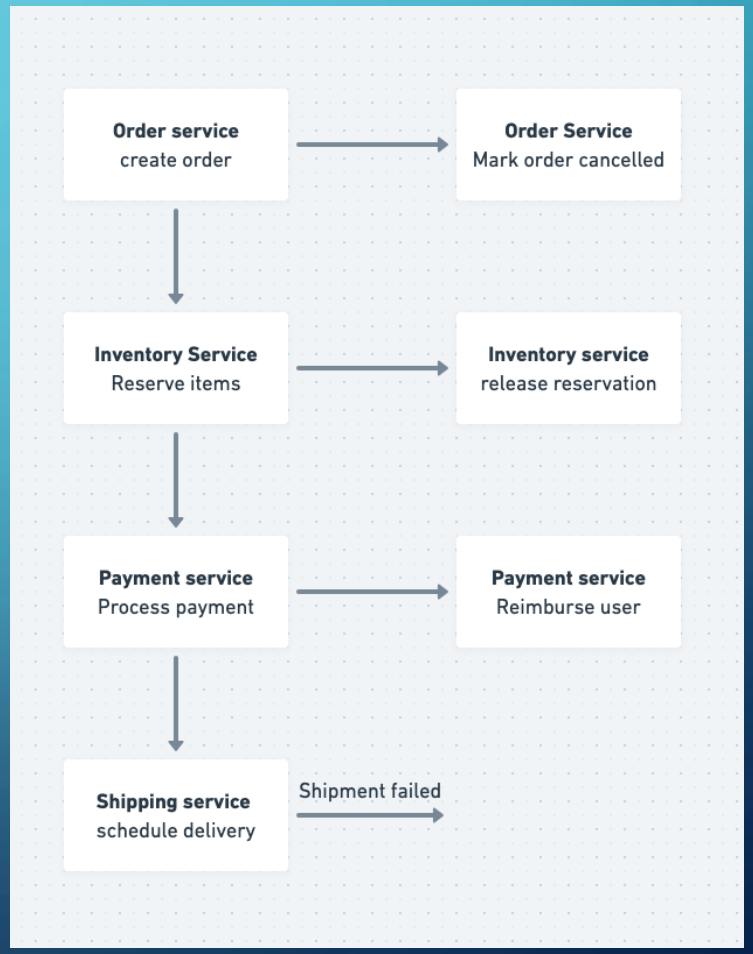
WHAT IS A SAGA?

- A Saga is a sequence of local transactions where each transaction triggers the next.
- If a local transaction fails, the saga performs a series of **compensation transactions** to reverse the changes that the preceding transactions made.

- Key Properties:
  - Guarantees eventual consistency
  - No need for distributed locking
  - Fault-tolerant by design

