Fachhochschule Graubünden
University of Applied Sciences

AISE502: AI in Software Engineering II

Lecture 5: Patterns II – Microservices and Event-Driven Architecture

Script: Part II, Sections MS / EDA

Agenda

1. Recap: one quantum, three cuts – today: many quanta
2. **MS** – Microservices: what buys team independence, and what it costs
3. Sagas vs. ACID (atomic, consistent, isolated, durable) transactions: why compensation is not rollback
4. **EDA** – Event-driven architecture: what an intermediary gives and takes
5. Resilience primitives for distributed edges
6. This week's exercise: Ontologies & JSON Schema

Recap: where we are in the catalogue

Last week – three patterns, all **single-quantum**:

- **L**: technical layers – cheapest entry, pays on every change axis
- **MM**: the *domain* cut at constant quantum count – D7/D9 rise from $-$ to $+$
- **HX**: a delta discipline – $++$ on D7/D9/D12 for any host

Today: the distributed half of the catalogue. Both patterns multiply quanta, both buy their strengths with the same currency – and both carry $--$ on D8:

- **MS** distributes *by domain*: one quantum per business capability
- **EDA** decouples *in time*: an intermediary between producers and consumers

Key Concept

Keep last week's lens: every $++$ and $--$ below traces to a **tactic** the topology bundles or impedes – and to the **quantum boundaries** in the figures.

MS – Microservices

Thirty teams share one release train and step on each other with every deployment: what buys their independence – and what does it cost?

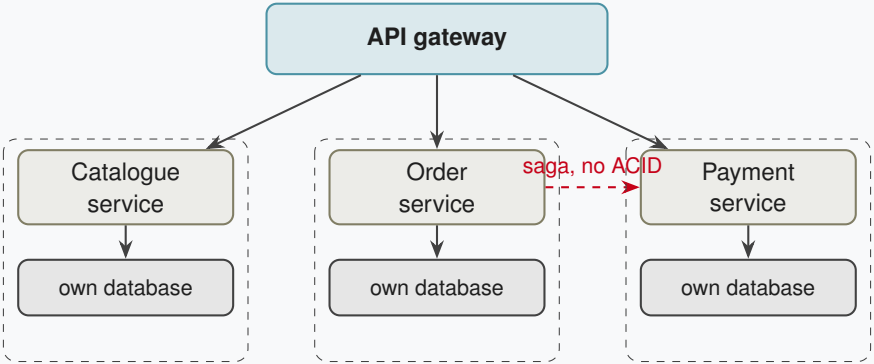
This section derives both halves of the answer – **and the cost half is the longer one.**

Definition: Microservices (MS)

A macro-structure of **independently deployable services**, each modelled around one business capability, each owning its persistent data exclusively (*database per service*), communicating via lightweight protocols. **Every service is its own architecture quantum**: its own unit of deployment, scaling, and failure.

In the taxonomy: **domain partitioning, many quanta** – the highest aggregate star score of all styles, and one star on overall cost and simplicity. Newman's one-sentence definition names the two load-bearing properties: *"independently deployable services modelled around a business domain."*

MS – topology



one quantum per service: own pipeline, own release, own blast radius, own database

Cross-service consistency is bought with **sagas instead of ACID transactions** – the structural reason for D4 $= --$.

MS – the problem it solves

Organisational before technical. By the mid-2000s, Amazon and Netflix had hundreds of teams contributing to shared deployables – and a shared deployable means a shared **release train**: every team's change waits on every other team's; integration scope grows with the *organisation*, not the change; one team's defect rolls back everyone's release. Past a certain size, *the queue for the release train* – not compute, not traffic – binds how fast the company ships.

The answer: **make the team's unit of ownership the system's unit of deployment.** Cut along business capabilities, give each service exclusive data ownership and its own pipeline – "you build it, you run it."

