Fachhochschule Graubünden
University of Applied Sciences

AISE502: AI in Software Engineering II

Lecture 12: The AI Dimension I – Axis A Evidence, Axis B Foundations

Script: Part V, Sections 40–41, 42.1–42.5

Agenda

1. **Two axes, one method** – Assumption A6 falls due
2. **Axis A**: two contradictory RCTs and the empirical record
3. Reconciling the divergence; the verification bottleneck (**Maxim 7**); Axis A compact
4. **Axis B**: the news-sentiment call, wired the obvious way
5. The three component types; why containment: the SE4AI classics
6. The reference architecture: **LLM gateway, queue, ontology guard**
7. The eval harness as an engineering artefact
8. This week's exercise: **AdvisorAgent** + **sub-agents behind the gateway**

Recap: where we are

- **Parts I–IV complete; Lecture 11 closed Part IV**: fitness functions – three families (dependency gates, budgets, chaos experiments); the four-layer cascade and the eight-row C10 reference contract; cost of change flat *within* / steep *across* architecture boundaries; Conway – the fit is three-way; six limits; **Maxim 9** – Part IV closed
- **AI Lens threads so far** – deck 1: the two AI axes and Assumption A6; deck 3: an LLM component stresses D3, D10, D12; an agent drafts the ADR, a human owns the decision; **ADR-011**: all LLM calls through one gateway port
- **Deck 6**: C10 profile – D12 = H, evals as the operative meaning of testability, cost per request; outlook: agent orchestration reuses the catalogue's topologies, workflows before agents (15× token finding); **deck 11**: fitness functions are the operating licence for agents (A); eval pass rate and token budget are fitness functions with old mechanics (B)
- **Today**: Part V redeems A6 systematically – the two axes as one method; the Axis A evidence and its resolution; the Axis B foundations up to the eval harness
- **Project**: M4 delivered (deterministic core fully tested and resilient); **M5 begins** – AdvisorAgent + 2–3 sub-agents behind the gateway, ontology guard active; the reference architecture today is just-in-time

Part V opens: the promissory note falls due

Four parts built a complete decision theory without ever making artificial intelligence its subject – does the construction survive the technology that defines its decade?

- Not a rhetorical flourish but a **promissory note falling due**: Part I issued it as **Assumption A6** – *AI components extend the quality attribute space but do not change the method*. The bet in two sentences: everything AI does to software engineering can be absorbed by the apparatus you now own
- If AI-bearing systems required a genuinely different method, the bet would be lost – this part is where the claim must **survive contact with the evidence**
- Roadmap – **cases first, generalisation after**: two contradictory randomised experiments open Axis A (§41); one concrete LLM call, wired wrongly and then rightly, opens Axis B (§42); the matrix reading (§43) and the emergent pattern (§44) follow next week

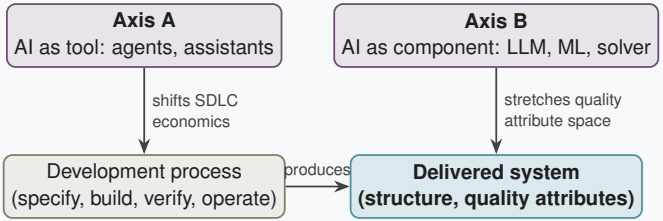
The two axes of the AI dimension (1/2): the definition

The module analyses AI along **two strictly separated axes**:

Definition: The two axes of the AI dimension

- **Axis A – AI as a tool in the SDLC.** Code assistants, agentic coding tools, review bots participate in *building* the software: they generate code, tests, documentation, draft design artefacts. The software that ships may contain no AI at all. Unit of analysis: the *development process* and its economics
- **Axis B – AI as a runtime component.** LLM services, trained ML models, optimisation solvers are *part of the delivered system* and execute in production. Unit of analysis: the *running system* and its quality attributes
- **The axes are independent**: a classical payroll system built with heavy agent support (A without B); a hand-crafted AI-native advisory platform (B without A). In practice, and in the course project, both apply simultaneously – which is precisely why they must be kept conceptually apart

The two axes of the AI dimension (2/2): attack points, one method



Axis A changes *how* systems are built; Axis B *what* the built system contains – both absorbed by the same method: scenarios, tactics, profiles, ADRs, fitness functions.

Key Concept

Two axes, one method: Axis A changes *how* systems are built, Axis B *what* they contain. The axes are independent and must be kept apart – and both are analysed with the apparatus of Parts I–IV, *nothing new*.

Case 1 – the Copilot RCT: +55.8 % on a greenfield task

AI makes developers 55.8 % faster – or 19 % slower. Which study is wrong?