EXAMPLE: ORDER PROCESSING SAGA



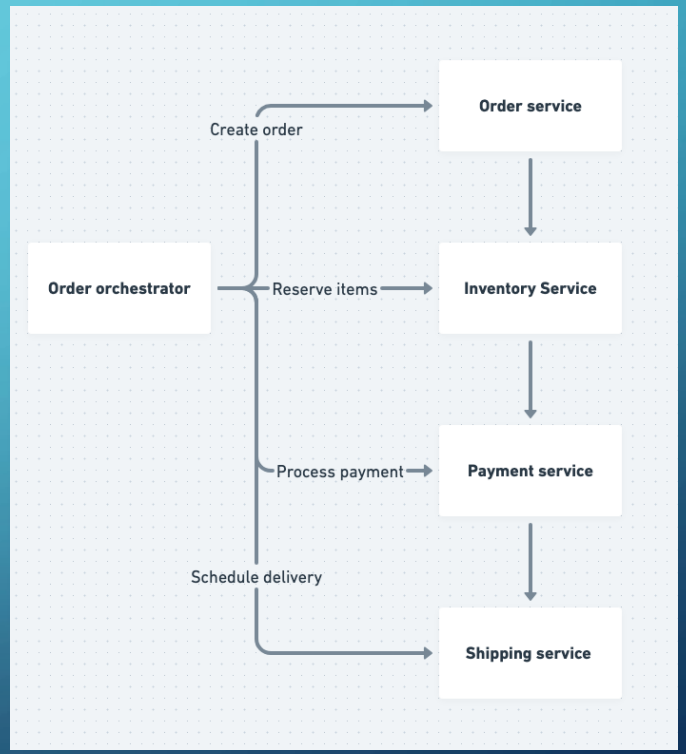


CHOREOGRAPHY-BASED SAGA

- Services listen to events and react accordingly
- No central coordinator
- Each service knows what to do next
- Events: Order created, Inventory reserved, Payment requested, Payment completed, Shipment planned
- Trade-offs:
  - ✅ Simpler
  - ❌ Harder to debug
  - ❌ Harder to change flow logic

ORCHESTRATOR-BASED SAGA

- A central controller (Orchestrator) manages the sage
- Orchestrator tells each service when to act
- Trade-offs:
  - ✅ Easier to manage complex flows
  - ✅ Centralized error handling
  - ❌ Tighter coupling
  - ❌ SPOF



COMPENSATING ACTIONS

- Compensating transactions are logical undos not rollbacks
- Example : Create order => Mark order cancelled and not delete created order
- Must be idempotent because failure can be triggered twice (i.e do not refund payment twice)

PART 4 - MICROSERVICES TECHNIQUES

2 – HANDLING BREAKING CHANGES

WHAT IS A BREAKING CHANGE?

- A breaking change is a modification to a service that causes dependent services or clients to fail, behave incorrectly, or require changes to continue functioning.
- In microservices architecture, many services might be consuming your APIs
- Usually a breaking change means one change in your service will cause other services to break.

HANDLE BREAKING CHANGES

- Use API & message versioning
- Detect breaking changes in CI/CD
- Validate contracts in CI/CD
- Maintain backward compatibility
- gRPC is great to well manage breaking changes

WHAT BREAKING CHANGES CAN WE INTRODUCE

POST /users/{id}/prefs

```
Request
{
    "firstName": "John",
    "lastName": "Doe",
    "email": "a@b.c",
    "phone": "+3361001",
    "currency": "EUR"
}
```

```
Response
{
    "date": "03/01/2025",
    "age": "37",
    "counrty": "FRA"
}
```

MAINTAIN BACKWARD COMPATIBILITY

- Never change field type
- Never delete a required field
- Add new fields instead of changing old fields
- Do not map network object to database objects
- Keep documentation of what is deprecated

PART 4 - MICROSERVICES TECHNIQUES

3 – CROSS DOMAIN SEARCH

CROSS DOMAIN SEARCH

- Cross-domain search refers to querying and aggregating data across multiple microservices
- Since each microservice data is separate we cannot perform SQL joins
- Example scenario:
  - In fitness coach search users who follow a “weight loss” workout and have had “low protein intake” recently
  - We need to search in Profile service, nutrition service and workout recommendation service

CROSS DOMAIN JOINS STRATEGIES

COPY DATA

Duplicate information in a service when they are commonly used for search

Trade-offs:

✅ Easy to implement

❌ Requires data synchronization patterns

❌ Should be designed in advance

API AGGREGATOR

A dedicated service calls other services APIs and assembles the results

Often uses caching, filtering or heuristics

Trade-offs:

✅ Easy to implement

❌ May be inefficient

❌ Hard to paginate and count

DATA INDEXING SERVICE

Each service emits events to a centralized search service (e.g Elasticsearch)

Trade-offs:

✅ Fast, flexible search

❌ Adds a new service an integrations

PART 4 - MICROSERVICES TECHNIQUES

4 – REPORTING DATABASE

WHAT IS THE REPORTING DATABASE PATTERN?

- The Reporting Database Pattern involves maintaining a separate data store that aggregates data from multiple microservices for analytics, reporting, or dashboards.

Why?

