Fachhochschule Graubünden
University of Applied Sciences

AISE502: AI in Software Engineering II

Lecture 4: Patterns I – Layered, Modular Monolith, Hexagonal

Script: Part II, Sections L / MM / HX

Agenda

1. How Part II works: examples first, generalisation after
2. **L** – Layered architecture: the simplest structure that does the whole job
3. **MM** – Modular monolith: hard boundaries inside one deployable
4. **HX** – Hexagonal architecture: dependency direction as a compiler-checked property
5. The three side by side
6. This week's exercise: architecture study I

How Part II works – and why in this order

Every pattern section follows the **same rhythm**:

1. what the pattern *is* – definition, topology, variants
2. what real-world **problem** it was invented to solve, with named production systems
3. how it behaves on the **twelve dimensions** – rating and structural reason, side by side
4. what it does to your **engineering day** – build, test, CI/CD, operations, teams
5. where you can **build it and study it** – plus **anti-patterns** with measurable alarms, and **selection signals**

Key Concept

The order is the point: software engineering is not mathematics. Its patterns were not derived from axioms – they were **abstracted from systems that worked and systems that failed expensively**. Part II follows that order of discovery; the generalisation comes at the *end*, once you have seen the cases.

Every rating is **anchored**, and **every deviation is footnoted** in the script.

L – Layered architecture / 3-tier

Two of you must ship a working product by December: what is the simplest structure that does the whole job?

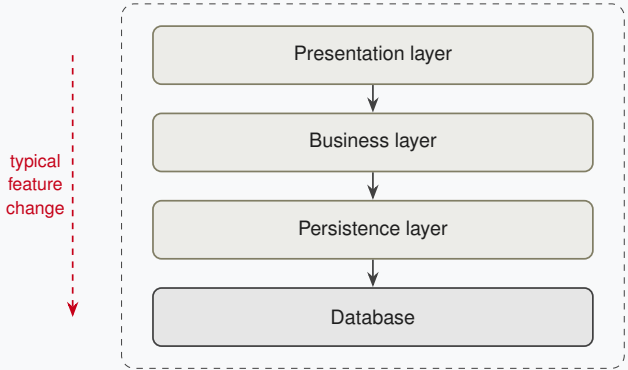
Definition: Layered architecture (L)

A macro-structure that partitions a system into **horizontal technical layers** – typically presentation, business logic, persistence, database – with a directed dependency rule: each layer may use only the layer(s) beneath it. The whole system is deployed, scaled, and fails as **one unit**: exactly one *architecture quantum*.

Two terms to keep in view – they recur in *every* pattern section:

- **technically partitioned**: units cut by technical role, not by business capability
- **architecture quantum**: one unit of deployment, scaling, and failure

L – topology



one architecture quantum: one unit of deployment, scaling, and failure

The dashed red arrow marks the structural weakness: a typical business-facing feature **cuts across every layer** – the root cause of the − on evolvability (D7).

L – the problem it solves

As old as commercial computing: a business needs a tool – capture orders, look up customers, post invoices – and it needs it *soon*. A small team, one relational database, a deadline in months. What such a project needs first is not elasticity; it is **a structure a handful of developers can hold in their heads while shipping**.

Layering was the natural first answer because it **mirrors both the technology and the team**:

- the stack already separates concerns: something renders screens, something executes rules, something persists rows – and so do the skills in the room
- one added rule – *dependencies point downwards only* – turns the habit into a pattern
- the reward: one build, one artefact, one ACID database, one thing to operate

De-facto standard since the 1990s: Spring MVC, .NET, Rails and Django scaffold it by default; Metabase ships its whole BI tier as a single JAR *deliberately*. What it was never designed to give: independent scaling and change isolation.

L – capability profile (column L of the consolidated table)

Dimension	Rating	Structural reason
D1 Read scalability	○	stateless replication + caching scale reads; the single write path remains
D2 Write scal. & elasticity	−−	every write funnels through one database inside one quantum
D3 Latency & predictability	+	in-process calls: no network hop, no tail amplification
D4 Consistency & integrity	++	one ACID transaction boundary under one process
D5 Availability & isolation	−	one process: blast radius 100 % by construction
D6 Security & auditability	+	one audit log, one small compliance scope
D7 Evolvability	−	technical partitioning: a feature cuts across every layer
D8 Simplicity & TTM	++	one repo, one pipeline – lowest entry threshold in the catalogue
D9 Testability & deployability	−	any change redeploys the whole artefact, full regression scope
D10 Operating cost	++	one cheap deployable, near-zero platform staff
D11 Team scaling	−	one quantum, one release train: a single-team pattern
D12 AI integrability	○	an AI call is easy to host – but gets no boundary, queue, or measurement point
Native shape $S(p)$	interactive	