Both numbers come from randomised controlled trials, both methodologically sound; few topics in software engineering carry a larger gap between headline and evidence.

- **Case 1 (published 2023)**: 95 professional developers randomly split into two groups; same task – implement an HTTP server in JavaScript; one group with GitHub Copilot, one without; the clock measured time to completion
- The treatment group finished **55.8 % faster**
- **Qualification 1**: the confidence interval (**21–89 %**) is very wide – the headline number is a point estimate, not a natural constant
- **Qualification 2**: a bounded, well-defined *greenfield* exercise – no legacy context, no architectural constraints, no review process
- **Qualification 3**: speed was measured, not quality; completion rates did not differ significantly
- Within those bounds the result is real – and it is the origin of the “AI doubles productivity” headline genre

Case 2 – the METR RCT: 19 % slower in your own mature codebase

What happens when the same technology meets experts on their own terrain?

- **16 experienced open-source maintainers**; 246 real issues in repositories they had maintained for years – large, mature codebases (over a million lines) with high implicit quality standards; each issue randomly assigned to an AI-allowed condition (predominantly Cursor with frontier models of early 2025) or an AI-forbidden condition
- With AI, the developers took **19 % longer**
- **The perception data are the didactic core**: forecast before the study $+24\,\%$ speed-up; measured $-19\,\%$; post-hoc estimate $+20\,\%$ faster – even experts cannot validly introspect their own AI-assisted productivity
- METR's own explanation *maps boundary conditions* rather than refuting Case 1: deep repository familiarity left little for AI-supplied context to add; codebases large and conventionally dense; substantial time spent checking, repairing, discarding AI proposals

The full empirical record, 2023–2025

The two cases are the extreme corners of a seven-strand record, 2023–2025 – read every row *setting first, finding second*:

Evidence	Setting	Finding
Copilot RCT	95 developers; greenfield HTTP server (JavaScript)	+55.8 % task speed (95 % CI 21–89 %); completion rate not significantly different
Three field experiments	4,867 developers; Microsoft, Accenture, Fortune-100 firm	+26.1 % completed tasks (s.e. 10.3 %); less experienced developers gain most
METR RCT	16 expert OSS maintainers; 246 real issues, own mature repositories	**19 % slower** with AI – while estimating afterwards that AI had made them 20 % faster
DORA 2024	∼3,000 respondents; organisational delivery level	+25 % AI adoption associated with −1.5 % **throughput** and −7.2 % **delivery stability**
DORA 2025	∼5,000 respondents	throughput association now positive; **instability persists**; AI acts as an *amplifier* of existing strengths and dysfunctions
GitClear longitudinal	211 million changed code lines, 2020–2024	4× growth in code duplication; moved-code share (the refactoring signature) collapsed from ∼25 % to below 10 %
Stack Overflow survey	>49,000 developers	84 % use or plan to use AI; **46 % actively distrust** its output; top frustration: “almost right” code

Caution (GitHub’s telemetry-plus-survey study): the best predictor of *perceived* productivity is the suggestion acceptance rate, not the persistence of accepted code; Case 2’s perception gap is the controlled-trial demonstration of this fact.

The system level: DORA 2024 and 2025

- DORA measures neither task times nor perceptions but **delivery performance at the level of the organisation** – throughput and stability – exactly the level at which architecture acts
- **2024** ($\sim$3,000 respondents; 75.9 % use AI for at least part of their work, roughly three quarters report productivity gains) – a 25 % increase in AI adoption is associated with **gains** of $+7.5$ % documentation quality, $+3.4$ % code quality, $+3.1$ % review speed – and **losses** of -1.5 % delivery throughput and -7.2 % delivery stability. Proposed mechanism is classical: more code per change, and larger batch sizes have been a documented risk driver for years
- **2025** ($\sim$5,000 respondents): adoption near saturation (90 %, median about two hours of daily use); more than 80 % report productivity gains; 30 % still express little or no trust in AI-generated code; the throughput association has **turned positive** as tools and practices matured – the negative association with delivery stability **persists**
- Central metaphor: **AI is an amplifier** – it magnifies the strengths of well-run organisations and the dysfunctions of badly run ones
- *Individual acceleration and system-level performance are different quantities, and only the second one pays salaries*