- Microservices own private databases
- Operational DBs are not designed for complex, read-heavy queries

USE CASES

- Business dashboards : active users, sales analytics
- Regulatory compliance or audits
- KPI aggregation
- IOT applications

IMPLEMENTATION PATTERN 1: EXTRACT

- A pipeline job will extract from each service database all changed data in the last x minutes and stores it in data file (Parquet)
- A second job will read the file and perform aggregations and transformations before inserting to the reporting database
- Analysts or dashboards query the reporting DB for insights
- Trade-offs
  - ✅ Easy to implement
  - ✅ Failure will only delay data freshness
  - ❌ Requires managing deletes, updates
  - ❌ Data freshness

IMPLEMENTATION PATTERN 2: EVENT

- Microservices emit events on data changes
- Data is transformed and sent to the reporting database (e.g., via ETL or streaming).
- Analysts or dashboards query the reporting DB for insights
- Trade-offs
  - ✅ Near real time data
  - ✅ Lower data volume
  - ❌ Requires each service to implement events publishing
  - ❌ Implementation complexity

PART 5 - ENABLING TECHNOLOGIES

ENABLING TECHNOLOGIES: PLAN

1. Observability: Logs, traces & metrics
2. Containers & orchestrators
3. Public cloud & serverless

ENABLING TECHNOLOGIES

1. OBSERVABILITY: LOGS, TRACES AND METRICS

WHY OBSERVABILITY MATTERS IN MICROSERVICES

- Observability is the ability to understand what's happening inside a system based on its external outputs.
- Observability is essential in microservices applications because
  - Microservices are distributed — one request may touch many services.
  - Failures become harder to detect and harder to debug.
  - Traditional debugging (logs only) is **not enough**.
- Goals of observability
  - Detect failures early
  - Understand performance bottlenecks

THE THREE PILLARS OF OBSERVABILITY

Pillar	Purpose	Key questions answered
Logs	Contextual data about events	What happened? Why?
Traces	End-to-end request flow	Where did it go? What took the longest
Metrics	Numeric measurements over time	Is it healthy? What is the trend?