L – three cells with a story

- **D1 $= \circ$, D2 $= --$: the read/write split.** The one-star scalability rating conflates two dimensions: replication $+$ caching demonstrably scale *reads* (Stack Overflow, Instagram) – the single write path remains the bottleneck.
- **D3 $= +$: a footnoted deviation.** In-process calls give low, *predictable* latency; the source's two stars reflect throughput, which D1/D2 capture separately.
- **D7, D9, D11 $= -$: the negative change axis.** A feature disperses across all layers; any change redeploys everything; one release train $=$ a single-team pattern.

Example: Stack Overflow – a layered monolith at planetary scale

$\sim$ 1.3 billion page views/month from $\sim$ 9 web servers before a RAM-resident SQL Server cluster plus Redis – $\sim$ 12 ms renders. A read-heavy, cache-friendly workload is exactly what carries D1: a counter-example to "scale requires microservices" *and* evidence for the single-write-path limit (D2 $= --$).

L – software engineering implications

Build and CI/CD

- one build, one pipeline, one artefact – CI trivially simple
- but release cadence is bounded by **full-regression scope**: the pipeline is cheap, its gate is expensive

Test

- in-process integration tests: fast, no network doubles
- risk: suites coupled to technical layers – refactoring breaks hundreds of tests that verify *structure*, not behaviour

Deployment and operations

- whole-system releases and rollbacks; classical APM suffices, **no distributed tracing needed** – an underrated virtue

Maintenance and teams

- without enforced conformance the pattern **decays predictably** – Lehman's second law has no structural counterweight here
- one team; sub-teams by layer reproduce the layers as hand-off friction (textbook Conway)

L – build it and study it

Example: Build it and study it – layered

Build. Django (Python, BSD-3, very active) scaffolds the pattern out of the box: model–view–template *is* an enforced persistence/logic/presentation layering. Canonical Java stack: Spring Boot MVC with Controller → Service → Repository.
Study. `healthchecks/healthchecks` (∼10k stars, BSD-3, active): a production Django monolith – four apps (`accounts`, `api`, `front`, `payments`), each cleanly layered. Runs locally on SQLite: *the easiest start in this catalogue*.

Important Note

A habit to build now: before adopting *any* system – check its **licence** and its **maintenance status** (last commit, open issues). The boxes in this part deliberately leave findings visible: a no-derivatives licence here, an archived flagship there. Verifying this is part of the engineering, and it takes two minutes.

L – anti-patterns, alarms, selection signals

Anti-patterns with measurable alarms:

- *Architecture sinkhole* – requests pass through layers that add no logic; rule of thumb: $> 80\,\%$ pass-through requests signal structure without function. **Alarm:** instrument the pass-through share; track change dispersion (D7 measure).
- *Big ball of mud* – unguarded decay. **Alarm:** rising cross-layer dependency violations; countermeasure: the dependency-rule-as-CI-gate machinery of the modular monolith.

Choose when

- team $\leq$ roughly ten; budget and schedule tight
- domain not yet understood; CRUD-dominant at modest scale
- one ACID database satisfies consistency

Avoid when

- subdomains need independent scaling
- high change rate per subdomain; fault isolation required
- several teams must release independently

L – AI lens and key concept

AI Lens: The layered pattern and AI components

D12 $=$ $\circ$ – instructive because the problem is not difficulty but **ease**: nothing stops a developer from calling an LLM synchronously from a business-layer service, *and that is precisely the risk*. No structural boundary, no queue, no measurement point; token costs are invisible to classical APM; non-determinism leaks freely across layers, which encode technology, not trust. **Hosting an AI call is easy; containing one is unsupported.**

Key Concept

The layered architecture buys the **lowest entry cost in the catalogue** ($++$ on D8, D10, an honest $++$ on D4) and pays on every axis of change and scale (D2, D5, D7, D9, D11 all negative). Not a defect – a *specific trade*: maximal day-one simplicity against minimal structural options later. Mis-chosen only when the profile weights the options it sold off.