- strongest empirical support is organisational: in high performers, **deployment frequency scales linearly with team count** instead of collapsing under coordination (DORA, DevOps Research and Assessment)
- the documented *upper* bound: Uber, at $\sim$ 2,200 critical services, re-introduced a second structuring level – $\sim$ 70 domains with gateways (DOMA, Domain-Oriented Microservice Architecture) – a convergence back towards macro-modules

MS – capability profile (column MS)

Dimension	Rating	Structural reason
D1 Read scalability	$++$	independently replicated, cacheable read paths per service
D2 Write scal. & elasticity	$++$	per-service horizontal scaling; fine-grained provisioning
D3 Latency & predictability	$-$	network hops, serialisation, tail amplification along synchronous chains
D4 Consistency & integrity	$--$	database-per-service: no ACID across a boundary
D5 Availability & isolation	$++$	bulkheads: one failing service is not a failing product
D6 Security & auditability	$\circ$	expanded attack surface, scattered audit trails vs. isolation
D7 Evolvability	$++$	domain partitioning: a change lands inside one deployable service
D8 Simplicity & time-to-market	$--$	platform engineering before the first feature – the premium
D9 Testability & deployability	$+$	per-service excellent; system-level verification shifts towards production
D10 Operating cost	$--$	the premium is paid mostly as platform staffing
D11 Team scaling	$++$	independent deployability: teams release without queueing
D12 AI integrability	$\circ$	isolation helps, but synchronous chains multiply LLM (large language model) latency and failure probability
Native shape $S(p)$	interactive	

MS – the catalogue's steepest trade, cell by cell

- **D11 $= ++$ is the real payoff.** Deployments per developer per day scale *linearly* with team count in high-performing organisations – teams stop queueing. **This, not raw traffic, is the requirement that legitimately forces the pattern.**
- **D4 $= --$ is structural** (the method's negative example from week 3). A cross-service invariant (order $\rightarrow$ stock $\rightarrow$ ledger) becomes a saga – and *compensation is not rollback*: a rollback erases an intermediate state as if it never existed; a compensating action cannot, because **other services have already seen and acted on that state**. Every "undo" is a new business operation (cancel, restock, refund) with its own logic, tests, and failure modes. No implementation skill removes this cell – it can only be *relocated* by cutting boundaries so invariants live inside one service.
- **D8/D10 $= --$: the premium is staffing.** Self-managed Kubernetes TCO (total cost of ownership) runs roughly $3\times$ managed offerings, dominated by personnel.
- **D9 $= +$ averages a genuine split**: per-service testing is excellent; *system-level* verification shifts towards production – consumer-driven contracts, canary releases, progressive delivery.

MS – software engineering implications

Omitting these does not produce a leaner variant of the pattern – it produces a **broken** one:

Build, test, deploy

- one pipeline and one contract-test suite *per service*; **consumer-driven contracts**; canaries, **feature flags**, progressive delivery

Operations, maintenance, teams

- distributed tracing, per-service SLOs (service level objectives), **circuit breakers**, **bulkheads**
- **cross-cutting change multiplies by service count** – the Segment mechanism
- stream-aligned teams + a platform team: a *precondition*

Important Note

The distributed monolith – services that only build, test, and release together – combines the costs of both worlds without the payoff. Measurable: *lockstep release ratio*, cross-service change dispersion, synchronised version bumps.

MS – Monzo and Segment: the homogeneity condition

Example: two documented cases bracket the viability conditions

Monzo operates a licensed retail bank on $\sim$ 2,800 microservices – viable because the company enforces **extreme technological homogeneity**: one language (Go), one monorepo, shared infrastructure libraries, central migration automation that upgrades hundreds of services mechanically.

Segment cut its pipeline into $>$ 140 services – one per analytics destination, i.e. along *instances of configuration* rather than domain seams – and publicly reversed course in 2018: one shared-library change required over a hundred deployments; test and operations load crushed a small team.

Key Concept

Read jointly: viability at scale depends on **where the boundaries run** (Maxim 4) and on paying the platform premium *centrally* – not on service count.

MS – build it and study it, signals, anti-patterns

Example: Build it and study it – microservices

Build. FastAPI (Python, MIT): one small HTTP service with a generated OpenAPI contract per capability; Kubernetes underneath. Java: Spring Cloud (Apache-2.0).
Study. Google's Online Boutique (∼20.6k stars, Apache-2.0): twelve polyglot services under `src/`, gRPC contracts in `protos/`, one deployment per service. Runs on kind/minikube via `skaffold run`: moderate to hard – *instructively* so, because the platform effort you feel **is** D8 = −−. Alternative: `dotnet/eShop`. (Sock Shop, cited by older literature, was archived in 2023 – the maintenance check in action.)