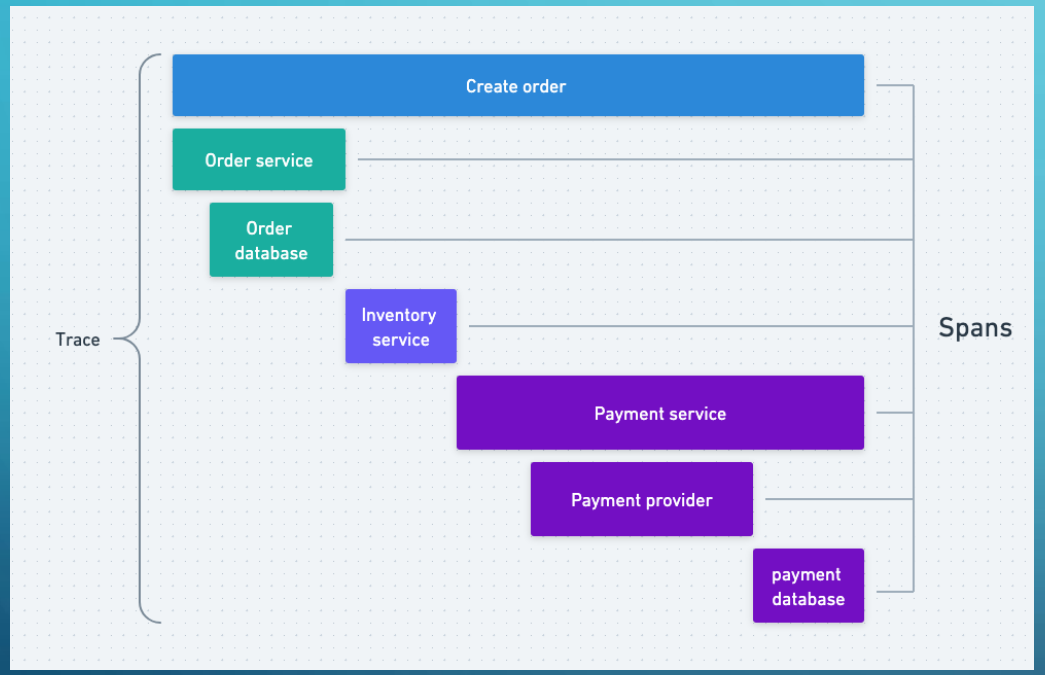
LOGS

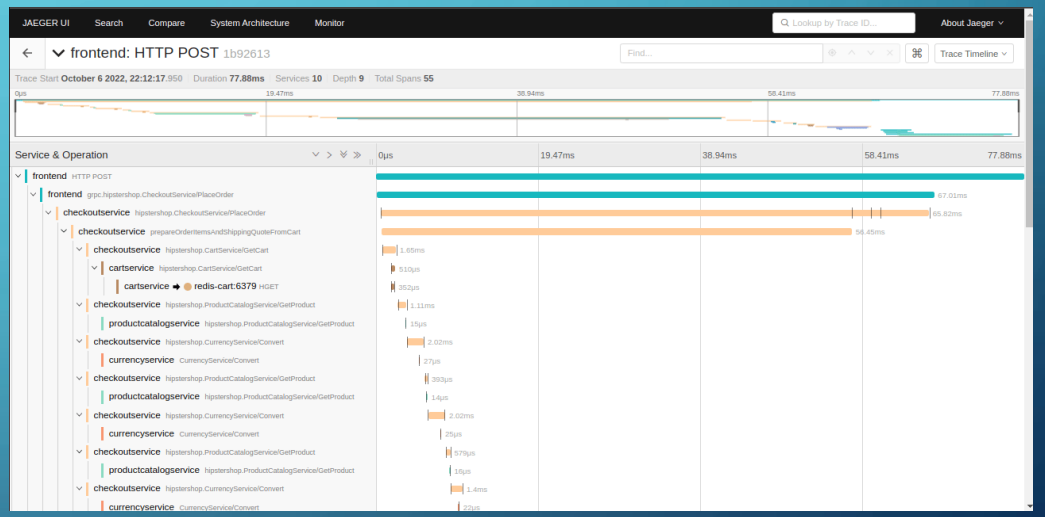
- Structured or unstructured messages describing events that happened during service execution.
- Usage:
  - Debugging and error diagnosis
  - Capturing contextual info (user, request ID, error details)
  - Forensics after incidents
- Best Practices:
  - Use structured logs (JSON preferred)
  - Include correlation IDs
  - Centralize using tools like ELK, Fluent Bit, or Loki

TRACES

- Traces track the entire lifecycle of a request as it flows through multiple microservices.
- Includes:
  - Spans: units of work in each service
  - Context propagation: pass request IDs across services
- Usage:
  - Identify bottlenecks in distributed flows
  - Trace errors across systems
  - Visualize request paths (e.g., Jaeger, Zipkin)
- Key Tools: OpenTelemetry, Jaeger, Zipkin, Datadog APM

TRACES


Create order
Order service
Order database
Inventory service
Payment service
Payment provider
payment database
Trace
Spans


JAEGER UI
Search
Compare
System Architecture
Monitor
About Jaeger
frontend: HTTP POST 1b92613
Trace Timeline
Service & Operation
0µs
19.47ms
38.94ms
58.41ms
77.88ms
frontend
checkoutservice
cartservice
productcatalogservice
currencyservice
redis-cart:6379

METRICS

- Numerical indicators collected over time, used to monitor system health and performance.
- Examples:
  - CPU/memory usage
  - Request per second
  - Error rates (5xx count)
- Usage:
  - Alerting and dashboards
  - SLO monitoring
  - Capacity planning

EXAMPLE OBSERVABILITY

- Scenario : an order processing fails

1. Logs show error in payment service with stack trace
2. Trace: shows the request flow: order service => Inventory service => payment service
3. Metrics: reveal a spike in 500 errors on payment service and drop in successful payments