Code structure and practitioner trust in the longitudinal record

- **GitClear**, 211 million changed lines (2020–2024): duplicated code blocks (five or more lines) at **four times** their pre-AI level in 2024; *moved* lines – the fingerprint of refactoring and modularisation – fell from roughly **25 % to under 10 %**: 2024 was the first year in which copy-paste exceeded code movement
- **Churn** – code reworked or discarded within two weeks of commit – rose from a pre-AI baseline of roughly 3–4 % to **5.7 %** in 2024, trend continuing; two obligatory caveats: GitClear is a commercial analytics vendor, and the analysis is *correlational* – AI's causal share is plausible but not isolated
- Converges with DORA's stability data: **more code, produced faster, structurally worse maintained** – reuse by abstraction displaced by reuse by duplication, the opposite of what Parnas-style modularisation (Part II) works to achieve
- **Stack Overflow 2025** (>49,000 developers): **84 %** use or plan to use AI tools; **46 %** actively distrust the accuracy of the output; most-cited frustration (45 %): "almost right, but not quite" – *adoption rises while trust falls*, consistent with METR and DORA: the effort has migrated from writing to verifying

Reconciling the divergence: five moderator variables

So which study is wrong? Neither – resolving the contradiction *is* the lesson: different populations (task novices vs. domain experts in their own code), different codebases (greenfield vs. mature), different tasks (bounded vs. real issues) – the results never actually compete; the resolution requires reading *study designs*, not abstracts. Case 1 and Case 2 sit at opposite corners of a five-dimensional design space – and every other row of the record finds its place in the same coordinates.

Moderator	Gains high	Gains low or negative
Experience	juniors, task novices (Copilot RCT, field experiments)	domain experts in their own code (METR)
Codebase	greenfield, small, standard stack	mature, large, dense implicit conventions
Task	well-defined, bounded	under-specified, cross-cutting
Measurement	task time, perceived productivity	delivery stability, maintainability, churn (DORA 2024, GitClear)
Organisation	small batches, test automation, loose coupling	large batches, weak guardrails, tight coupling (DORA 2025)

The same technology yields $+55.8\,\%$ and $-19\,\%$ because the two cases differ on **all five rows**.

Discussion: which setting is yours?

Discussion

- The same class of technology produced $+55.8\,\%$ in one randomised experiment and $-19\,\%$ in another. Walk through the five moderators: **on which rows do the two studies differ?**
- Consider the systems you are likely to work on two years after graduation – greenfield exercises, or mature codebases with implicit conventions? **Which study's setting is closer to that reality?**
- What does the METR perception gap (forecast $+24\,\%$, measured $-19\,\%$, post-hoc estimate $+20\,\%$) imply about **relying on your own felt productivity as evidence**?

Project transfer: which moderator row describes your repository in week 12?

The verification bottleneck

The structural conclusion underneath the moderator table, in one sentence:

Code generation became cheap; specification, verification, and architecture became the binding constraints.

- When the marginal cost of producing plausible code approaches zero, the scarce resource is no longer typing but **everything that surrounds it**: understanding the requirement precisely enough to specify it, reviewing and testing what was generated, and accepting responsibility for shipping it
- **The strands converge**: DORA – individual acceleration coexists with delivery instability where control systems are weak; two thirds of surveyed developers spend more time on almost-right code; a substantial share of METR's slow-down is time spent checking, repairing, discarding AI proposals; industry analyses describe *code review as the new bottleneck* – more and larger pull requests meeting unchanged human review capacity
- Economically put: **AI lowers the cost of *producing* code, not the cost of *taking responsibility* for code**

Three consequences bind Axis A into the fit theory – Maxim 7

1. **Architecture quality gates AI gains** – DORA 2025's core finding: teams in loosely coupled architectures with fast feedback loops convert AI adoption into throughput; tightly coupled systems with slow processes do not – the AI-era echo of loosely coupled architectures and teams as the strongest predictor of continuous delivery performance (the coupling finding of Lecture 11). In the theory's vocabulary: **D7** (evolvability) and **D9** (testability and deployability) gain weight in *every* requirements profile – architecture–application fit acquires a second reading: fit to a *mode of work* in which change volume rises by an order of magnitude
2. **Architecture documentation becomes a control interface** – ADRs, repository convention files, and machine-readable rules are no longer passive records; agents execute them on every run (§41.6)
3. **Fitness functions become the operating licence for agents** – an agent iterating against a dense test suite and CI-enforced architecture rules is contained; without them, every agent change is unpriced risk (§41.7)

Key Concept

Maxim 7. Good architecture was always the art of making change cheap and safe; AI raises the change rate by an order of magnitude – and therefore **raises, not lowers, the value of architecture**.