Anti-patterns: *entity services* (cut around nouns – alarm: fan-out per business transaction); *grains of sand* (too small to own an invariant – alarm: service count outgrowing team count).

Choose when many teams must deliver in parallel (D11 *measurably* binding); subdomains have independent scaling/failure/compliance profiles; the organisation can staff a platform. **Avoid when** the premium exceeds the system's complexity; the domain is not yet stable enough to cut boundaries; or the motivation is fashion.

MS – AI lens and key concept

AI Lens: Microservices and AI: the synchronous-chain trap

D12 $= \circ$, and the ambivalence is precise. Per-service isolation is welcome for a fallible component – an AI capability can be its own service with its own SLO. But an LLM call inside a *synchronous* service chain multiplies seconds-scale latency and per-hop failure probability – without constitutive stability patterns this is a **cascade design**. And token cost per request is a *gateway* concern orthogonal to distribution: splitting a system into services does nothing to measure or cap it.

Key Concept

Microservices are the **only pattern rated $++$ on team scaling (D11)** – and everything else in the column is the bill: $--$ on consistency, simplicity, and cost are structural properties of many quanta, not implementation accidents. Adopt for *measured organisational scale*, never for traffic alone – traffic has cheaper answers.

EDA – Event-driven architecture

Tomorrow a new consumer needs every order event: can you attach it without touching the producer?

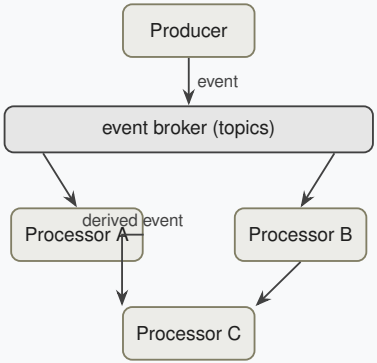
If the answer must be yes, an **intermediary** has to stand between the two – and everything in this section follows from what that intermediary gives and takes.

Definition: Event-driven architecture (EDA)

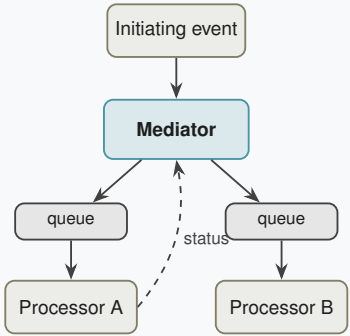
A macro-structure of **asynchronously decoupled** event producers and consumers connected through a messaging substrate. Processing is a reaction to *events* – immutable records of facts – rather than a response to synchronous calls; producers and consumers share knowledge of **event schemas, never of each other**.

Two topologies, and the distinction is load-bearing: the **broker topology** (decentralised event chains – maximal scaling, no central workflow control, hard error recovery) and the **mediator topology** (central orchestrator dispatching via queues – workflow visibility and recoverability, at the price of coupling and a potential bottleneck).

EDA – the two topologies



(a) broker topology



(b) mediator topology

Four things commonly conflated under “event-driven” (Fowler): **event notification** · **event-carried state transfer** · **event sourcing** (the log *is* the system of record – LMAX: 6M orders/s on one event-sourced JVM – Java virtual machine – thread) · **CQRS** (command query responsibility segregation: separate write and read models).

EDA – the problem it solves

Integration at scale. Around 2010, LinkedIn's activity data – page views, profile edits, connections – was wanted by an ever-growing set of consumers: search indexes, recommenders, metrics warehouses, security monitoring. Point-to-point wiring grows roughly **quadratically** with the number of systems and breaks with every schema change.

LinkedIn's answer: put **one durable, partitioned, replayable log** in the middle and let every consumer read at its own pace – the system that became **Apache Kafka**.