ENABLING TECHNOLOGIES

2. CONTAINERS AND ORCHESTRATORS

WHAT ARE CONTAINERS & WHY WE USE THEM

- A container is a lighweight, standalone, executable package that includes:
  - Application code
  - Dependencies
  - System tools & libraries
- Why do we use containers in microservices
  - Consistency across dev, test and prod environment
  - Fast deployment and startup time
  - Isolation between microservices (e.g one container cannot glob up all the CPU)
  - Easy packaging and shipping
- Main technology: Docker

ORCHESTRATOR: KUBERNETES

- In a microservices world, you may have hundreds of containers. You need a tool to automatically manage these containers.
- Kubernetes (k8s) is an open-source container orchestration platform that automates:
  - Deployment: run containers reliably
  - Scaling: scale up/down based on load
  - Self-healing: restarts crashed containers
  - Networking & service discovery
  - Secrets & config management
  - Rolling updates & rollbacks
- K8s has become the industry standard for managing containers (and microservices) at scale

EXAMPLE MICROSERVICE DEPLOYMENT FILE

- Ensures there are 3 replicas of this container
- Injects environment variables
- Defines min & max resources
- Defines healthcheck
- Uses an auto-scaler base on memory usage

deployment.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: workout-recommendation
spec:
  replicas: 3
    spec:
      containers:
        - name: workout-service
          image: myregistry/workout:1.0
          env:
            - name: MODEL_PATH
              value: /models/v1
          resources:
            requests:
              memory: "256Mi"
              cpu: "250m"
            limits:
              memory: "512Mi"
              cpu: "500m"
          livenessProbe:
            httpGet:
              path: /health
              port: 8080
            initialDelaySeconds: 10
            periodSeconds: 10
            failureThreshold: 3
horizontalPodAutoscaler:
  enabled: true
  maxReplicas: 5
  avgMemoryUtilization: 80

```

ENABLING TECHNOLOGIES

3. PUBLIC CLOUD & SERVERLESS

PUBLIC CLOUD SERVICES

- Public cloud providers (AWS, Azure, Google cloud) offer many services to help managing microservices application: managed K8s, messaging systems, managed databases, monitoring etc.
- They remove the need to carefully plan your application needs by providing everything as a service
- Moving to microservices without the help of cloud providers is extremely difficult
- However, cost planning is impossible with cloud providers and managed services become very expensive at high scale

SERVERLESS

- Serverless is a cloud-native execution model where you write and deploy code without managing infrastructure. The platform handles provisioning, scaling, and availability automatically.
- You deploy functions, not applications.
- Code runs on-demand and scales automatically.
- Common serverless platforms: AWS Lambda, OpenFaaS, Knative

SERVERLESS VS MICROSERVICES

- Usually, a single function is not a replacement of a microservice
- A group of functions can be used to represent a microservice domain with separate logic
- Some companies are running 100% serverless allowing them to get rid of infrastructure entirely
- This may or may not be the future of microservices

RESOURCES

- Resources:
  -  What are microservices by Martin Fowler : https://www.youtube.com/watch?v=wgdBVIX9ifA
  - https://microservices.io/
  - https://martinfowler.com/microservices/
  -  What Netflix learned from microservices https://www.youtube.com/watch?v=TOM6UhCetQ0
  - https://newsletter.systemdesign.one/p/netflix-microservices
  - https://www.hys-enterprise.com/blog/why-and-how-netflix-amazon-and-uber-migrated-to-microservices-learn-from-their-experience/
  -  Amazon move from monolith to microservices https://vimeo.com/29719577
  - Netflix outliving AWS major region incident 2011 https://netflixtechblog.com/lessons-netflix-learned-from-the-aws-outage-deefe5fd0c04
  - Uber Introducing Domain-Oriented Microservice Architecture https://www.uber.com/en-FR/blog/microservice-architecture/

RESOURCES

- Microservice communication patterns https://www.cerbos.dev/blog/inter-service-communication-microservices
- Service to service communication https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/service-to-service-communication
- Avoiding synchronous communication https://harish-bhattbhatt.medium.com/avoiding-synchronous-communication-in-microservices-a-better-way-1f1780eb3357