Architecture documentation as a control interface for agents (1/2)

- Agentic tools are **context-driven**: they produce architecture-conformant code only if the architecture is *explicit, machine-readable, and in the repository* – Part I's documentation artefacts, written for human readers, upgrade into a **control interface for machine collaborators**. **ADRs** (preferably MADR) serve agents twice: as *input context* (why is the system structured this way? which options were rejected, and why?) and as *output format* – an agent drafts, a human decides and signs, per Assumption A1 (deck 3)
- **Agent instruction files** – project-local `CLAUDE.md` and the vendor-neutral `AGENTS.md` (published 2025, adopted within months by over 60,000 open-source repositories) – carry stack, conventions, build and test commands, module boundaries, no-go zones; loaded at every session start: documentation once "too expensive to maintain for human readers" now amortises because it is *executed* on every agent run. **Machine-checkable conventions** (dependency directions, naming, layering) are a failing test rather than a prose exhortation – the fitness-function discipline of Part IV
- **The corollary cuts both ways**: documentation debt is now reproduced at machine speed – a stale convention file or ADR is executed by every agent session; DORA 2025: "AI-accessible internal knowledge" and healthy data ecosystems rank among the seven capabilities that amplify AI benefits

Architecture documentation as a control interface for agents (2/2)

Excerpt from the course project's agent instruction file (`AGENTS.md`):

```
# Portfolio Intelligence Platform - agent instructions
## Architecture (binding; see docs/adr/)
- Modular monolith, module boundaries enforced by CI
  (see fitness_functions/boundaries_test.py).  Do not add
  cross-module imports; use the module's public API.
- All LLM access goes through gateway/ - never call a
  provider SDK from domain code (ADR-011).
## Verification (run before proposing changes)
- make test          # unit + module-boundary rules
- make evals         # eval harness; required for any
                     # change under prompts/ or gateway/
## No-go zones
- ledger/ :  append-only audit journal.  Propose changes
  as an ADR draft instead of editing code.
```

Every line is a control statement that an agent executes on each run – and that therefore must be kept as current as code.

Guardrails as the precondition for safe agent use

- If verification is the scarce resource, then everything that *automates* verification multiplies the value of AI tooling – and everything that leaves verification informal converts AI speed into instability
- **Test suites are the operating licence**: against a dense, fast test suite an agent can iterate – wrong code fails immediately and is repaired or discarded at machine speed; without that net every agent-generated change ships *unpriced risk* (DORA's "strong version control and test automation" amplifier pair)
- **Architectural fitness functions fence the structure**: an objective integrity assessment of an architectural characteristic is the machine-readable form of an architecture decision – dependency rules, cycle checks, module-boundary verification as CI gates were good practice before AI; with agents in the loop they are the mechanism by which an architect constrains *a collaborator who never attends design meetings*
- **The delivery pipeline becomes a defence instrument**: static analysis, SAST, dependency and secret scanning, contract tests, progressive delivery move from hygiene to necessity – the only controls that *scale with generation volume*
- **Continuity with Part IV**: nothing here is new machinery – the measurement contract already demanded executable invariants; Axis A merely adds a new class of change producer whose volume makes the contract *non-optional*

The tool landscape, soberly

- Record the landscape *as a geologist records a riverbed* – evidence of forces, not a map that stays accurate. Generation 2021–2023 (autocomplete-style assistants) suggested lines; generation 2024/2025 onwards **plans, edits multiple files, runs builds and tests, iterates on failures** – agentic loops with tool access
- **Claude Code** (Anthropic): agentic CLI tool, research preview February 2025, GA May 2025; repository-level anchor `CLAUDE.md`
- **Cursor** (Anysphere): AI-first IDE with an agent mode; the dominant tool among the METR study's experts
- **GitHub Copilot**: Copilot Workspace retired May 2025; its concepts live on in the asynchronous *Copilot coding agent* (issues to pull requests, in CI) and the synchronous IDE agent mode
- **Devin** (Cognition): "first AI software engineer" (2024); its 13.86 % SWE-bench result in March 2024 helped trigger the agent wave; acquired Windsurf July 2025 – rapid market consolidation

Examinable are the *patterns* (next frames), not the product names.

Two open standards – and the AI Lens on MCP

- **Two open standards matter more than any product, because they are architectural**
- **Model Context Protocol (MCP)** – Anthropic, November 2024: JSON-RPC; servers expose tools, resources, prompts; adopted by OpenAI, Google DeepMind, Microsoft in 2025; December 2025 to the Agentic AI Foundation under the Linux Foundation; over 10,000 public servers
- `AGENTS.md` for project-level instructions
- Vendor SDKs extract the agent loop as a library – *the bridge to Axis B*: the same building blocks that run SDLC agents also run runtime agent workflows (§44, next week)

AI Lens: MCP is ports-and-adapters at ecosystem scale

Strip the branding and MCP is a familiar shape: a technology-neutral *port* (the protocol) with swappable *adapters* (servers wrapping databases, ticket systems, browsers), letting any conforming client use any conforming tool – the role JDBC/ODBC played for databases. **The hexagonal pattern of Part II did not become obsolete in the agent era; it became an ecosystem standard.**