MM – Modular monolith

You want monolith economics but fear the big ball of mud: can hard boundaries live inside one deployable?

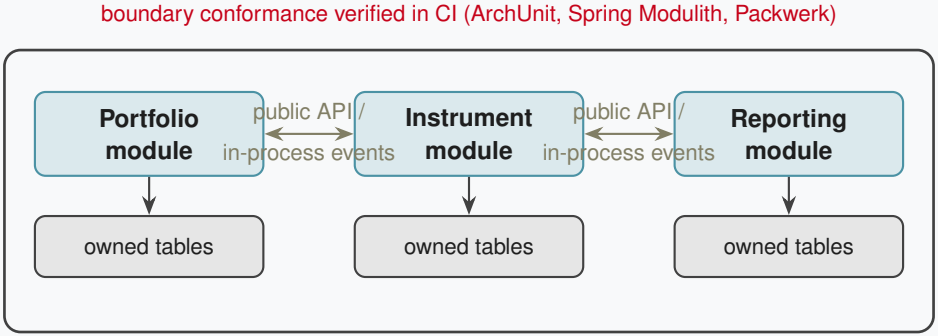
Yes – **provided the boundaries are checked by a machine, not by good intentions.**

Definition: Modular monolith (MM)

A single deployment artefact (one process, one quantum) whose interior is partitioned **by domain** into modules with enforced boundaries: each module encapsulates one *bounded context* – including sovereignty over its own persistent data – and inter-module communication passes exclusively through published module APIs or in-process events. **Boundary conformance is verified automatically.**

Bounded context (domain-driven design): a business subdomain with its own self-consistent vocabulary and model.

MM – topology



single deployable: one process, one architecture quantum

The boundaries are **not a diagram convention but a CI subject** – violations fail the build.

MM – the problem it solves

The failure mode every successful layered system eventually demonstrates: **growth without boundaries** – features smear across the layers, and the system decays into the big ball of mud.

The fashionable 2010s escape was to *distribute*: let the network enforce the boundaries. But **distribution charges rent** – platform staffing, eventual consistency, operational complexity – even when all the organisation needed was the boundaries.

Hence the precise problem: *how does a team keep monolith economics – one build, one deployable, one ACID database – and still get hard, change-absorbing domain boundaries?*

Example: Shopify – the canonical modular monolith

One of the largest Rails codebases ($>$ 2.8M lines of Ruby); decided deliberately *against* microservices. Since 2017: components with enforced boundaries, checked by the purpose-built tool **Packwerk**. Scaling: **sharding whole monolith instances into pods** – carrying BFCM peaks of $\sim$ 280M requests/minute. Both halves in one case: microservices-grade modularity (D7) at monolith-grade cost (D10) – and the mitigation that lifts D2 from $--$ to $-$.

MM – capability profile (column MM)

Dimension	Rating	Structural reason
D1 Read scalability	○	same single quantum as L: replication and caching scale reads
D2 Write scal. & elasticity	−	one database inside one quantum; pod/tenant sharding is the documented mitigation (lifts it from −−)
D3 Latency & predictability	+	in-process calls between modules: no network hop between contexts
D4 Consistency & integrity	++	one process, one ACID transaction scope across all modules
D5 Availability & isolation	−	single process, single blast radius; replication mitigates
D6 Security & auditability	+	one audit log, one compliance scope; module boundaries as policy seams
D7 Evolvability	+	the domain cut absorbs a typical feature *inside one module*
D8 Simplicity & TTM	+	monolith-simple minus one step: boundary governance is a permanent line item
D9 Testability & deployability	+	module APIs as natural test seams; module-scoped test selection
D10 Operating cost	++	one artefact, one process, classical monitoring
D11 Team scaling	○	roughly three to five teams can share one release train
D12 AI integrability	+	a hard, CI-verifiable module boundary contains the AI subsystem at monolith cost
Native shape $S(p)$	interactive	

MM – read the profile *against* L

The most instructive reading isolates the effect of the **partitioning axis** at a constant quantum count:

	L	MM	
D4 Consistency	++	++	unchanged – same single quantum
D10 Operating cost	++	++	unchanged – same single quantum
D7 Evolvability	−	+	**domain partitioning buys this**
D9 Testability	−	+	**domain partitioning buys this**
D8 Simplicity	++	+	the price: permanent boundary governance