The general form recurs wherever the consumer set is *open-ended*: the producer of a fact – an order placed, a sensor fired, a light switched on – **cannot know today who will need that fact tomorrow**. Home Assistant faces the domestic version: thousands of device integrations, none depending on any other, coordinated by one event bus.

Example: Kafka at LinkedIn; the Uber real-time stack

By 2019, LinkedIn's Kafka carried > 7 *trillion* messages per day – the existence proof for D1/D2 $= ++$. Uber composes Kafka $+$ Flink $+$ Pinot into an end-to-end event-driven stack – the canonical EDA-plus-pipeline hybrid for the IoT (Internet of Things) class, C8.

EDA – capability profile (column EDA)

Dimension	Rating	Structural reason
D1 Read scalability	$++$	consumers replicate and cache independently behind the broker
D2 Write scal. & elasticity	$++$	partitioned ingest with elastic, decoupled consumers
D3 Latency & predictability	$+$	async throughput excellent; request/response through events is not the mode
D4 Consistency & integrity	$--$	eventual consistency moves correctness into the design
D5 Availability & isolation	$++$	a slow or dead consumer does not stall producers
D6 Security & auditability	$\circ$	durable log supports audit; causal trails need correlation IDs
D7 Evolvability	$++$	new consumers attach without touching producers
D8 Simplicity & time-to-market	$--$	eventuality and broker operations designed and staffed first
D9 Testability & deployability	$-$	non-deterministic event flows resist end-to-end testing
D10 Operating cost	$\circ$	few quanta, no premium – but the broker is a standing cost
D11 Team scaling	$+$	teams per processor decouple well; the broker needs an owner
D12 AI integrability	$++$	queues absorb LLM latency, limits, outages; log $=$ audit journal
Native shape $S(p)$	stream / async	

EDA – the same coin, two sides

- **D7 $= ++$: the decoupling is threefold.** In *topology* (producers know the event schema, never the audience), in *time* (a consumer may be down, slow, or not yet written when the event is published), and in *organisation* (the team attaching a fraud detector needs no meeting with the team owning the order flow).
- **D4 $= --$ is the same coin, seen from the other side.** Eventual consistency moves correctness *out of the database and into the design*: read-your-writes anomalies, ordering across partitions, duplicate delivery are all **design obligations** – and their testing bill is what keeps D8 at $--$ and D9 at $-$.
- **D3 $= +$ refines the five stars**: throughput and responsiveness are excellent, but synchronous round trips *through* asynchronous flows are not the pattern's mode.

Key Concept

EDA packages "use an intermediary" and "introduce concurrency" by construction – and *impedes* "transactions" by the same construction. **No broker product removes the trade.**

EDA – software engineering implications

Build and test

- the hard artefact is the **event schema**: schema registry + compatibility rules (backward/forward/full) as pipeline gates; AsyncAPI in OpenAPI's role
- contract tests per event type; consumer replays against recorded streams
- **idempotency tests**: processing an event twice must equal processing it once – under deliberate duplicate delivery

Operations and maintenance

- the RED metrics (rate, errors, duration) of this world: **consumer lag**, **dead-letter queues**, duplicate rates
- debugging = **correlation-ID tracing** across hops – a core competency, not an advanced topic
- additive evolution is the glory (attach a consumer); *changing* an established schema is the most expensive operation

EDA – build it and study it, anti-patterns

Example: Build it and study it – event-driven

Build. Apache Kafka (durable, partitioned, replayable log); RabbitMQ (classic AMQP – Advanced Message Queuing Protocol – routing). Python-first layer: FastStream (Apache-2.0) – FastAPI-style producers/consumers with the broker behind a decorator.
Study. Home Assistant (`home-assistant/core`, ∼89k stars, Apache-2.0): genuinely event-driven Python – `homeassistant/core.py` defines `Event` and `EventBus` with `async_fire`, and *every* state change flows through that one bus. Runs via `pip install homeassistant`; the codebase is huge, so **read exactly that one file** – the whole pattern is in it.

Anti-patterns with alarms: *event soup* (no workflow visibility – alarm: share of events without correlation IDs; time to reconstruct one causal trail); *events as disguised RPCs* (remote procedure calls: a producer blocking for a reply event – synchronous coupling with extra steps); *missing idempotency* (at-least-once + non-idempotent consumers = silent corruption); *event sourcing as default* (the burden of proof lies with the auditability requirement).