Benchmarks and their limits – an expiry date on this section

- **SWE-bench**: 2,294 real GitHub issues from twelve Python projects – given repository and issue text, produce a patch that passes hidden tests. Trajectory: **1.96 %** (best 2023 setup) → **13.86 %** (Devin, March 2024) → around **77–81 %** for frontier models by late 2025 on the human-validated 500-task *SWE-bench Verified* subset
- **Four qualifications keep the number honest**: (1) *contamination* – the repositories are in the training data; (2) *scope* – Python only, issues with tests only; (3) *criterion* – "tests pass" is not "maintainable, architecture-conformant"; (4) *saturation* – on the contamination-resistant SWE-bench Pro, frontier models initially scored around 23 %. Near-80 % benchmark scores next to METR's measured slow-down: **the module's canonical exercise in benchmark literacy**

Important Note

This section encodes the state of **early 2026**; product names carry an expiry date measured in months (Copilot Workspace lived roughly a year). Stable – and **examinable** – are the *patterns*: the synchronous pair-agent versus the asynchronous task-agent as interaction modes, context files and ADRs as the control interface, fitness functions as the containment mechanism. Every concrete tool claim carries its own temporal fitness function: *re-verify on every tool generation*.

Risks and responsibility (1/2): security, automation bias

- **Security** – the evidence predates the agent wave and gains relevance with volume: roughly **40 %** of 1,689 Copilot-generated programs (89 security-relevant scenarios) contained CWE top-25 vulnerabilities; a user study: participants with an AI assistant wrote **less secure code on most tasks while believing it more secure**; package hallucination (“slopsquatting”): across roughly 576,000 generations, **about a fifth** of recommended package references did not exist – names an attacker can register pre-emptively. Consequence: SAST, dependency and secret scanning, licence checks in CI are not optional; *security review capacity must scale with generation volume*
- **Automation bias**: over-trust in automated systems is a decades-old human-factors finding – Perry et al.’s participants overestimated their security, METR’s experts overestimated their speed

Risks and responsibility (2/2): skill, accountability

- **Skill formation**: a randomised study of engineers learning a new library – AI assistance reduced comprehension-test scores by roughly **17 %**; the usage pattern is the decisive moderator (conceptual questions preserved learning, wholesale delegation destroyed it); **entry-level developer positions are measurably declining**. For this module: the role being trained is the *specifier, verifier, and architect* – rebuild the competence ladder deliberately, including AI-free practice of fundamentals
- **Accountability**: legally and professionally, **the person who merges code answers for it**, regardless of what generated it; AI tools are not liability-bearing entities – treat AI output as *the contribution of an unknown third party*: mandatory review, provenance labelling, an explicit policy for permitted uses (DORA 2025: a clearly communicated AI policy *first* among the seven amplifier capabilities)
- AI may *draft* an ADR; a nameable person decides, signs, and defends it (deck 3)

Architecture is an accountability performance, not a text-production performance.

- **IP risk open but manageable**: *Doe v. GitHub* – the DMCA claim dismissed in 2024, licence-related claims continue; response: provider duplication filters and indemnification, licence scanning in CI, a documented residual risk in the governance record

Project link: Axis A governs how you build the platform

Project Link: Portfolio Intelligence Platform

Axis A governs *how* you build the Portfolio Intelligence Platform; the project applies every mechanism of this section:

(i) the repository carries an `AGENTS.md` / `CLAUDE.md` in the spirit of the listing – and you are expected to **keep it as current as code**

(ii) every architecture decision is an **ADR** – agents may draft, but a named team member signs

(iii) agent-generated changes enter the main branch **only through the CI gate**: module-boundary fitness functions, the test suite, and – for anything touching prompts or the gateway – the **eval harness** of §42.5

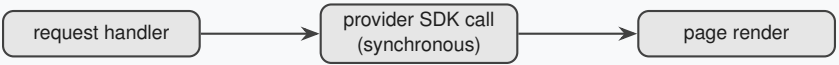
(iv) your project handbook contains a **one-page AI policy**: permitted tools, provenance labelling, review rules

The graded artefact is not the generated code – it is the control system around it.

Axis B opens: the news-sentiment call, wired the obvious way

One of the platform's features is a single LLM call – news in, sentiment out. Why not call it like any other function?

