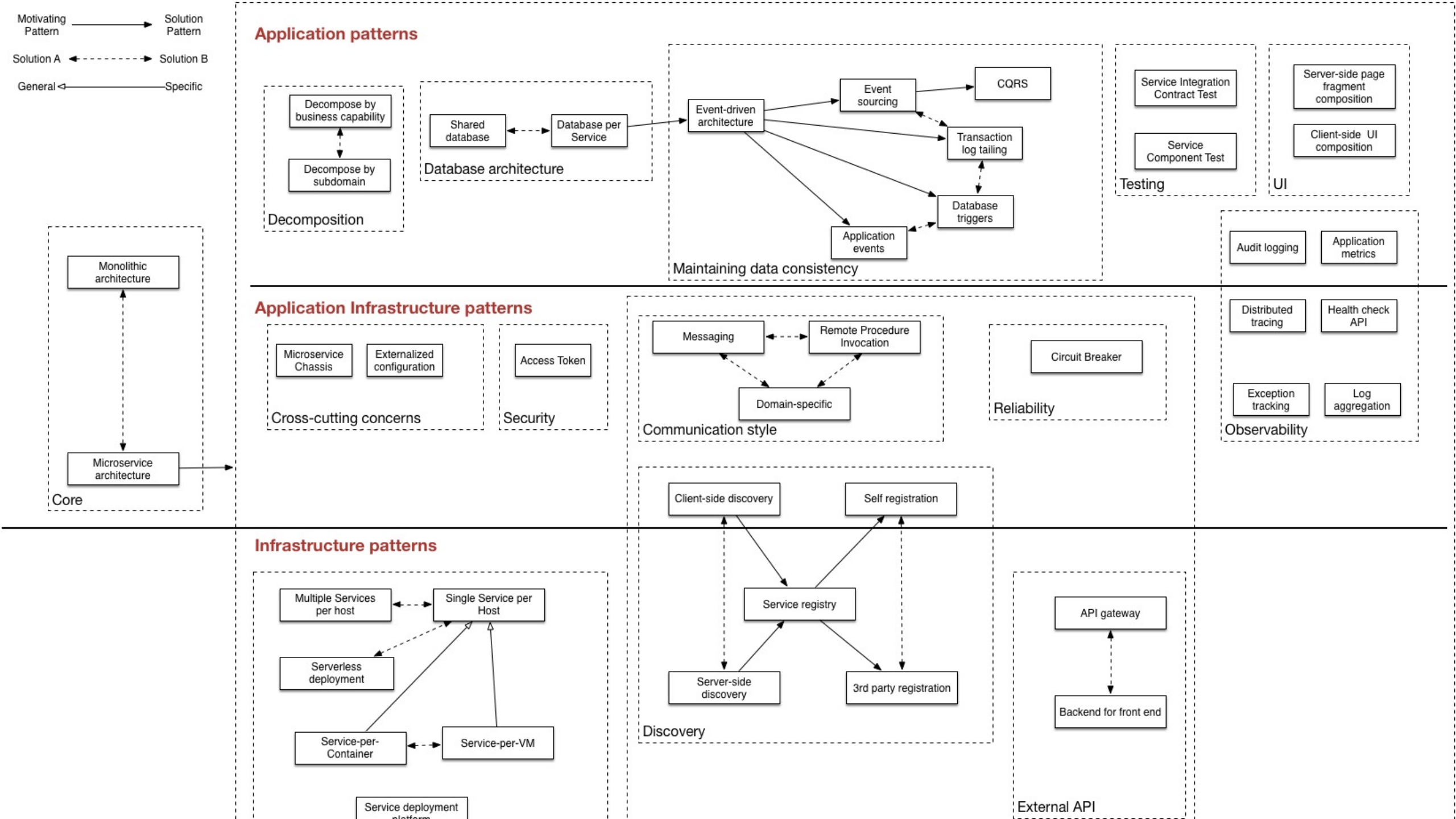
Micro Services Architecture

- for cloud native applications

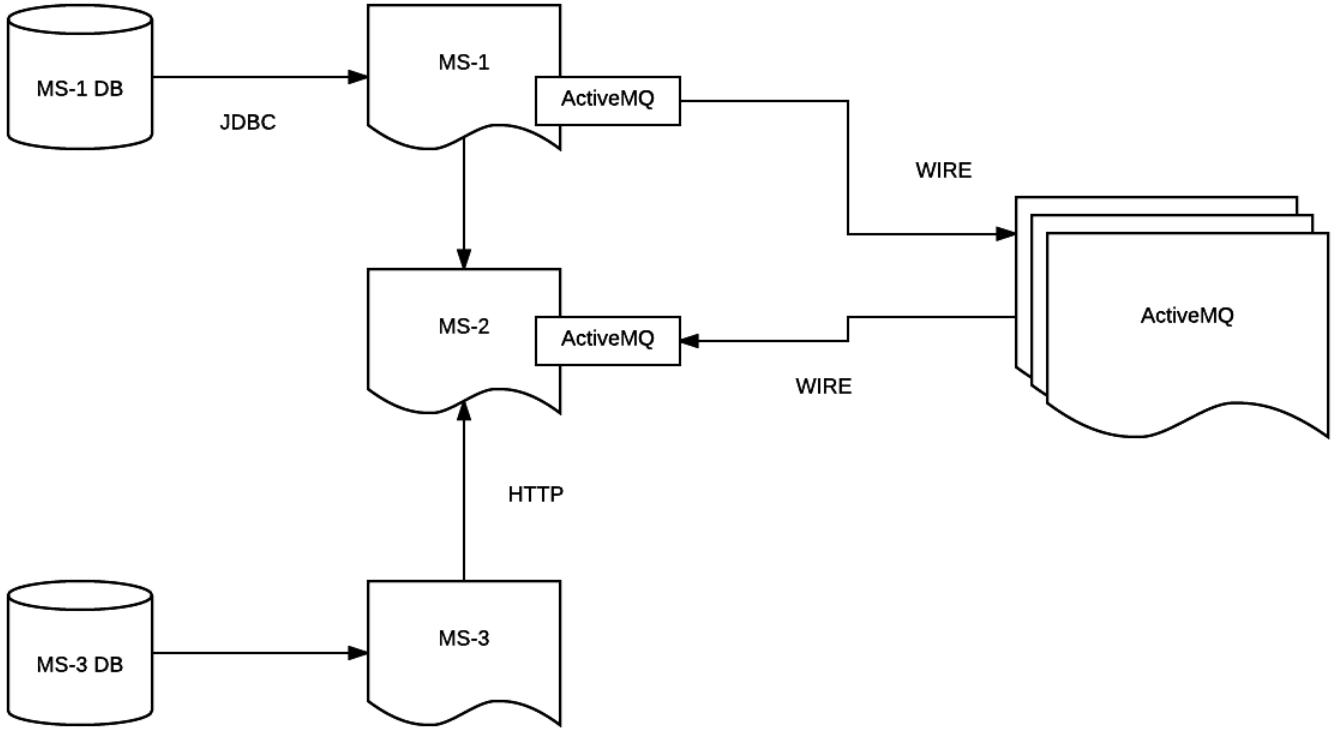
Microservice Architecture can be broken down to **Patterns**….


Motivating Pattern
Solution Pattern
Solution A
Solution B
General
Specific
Application patterns
Decompose by business capability
Decompose by subdomain
Decomposition
Shared database
Database per Service
Database architecture
Event-driven architecture
Event sourcing
CQRS
Transaction log tailing
Database triggers
Application events
Maintaining data consistency
Service Integration Contract Test
Service Component Test
Testing
Server-side page fragment composition
Client-side UI composition
UI
Audit logging
Application metrics
Monolithic architecture
Microservice architecture
Core
Application Infrastructure patterns
Microservice Chassis
Externalized configuration
Cross-cutting concerns
Access Token
Security
Messaging
Remote Procedure Invocation
Domain-specific
Communication style
Circuit Breaker
Reliability
Distributed tracing
Health check API
Exception tracking
Log aggregation
Observability
Infrastructure patterns
Multiple Services per host
Single Service per Host
Serverless deployment
Service-per-Container
Service-per-VM
Service deployment platform
Client-side discovery
Self registration
Service registry
Server-side discovery
3rd party registration
Discovery
API gateway
Backend for front end
External API