Discussion

Discussion

D4 $= --$, and yet Part IV will rate EDA the *primary* pattern for three application classes.

Which classes can afford to make correctness *eventual* – and what, precisely, is the **response measure** that would tell you your class cannot?

Formulate it as a six-part quality attribute scenario before looking at Part III.

EDA – AI lens and key concept

AI Lens: Queues absorb what LLMs are worst at

D12 $= ++$. The three operational pathologies of LLM components – seconds-scale latency, rate limits, provider outages – are **exactly what a queue absorbs**; asynchronous integration is the default for non-interactive AI work. Queues also create natural *batching points*: batch APIs price roughly 50 % below synchronous calls. And the durable event log **doubles as the audit journal** that logging obligations for AI systems demand – every prompt, every response, every agent step, replayable. The one thing EDA does not give: a synchronous answer – where the user is waiting, the latency budget must be engineered explicitly.

Key Concept

For your project this cell is load-bearing: the **event-driven edges** around the hexagonal modular monolith carry the AI job spine – analysis requests queue asynchronously, and the event journal is the audit trail.

Resilience primitives for distributed edges

Distribution and asynchrony are bought with **explicit failure engineering** – constitutive for MS, standard for EDA edges, and the toolkit of this week's exercise:

Primitive	What it does	Measured by
Timeout	bounds how long a caller waits – no call may wait forever	timeout rate per dependency
Retry with backoff	re-attempts transient failures with growing pauses – safe only on *idempotent* operations	retry rate; success-after-retry
Circuit breaker	callers stop calling a dependency that keeps failing; probes recovery	breaker state transitions; open time
Bulkhead	partitions resources so one failing component cannot drain the rest	saturation per pool
Fallback / degradation	a defined degraded answer beats no answer	share of degraded responses
Dead-letter queue (DLQ)	parks messages that repeatedly fail processing for inspection	DLQ depth and age
Idempotent consumer	processing twice = processing once – makes retries safe	duplicate rate under replay

Each is a **tactic** in the week-3 sense – and each becomes a *fitness function* in your measurement contract.

MS and EDA side by side

	MS	EDA	
D4 Consistency	−−	−−	both trade ACID away – differently: sagas vs. eventuality
D5 Availability	++	++	bulkheads vs. temporal decoupling
D7 Evolvability	++	++	domain cut vs. attach-a-consumer
D8 Simplicity	−−	−−	platform premium vs. asynchrony bill
D11 Team scaling	++	+	the release train vs. the broker owner
D12 AI integrability	○	++	synchronous chains vs. absorbing queues

- **MS distributes by domain**; **EDA decouples in time** – and they compose: event-driven communication *between* services is the standard hybrid
- both demand the resilience primitives – and both pay D8 $= --$ *before the first feature ships*
- for your project: neither carries the *core* (the mini-match verdict stands) – but **EDA carries the edges**

Ontology basics: a data schema for meaning

Definition: Ontology

An ontology is a shared model of a domain: its **concepts**, **properties**, relations, and constraints.

What it describes

- **Classes**: Article, Organisation, Person
- **Properties**: title, publishedAt, mentions
- **Relations**: Article *mentions* Organisation
- **Constraints**: publication date required

Why it matters for AI

- shared vocabulary across sources and prompts
- separates meaning from one input format
- supports parseability and structural correctness

Important Note

A schema does **not** ensure factual correctness, only structural validity.

Key Concept

An ontology is the **meaning model**; JSON Schema validates one representation of it.

JSON Schema basics

Definition: JSON Schema

JSON Schema is a declarative specification for the **shape and constraints** of JSON data. It can validate instances, document an interface, and guide structured LLM output.

Schema concept	Question it answers
`type`	Is the value an object, array, string, number, boolean, or null?
`properties`	Which named fields can an object contain?
`required`	Which fields must be present?
`items`	What does each element of an array look like?
`enum` / `const`	Which values, or which exact value, are allowed?
`format`	Does a string follow a known format such as a date or URI?