Architecture conformance becomes a CI subject – the declared structure is executable, a failing test rather than a slide:

```
rule "module boundaries hold" {
  classes in module("portfolio") may only be accessed through its published API;
  no cycles between modules(); domain packages must not depend on framework packages;
  no module accesses tables owned by another module;
}
```

MM – software engineering implications

Build and CI/CD

- still one build – but the domain cut enables **module-scoped test selection**: a reporting change runs the reporting tests, not the world
- architecture verification runs on every build – *the first fitness function most teams ever write*

Test

- module APIs as test seams; hermetic module tests replace whole-system fixtures

Deployment and operations

- identical to the monolith, identically cheap

Maintenance and teams

- failure mode: **boundary erosion**; countermeasure: automated verification – Lehman's second law never sleeps, and code review alone demonstrably does not hold the line
- roughly three to five teams with accepted release coordination; beyond that, the single release train binds and D11 is exhausted

MM – build it and study it

Example: Build it and study it – modular monolith

Build. Spring Modulith (Java, Apache-2.0, active): module boundaries as first-class artefacts – verification tests, recorded event publication, generated module documentation inside one Spring Boot deployable. In Python the same discipline is a linter: `import-linter` (BSD-2, active) declares layer and independence contracts and fails CI on violation – *remember it, it returns unchanged for the hexagonal pattern*.
Study. Apache Fineract (∼2.3k stars, Apache-2.0, active): core banking at industrial scale – **34** `fineract-*` **Gradle modules** (`-loan`, `-accounting`, `-savings`, ...) composed into *one* deployable, plus a `custom/` extension directory. Runs via docker-compose (JVM, heavy: moderate effort). Smaller first contact: `sivaprasadreddy/spring-modular-monolith` (Apache-2.0).

Anti-patterns and alarms: *boundary erosion without automated verification* (alarms: declared-boundary violations > 0 in CI; rising share of features touching > 2 modules); *shared database tables across module boundaries* (alarms: foreign keys crossing module schemas; migrations that only pass when several modules deploy together).

MM – selection signals, AI lens, key concept

Choose when: domain known or explorable; organisation $<$ roughly fifty developers; time-to-market *and* long-term evolvability both matter (MonolithFirst). **Avoid when:** modules have fundamentally different scaling or compliance profiles, or independent deployment is business-critical.

AI Lens: The modular monolith as an AI host

D12 $= +$: hosts the deterministic majority of an AI-bearing system at monolith cost, and gives the AI subsystem what L cannot: a **hard, CI-verifiable module boundary**. Your project's determinism boundary – *no domain module talks to the LLM gateway except through its declared port* – is exactly one more rule in the boundary fitness function.

Key Concept

MM dominates L on almost every dimension except day-one simplicity, at the price of **continuous boundary governance**. The default starting point of modern systems: *domain partitioning now, distribution only when a measured requirement demands it*.

HX – Hexagonal architecture / ports and adapters

How do you make the domain logic testable in milliseconds when the database, the broker, and an LLM provider all sit at the edges?

By making “at the edges” **a property the compiler can check**.

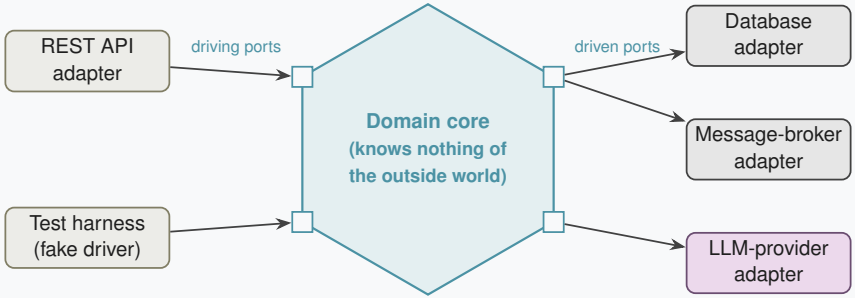
Definition: Hexagonal architecture / ports and adapters (HX)

A pattern of **dependency organisation**: the application core interacts with the outside exclusively through technology-neutral interfaces (*ports*) that *the core itself defines*; technology-specific *adapters* plug into them. **All source-code dependencies point inwards** – adapters depend on ports, never the reverse.

Important Note

Not a distribution style. Orthogonal to the monolith/microservices axis; *composes* with the other six. Its column is a **delta**: ◇ inherits the host; its own five cells are what the cut *adds*.