Communication Pattern

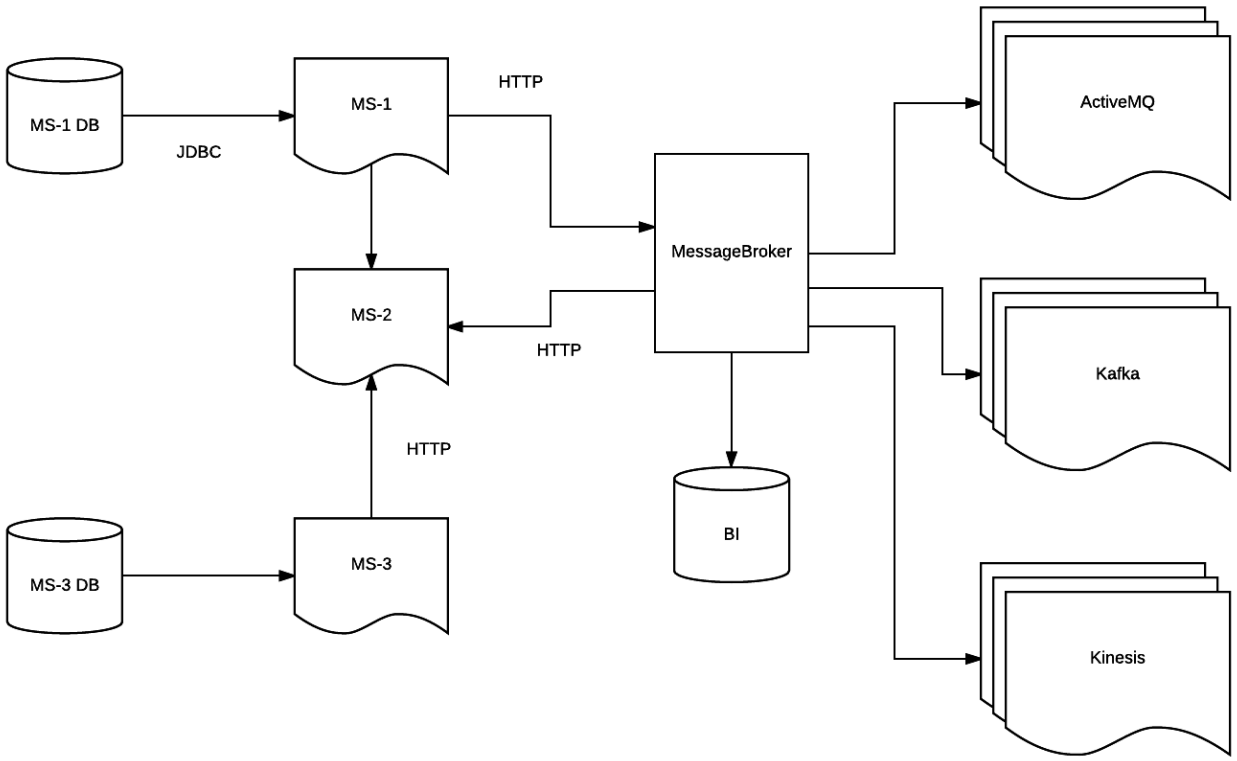
- Synchronous
  - Request – Response (user facing apps, mobile apps, portal etc, external REST API)
  - Point-to-point (service to service communication)
  - No additional moving part
  - But services are coupled. Evolution of independent service is hard
- Asynchronous
  - Good fit for Internal micro services communication
  - Services are de-coupled and can evolve independently
  - Requires a messaging broker (queue and pub/sub based)
  - Scalability (multiple consumers can listen to messages)
  - Supports Event Sourcing(OrderCreatedEvent, DisplayActivatedEvent, DisplayConnectedEvent etc…)

Current System with Broker



- Only Queues are used (Single consumer)
- Each MS is tightly coupled with Broker
- Not easy to upgrade broker
- Broker storage is not efficient
- No proper metrics
- No message tracking and tracing
- Consumer offset cannot be tracked
- Not possible to get BI for business people

Http Message Broker - Proposed



- Abstraction on Broker (few ms latency overhead)
- Queues and Topics (multiple consumers)
- Each MS is loosely coupled, HTTP Protocol
- Broker can evolve independently
- High performance storage can be plugged
- Metrics can be built in
- Message tracking and tracing can be built in
- Authentication and authorization can be built
- Consumer can be tracked, and offset can be tracked
- Backpressure support for slow consumers
- BI metrics can be created, and business is happy
- SSE, Websockets and webhooks can be used for near real time data push to subscribed consumers

Microservice Event Broker – MEB (requirements)

- Provide RESTful abstraction over queues like Kafka or AWS Kinesis or ActiveMQ
- Enable message oriented communication between services without the knowledge of underlying Queue technology
- Provide SAAS interface, so that individual team can connect/configure/monitor independently without depending on one team.
- Should provide authorization, for fine grained access control and auditing.
- Should be Stateless, so it is horizontally scalable
- Message tracking and QOS should be defined

- Subscription based event delivery, so that the offset and consumers can be tracked.
- BI Reports
  - How many SMS has been delivered
  - How many Display Inbox messages are delivered/failed/resend
  - How many DisplayNotifications are delivered/failed/resend
  - Should enable many reports
- Topics and Queues should be monitored, so that the topic owners know who are the consumers of their topic
- JSON Schema can be used for the schema evolution.
- With HTTP protocol, monitoring and load balancing are OOTB with the current Infrastructure.
- Provide client SDK for seamless integration.

The Reactive Manifesto