Key Concept

A valid JSON document is not automatically a valid **domain record**: the schema makes the expected contract explicit.

JSON Schema: the basic building blocks

Structure

- `object` groups named fields
- `array` repeats a defined item schema
- `properties` nests objects and relationships
- `required` distinguishes essential from optional data

Restrictions

- `minLength`, `minimum`, and `pattern`
- `enum` for controlled vocabularies
- `additionalProperties: false` for strict extraction
- `oneOf` / `anyOf` for alternatives

Example: A useful extraction contract

Require the article's `title`, `source`, and `publishedAt`; allow an optional array of `organisations`; restrict `sentiment` to a small, documented set of values.

The schema should be **strict enough to catch omissions** but **flexible enough** to represent legitimate variation between news outlets.

Example: a JSON Schema for a financial news article

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "FinancialNewsArticle",
  "type": "object",
  "required": ["title", "source", "publishedAt", "summary"],
  "properties": {
    "title": {"type": "string", "minLength": 1},
    "source": {"type": "string", "minLength": 1},
    "publishedAt": {"type": "string", "format": "date-time"},
    "summary": {"type": "string", "minLength": 20},
    "organisations": {"type": "array", "items": {"type": "string"}},
    "sentiment": {"enum": ["positive", "neutral", "negative"]}
  },
  "additionalProperties": false
}
```

Key Concept

This contract gives the LLM a target, the validator a test, and the benchmark a consistent unit.

Construct an ontology in four steps

1. **Identify key information**
   Extract the concepts and facts needed to answer the project's questions: article, source, date, organisations, claims, and market impact.

2. **Determine constraints**
   Decide what is required, which types and formats apply, which values are controlled, and what must be rejected as incomplete.

3. **Determine hierarchy (nesting)**
   Group information that belongs together: an article contains a publication record, entities, and possibly structured claims or events.

4. **Compose together**
   Combine concepts, relations, and constraints into one coherent ontology, then express as JSON Schema.

Key Concept

Validate the result against three different articles. If the schema cannot represent a legitimate case, the model is unfinished; if it accepts everything, the constraints are too weak.

This week's exercise: Ontologies & JSON Schema

Project Link: Portfolio Intelligence Platform

- **Identify Required Information**: From the financial news articles you scraped in last weeks exercise
- **Draft a Json Schema**: Which fields and data types are required to capture the necessary information from the financial news articles
- **Build a small Benchmark**: Manually extract the information from 3 news articles of different news outlets
- **Design a Prompt**: Come up with a system prompt for an LLM of your choice to extract the required information from the financial news articles
- **Evaluate the Prompt**: Run the prompt against the benchmark articles and assess the results
- **Document**: Document the results and how you derived the prompts

Summary

1. **MS**: the only $++$ on team scaling – adopt for *measured organisational scale*, never for traffic; D4 $= --$ is structural (compensation $\neq$ rollback), D8/D10 $= --$ is staffing
2. The **distributed monolith** is the most common failure outcome – and it is measurable (lockstep release ratio)
3. Monzo and Segment bracket the viability conditions: *where the boundaries run* and *central* platform investment – not service count
4. **EDA**: threefold decoupling (topology, time, organisation) – and D4/D8/D9 are the same coin's other side; no broker removes the trade
5. **Resilience primitives** (timeout, retry, circuit breaker, bulkhead, fallback, DLQ, idempotency) are constitutive, not hardening – each one a fitness function
6. For the project: MM+HX core unchanged; **EDA takes the edges** – queues absorb what LLMs are worst at

Next week

Lecture 6 – Patterns III and your class

- **Pipes-and-filters** and **Serverless**
- Stepping back: the quantum, partitioning beats distribution, the consolidated table
- Part III: **C10 in depth** + the C1/C2 mirror pair

Reading

- this week: Part II, MS / EDA
- ahead: Part II, PF / SL + closing; Part III, C10

Exercise

- edges, resilience, contracts; matrix pre-filter

Thank you!

Dr. Florian Herzog

Fachhochschule Graubünden, Chur

AISE502 – AI in Software Engineering II