HX – topology



runtime calls flow outwards; source-code dependencies point inwards:
adapters depend on ports, the core depends on nothing outside itself

The **LLM-provider adapter** previews the pattern's role as the *anti-corruption layer* for AI components.

HX – the problem it solves

The pain that produced the pattern is testability. Cockburn's 2005 diagnosis: business logic ends up *welded* to its surroundings – rules leak into UI event handlers on one side and into data-access code on the other, until the only way to check a discount computation is to start the whole application, click through screens, and inspect database rows. Such tests are slow, brittle, expensive – *and the logic most worth testing is precisely the logic hardest to reach*.

The same weld causes a second, slower problem: **the system cannot follow its technology**. A new UI generation, a database migration, an additional channel – each *ought* to be peripheral, but while rules are smeared through UI and persistence code, every peripheral change is open-heart surgery.

Cockburn's response **inverts the dependency**: the core defines technology-neutral interfaces in its own vocabulary, and every technology – *the test harness included* – becomes an interchangeable plug. A fake replaces the database in milliseconds.

Standard core-isolation discipline of banking backends; house style of domain-driven design.

HX – capability profile (a delta: ◇ inherits the host)

Dimension	Rating	Structural reason
D1 Read scalability	◇	inherited from the host pattern
D2 Write scal. & elasticity	◇	inherited
D3 Latency & predictability	◇	inherited
D4 Consistency & integrity	◇	inherited
D5 Availability & isolation	◇	inherited
D6 Security & auditability	+	ports are natural audit and policy chokepoints
D7 Evolvability	++	technology migrations become **localised adapter tasks**
D8 Simplicity & TTM	−	indirection that amortises only under change – ceremony for pure CRUD
D9 Testability & deployability	++	**hermetic domain tests**: fakes replace infrastructure, feedback in milliseconds
D10 Operating cost	◇	inherited
D11 Team scaling	◇	inherited
D12 AI integrability	++	an **imposed anti-corruption boundary** that an ML component cannot erode
Native shape $S(p)$	(host's)	

HX – the delta cells, measured

- **D7 $= ++$ and D9 $= ++$ are the pattern's reason for existing** – and both are directly measurable: the share of *hermetic* tests rises sharply once domain logic sits behind ports; test feedback for that logic falls to milliseconds. Technology migrations – database, broker, LLM provider – become *localised adapter tasks*: A1's cost-of-change criterion applied surgically at the anticipated seams.
- **The honest cost is D8 $= -$**: for pure CRUD pass-through with no domain logic, ports and adapters are ceremony – waste, in lean terms.

Example: Storage migrations behind a seam – Discord

Discord's message store migrated **twice at trillion-row scale** – MongoDB $\rightarrow$ Cassandra $\rightarrow$ ScyllaDB, hot paths re-implemented as Rust data services – cutting p99 reads from 40–125 ms to $\sim$ 15 ms. Survivable *because* data access sat behind stable interfaces: **the storage technology churned, the callers did not**. The D7/D9 delta observed in production – and a preview of why the same discipline turns an LLM-provider deprecation from a crisis into an adapter task.

HX – software engineering implications

- **Build and CI/CD:** the hexagon is invisible to the pipeline topology, but adds one gate: *the dependency rule itself is a fitness function* – “the domain imports no framework, no vendor SDK” – enforceable with the same ArchUnit-class tooling as module boundaries
- **Test:** the pyramid gains a wide, fast base of hermetic domain tests; each port acquires **contract tests** that every adapter – *including every fake* – must pass, so that fakes cannot drift from reality
- **Maintenance:** the payoff channel – volatile peripheries (UIs, providers, AI services) churn in adapters while the core stays still
- **Teams:** neutral – but the pattern demands *design skill*: ports must be cut in domain vocabulary, which is a modelling task, not a refactoring

HX – build it and study it

Example: Build it and study it – hexagonal

Build. The hexagon is a discipline, not a runtime – its "framework" is a rule checker. ArchUnit (Java, Apache-2.0, active): its `onionArchitecture()` API encodes the inward-pointing arrows directly. Python: `import-linter` (BSD-2, active) enforces the same contracts between packages.
Study. `cosmicpython/code` (∼2.7k stars): the example application of *Architecture Patterns with Python* (Percival & Gregory) – the book's full text is freely readable online, the ideal companion for this module's Python-first audience. Look at `src/allocation/`: the split into `domain/`, `service_layer/`, `adapters/`, `entrypoints/`, and the repository and unit-of-work *ports* with swappable adapters. Runs via docker-compose + pytest. **Licence caveat, as promised:** the repository is **CC-BY-ND** – free to study, *not* free to reuse in derived work.