- Axis B moves AI **from the workshop into the product**; as always the case precedes the taxonomy: walk one concrete call end to end, watch what breaks, and name every break with a dimension you already own
- **The feature**: when a user opens a portfolio, the platform fetches the latest news items for its positions and asks an LLM, per item – *is this news positive, negative, or neutral for this holding, and why?* One prompt, one structured answer: the simplest runtime AI component the course project owns
- **Wired the obvious way**: a provider-SDK call inside the request handler, synchronously in the page-rendering path



Five failures arrive on schedule – each landing on one of the twelve dimensions.

Five failures on schedule (1/2): latency, cost, non-determinism

The deck-3 AI Lens (D3, D10, D12; queue, port, measurement point), now concrete.

1. **Latency (D3)**: the call takes *seconds* – one to sixty-plus, depending on model and load – where every other call in the handler takes milliseconds: the page now blocks on the slowest and least controllable component in the stack
2. **Cost (D10)**: priced per token, so the feature bills per *request* – every portfolio open costs real money; a loop over twenty positions is a twenty-fold cost regression the way an $n+1$ query is a latency regression. No classical component in the platform has this property
3. **Non-determinism (D12)**: run the same article twice and the answers differ; sometimes an answer is garbage – a score for a company not in the portfolio, a negative headline read as positive

Wired synchronously, the component has **none of the three things Part I said such a component needs**: no *queue* to absorb its latency and outages, no *port* behind which a test can substitute a deterministic fake, no *measurement point* where the cost and quality of every call are observable.

Five failures on schedule (2/2): drift, injection, diagnosis

4. **Drift (D7)**: the provider ships a new model version or deprecates the old one – GA models carry deprecation windows of the order of *six months* – and the component's behaviour changes **without any local action**: no commit, no deployment, no reviewable diff. The feature's behaviour is now co-owned by a third party
5. **Injection (D6)**: the news article is untrusted input read by a component that cannot reliably separate instructions from data – a crafted "article" can carry instructions to the model; the feature has quietly opened an attack surface that no classical threat model in the platform covers (the attack surface in depth: §42.6, next week)

Nothing on this list is a bug in the model, and nothing on it is fixed by a better prompt – every failure is a property of the *wiring*: a non-deterministic, fallible, latency-heavy, per-call-priced component was integrated as if it were deterministic, reliable, fast, and free.

The rest of the section generalises: the component taxonomy → why containment, not mere integration (SE4AI classics) → the reference architecture that re-wires the call correctly → the test instrument for a component without exact assertions (eval harness).

The three component types – one species, three profiles

- The sentiment call is **one instance of a species**: for the first time, production systems contain building blocks that are non-deterministic, fallible, latency-heavy, priced per call, and capable of changing behaviour without any local action (model updates, data drift, provider deprecation)
- **Thesis, prepared by Assumption A6**: such components change no principle of software engineering but *shift the weights in the quality attribute space* – and thereby the pattern choice; loose coupling, asynchronous integration, explicit contracts, and observability migrate from “nice to have” to **mandatory**
- Industry speaks of **compound AI systems** (deck 6, C10) for this reason: state-of-the-art results come from systems of models, retrievers, validators, and deterministic services, not a single model call

Definition: AI runtime component

A component of the delivered system whose output is produced by a *learned or search-based model* rather than by explicitly programmed logic. Three types with systematically different profiles: **(a)** LLM components for analysis, extraction, and generation over unstructured input; **(b)** classical ML components for classification and regression; **(c)** optimisation components (LP/MIP and constraint solvers, metaheuristics). The types differ exactly on the dimensions this theory measures – determinism, latency, cost model, dominant risk, explainability – and therefore demand **different integration forms**.

The three AI component types and their profiles

Dimension	(a) LLM analysis / generation	(b) ML classification / regression	(c) Optimisation (LP/MIP/CP)
Determinism	non-deterministic (even at $T = 0$ only "mostly")	deterministic after training	reproducible at fixed seed/threads/limit; variance in practice
Latency	seconds (1–60+)	milliseconds possible	seconds to hours; anytime behaviour
Cost model	per token/call (operating expenditure)	training expensive, inference cheap	compute + solver licence
Dominant risk	hallucination, prompt injection, provider drift/deprecation	data/concept drift, training/serving skew	modelling errors, runtime explosion
Explainability	low (generated justifications are themselves model output)	medium (feature importance)	**high – provable**: optimality gap, duals, IIS
Integration form	gateway + async + cache	serving endpoint + MLOps pipeline	job queue / batch worker

Each column implies a **different integration form** – which is why "add AI" is never a single architectural decision.

Type (a): LLM components – RAG, prompts, structured outputs

- LLM components turn unstructured input – documents, e-mails, reports – into analyses, extractions, or generated text; **three engineering building blocks** define the type
- **Retrieval-augmented generation (RAG)**: knowledge is moved out of the model weights into a swappable, versionable, inspectable *data component* – updated by re-indexing rather than retraining, with provenance through citable sources
- RAG is an *engineering* problem, not a model problem: case-study evidence documents seven recurring failure points (missing content, failed ranking, extraction and formatting errors, incomplete answers) – with the sobering observation that RAG robustness *evolves* in operation rather than being designed in
- **Prompts are configuration artefacts**: version-controlled, regression-tested, behaviour-determining like code – exactly the configuration-debt territory Sculley et al. mapped
- **Structured outputs**: since 2024 provider APIs can enforce, via constrained decoding, that outputs conform to a developer-supplied JSON schema – *syntactic* correctness guaranteed; *semantic* correctness remains to be verified (reference architecture, eval harness)
- **Lifecycle risk is the provider**: GA models carry deprecation windows of the order of six months, shorter windows observed – a hard-coded model name is a *ticking dependency*: an architectural statement, not an operational one

Type (b): classical ML components – perishable models

- **(b)** Self-trained models (scoring, churn, fraud, forecasting) bring the full **nine-stage workflow** – model requirements and data collection through training, evaluation, deployment, monitoring – with dense feedback loops; characteristic problems: *training/serving skew* (divergent data preparation, one of the most frequent production failure sources) and *data/concept drift* (sudden, gradual, incremental, recurring)
- **(b) A deployed model is a perishable good** – monitoring and retraining are operating requirements, not options; tooling: feature stores (consistent online/offline views), model registries (model, data, code, configuration versioned together), the MLOps discipline (maturity ladder: §43, next week)

Type (c): optimisation components – heavy solvers

- **(c)** Routinely overlooked in the SE4AI literature but belongs in every advisory platform: LP/MIP solvers, constraint programming (CP-SAT: recent MiniZinc Challenges dominated, gold-medal sweep 2024), stochastic metaheuristics – the profile *inverts* the LLM's: **deterministic but heavy**. Reproducible at fixed seed, thread count, time limit (run-to-run variability in practice); runtimes seconds to hours, often anytime behaviour → **asynchronous integration**: job queue, status polling, callback – never a synchronous call in a web request path
- **(c)** Compensating strength: **provable explainability** – optimality gap, dual values and shadow prices, on infeasibility an irreducible infeasible subset (IIS) as the explanation: a minimal set of contradictory constraints. In regulated domains the load-bearing argument for the project's division of labour: *hard, auditable decisions belong to the solver and the deterministic services, not to the LLM*

Key concept: adding AI is a per-component matching problem

The three types differ exactly where the twelve dimensions measure: **determinism (D4)** · **latency (D3)** · **cost (D10)** · **auditability (D6)** · **testability (D9)**

Key Concept

"We are adding AI" is therefore **never one decision** – it is a *per-component matching problem*, answered with the same profile logic as everything else in this module. One rule spans all three types: **contain the component behind an explicit boundary; never scatter it through the domain.**

Project transfer: the sentiment call and the AdvisorAgent's insights are type (a); the Optimization service is type (c); type (b) has no instance in the project.

Why containment: the SE4AI classics – hidden debt and CACE

Two foundational results explain why AI components need architectural **containment** rather than mere integration.

- **Sculley et al. (2015)** transferred the technical-debt metaphor to ML systems. **Observation 1**: *only a small fraction of a real-world ML system is ML code* – the famous figure shows the model as a small black box amid large blocks of configuration, data collection, feature extraction, data verification, serving infrastructure, and monitoring. The system around the model is the actual engineering task – precisely this module's perspective
- **Observation 2**: ML components resist modularisation:

Definition: CACE – Changing Anything Changes Everything

ML models *entangle* their input signals – no feature is ever truly independent, so a change to one feature distribution, hyperparameter, or upstream data source changes the behaviour of the whole model. Architectural consequence: **boundary erosion** – the strong abstraction boundaries on which modular design relies are systematically undermined by ML components.

Alongside the paper's system anti-patterns: glue code, pipeline jungles, dead experimental code paths, configuration debt, hidden feedback loops, undeclared consumers of model outputs.