Selection signals – **choose when** long-lived domain logic meets volatile peripheries (many UIs, exchangeable providers, AI services) and testability is High; **avoid when** the component is pure CRUD pass-through – then D8 $= -$ buys nothing, and the indirection never amortises.

HX – anti-pattern: hexagonal in name only

The folder structure says ports and adapters; the dependencies say otherwise. **Three measurable symptoms:**

1. **Ports leaking vendor types** – a port signature mentions a provider SDK class. *Alarm:* static rule “no vendor or framework types in port signatures”, target zero violations.
2. **Adapters containing business logic** – decisions made where they cannot be hermetically tested. *Alarm:* the share of hermetic tests *stagnates* after the supposed hexagonal refactoring (if the cut were real, it would rise).
3. **The anaemic domain behind perfect ports** – all logic in application services, the core reduced to data bags. *Alarm:* core modules with high afferent coupling but near-zero cyclomatic complexity (almost no decision logic inside).

HX – AI lens and key concept

AI Lens: The anti-corruption layer for AI components

D12 $= ++$ – **the strongest cell in the D12 row**. ML components resist modularisation: “changing anything changes everything” (CACE), and they *erode* abstraction boundaries unless a boundary is **imposed** on them. The port is that imposed boundary – an *anti-corruption layer* in the DDD sense. An LLM behind a port is **swappable** when the provider deprecates the model, **mockable** in every test, and **replaceable by a deterministic fake** – so the deterministic 95 % of the system can be tested deterministically.

Key Concept

HX is not a competitor of the other six patterns but a **discipline inside them**. Its profile is a delta: $++$ on evolvability, testability, and AI integrability, bought with $-$ on day-one simplicity. It purchases **options on change** – and, like all options, it is worth exactly nothing where change never comes.

The three side by side

	L	MM	HX (delta)
D4 Consistency	++	++	◇
D7 Evolvability	−	+	++
D8 Simplicity & TTM	++	+	−
D9 Testability	−	+	++
D10 Operating cost	++	++	◇
D12 AI integrability	○	+	++

Three readings:

- L → MM isolates the **partitioning axis**: same quantum, same cost – evolvability and testability bought by the *domain cut*, not by distribution
- HX **composes**: a hexagonal modular monolith takes MM's column and lifts D7/D9/D12 towards ++
- exactly this composition won the mini-match for C10 last week – **your project architecture**

This week's exercise: architecture study I

Project Link: Portfolio Intelligence Platform

- **Inspect** Apache Fineract (34 modules, one deployable) and `cosmicpython/code` (`domain/`, `service_layer/`, `adapters/`, `entrypoints/`) – find the boundaries, find the ports
- **Shortlist candidates for the platform core**: which of L / MM / MM+HX carries your requirements profile from A1?
- **Draw the C4 context and container diagrams** of your platform draft

While you study the repositories, apply the two-minute habit: **licence and maintenance status first** – one of this week's study objects carries a no-derivatives licence, and finding that yourself is part of the exercise.

Summary

1. Part II works **inductively**: problem $\rightarrow$ topology $\rightarrow$ twelve-dimension profile $\rightarrow$ engineering consequences $\rightarrow$ inspectable open source
2. **L**: lowest entry cost in the catalogue ($++$ D4/D8/D10) – pays on every axis of change and scale; a specific trade, not a defect
3. **MM**: domain partitioning at constant quantum count – D7/D9 rise from $-$ to $+$ while D4/D10 stay $++$; boundary conformance is a **CI subject**
4. **HX**: a delta pattern – $++$ on D7/D9/D12 for any host, at $-$ on D8; the **anti-corruption layer** for AI components
5. Together: the **hexagonal modular monolith** – the composition that won the C10 mini-match, and your project's architecture

Next week

Lecture 5 – Patterns II

- **Microservices**: quanta, sagas vs. ACID, the operating premium
- **Event-driven architecture**: brokers, resilience patterns, eventual consistency

Reading

- this week: Part II, sections L / MM / HX
- ahead: Part II, sections MS / EDA

Exercise

- architecture study I; C4 drafts

Thank you!

Dr. Florian Herzog

Fachhochschule Graubünden, Chur

AISE502 – AI in Software Engineering II