Three differences, 28 tests – AI Lens: Parnas meets CACE

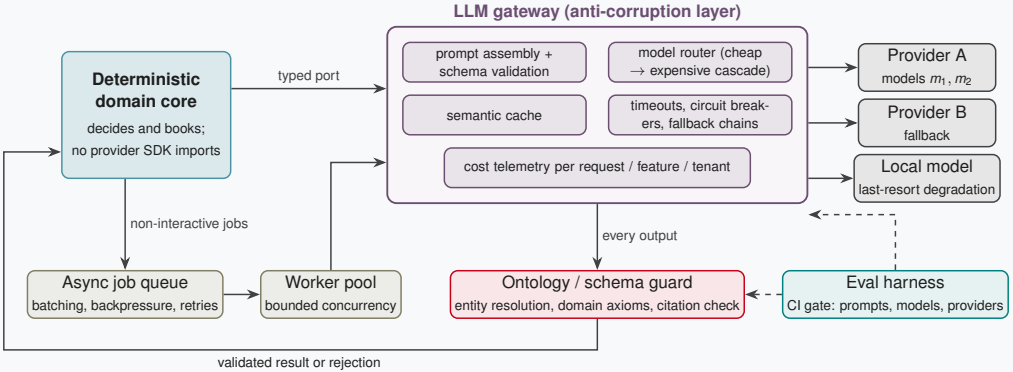
- **Amershi et al. (Microsoft product teams)** – three fundamental differences from classical development: (1) *data* discovery, versioning, labelling, schema management: harder than and qualitatively different from code management, no Git-equivalent of comparable maturity; (2) model customisation and reuse demand combined SE and ML competence; (3) **AI components are harder to modularise than software modules** – entangled (CACE), non-monotonic error behaviour, poorly predictable model interactions
- Operational counterpart – the **ML Test Score**: a rubric of **28 concrete tests** and monitoring requirements across data, model development, infrastructure, and monitoring, from Google production experience: production readiness made measurable – a checklist for the course project

AI Lens: Parnas meets CACE

Maxim 4 (Part II): domain-oriented partitioning around anticipated change is the strongest single predictor of evolvability. CACE identifies a component class in which change anticipation fails *inside* the component – everything co-varies with everything. The resolution is not to abandon Parnas but to apply him one level up: if the component cannot be decomposed, the decomposition happens *around* it – **the module boundary goes where the entanglement stops, at the component's contract**. That is the entire intellectual content of the gateway pattern – and it is sixty-year-old advice.

The reference architecture – the sentiment call, re-wired

Not new machinery but old machinery applied more strictly – the correct re-wiring of the sentiment call: behind a **typed port** into the gateway (curing drift, containing injection); non-interactive volume onto the **queue** (curing latency, buying batch pricing); every call across **one measurement point** (cost and quality observable).



This is the topology your AdvisorAgent and sub-agents are wired into this week – ADR-011 (deck 3) made structural.

The elements justified (1/4): gateway, deterministic core

- **Anti-corruption layer** / **LLM gateway**: from domain-driven design – a translation layer that prevents a foreign system's model from corrupting one's own. Applied to LLMs: *no domain code speaks to a provider API*
- A **facade** owns the provider SDKs, prompt construction, schema validation, retry logic, model selection, and cost telemetry; the domain sees only a typed interface: `analyse_report(document) -> RiskAssessment`
- Provider deprecation becomes an *adapter task* instead of a crisis; the facade is mockable in every test; as an industry pattern the gateway has consolidated into its own infrastructure layer – the AI counterpart of the API gateway. In hexagonal terms the LLM is an **adapter on a port** – the strongest single reason HX gains weight in the AI era
- **Deterministic core, probabilistic edge**: everything deterministically computable – validation, aggregation, key-figure computation, authorisation, persistence, booking – stays deterministic code; the LLM handles only what determinism cannot (language understanding, extraction from unstructured text, formulation). Keep the non-deterministic core *as small as possible* and push it to the edge

LLM agents propose; deterministic services decide and book.

The elements justified (2/4): queue, semantic cache

- **Asynchronous integration**: seconds-scale latency, rate limits, and outage risk put AI calls behind a queue wherever the domain allows – the caller enqueues a job, a worker pool calls the model at a controlled degree of parallelism, results return by event or callback
- The queue buys **backpressure** instead of overload, **retries** without blocking users, **smoothing** of rate limits, and natural **batching points**: provider batch APIs process non-urgent volume at roughly 50 % discount within processing windows up to 24 hours (figure from deck 6 – here placed where it lives: in the gateway)
- Axis B's direct coupling to **EDA and PF** (Part II) – exactly the mechanisms their D12 rows priced at ++
- **Semantic caching**: instead of exact-match keys, requests are compared by embedding similarity, so semantically equivalent queries hit the cache
- The engineering point not to miss: a false-positive cache hit is a **correctness risk**, not a performance blemish – the similarity threshold is a quality/cost regulator and *belongs in the eval harness*, not in a config file nobody reviews

The elements justified (3/4): model routing – AI Lens

- **Model routing**: model choice per request is one of the largest cost levers in the stack – cascades that start with the cheapest model and escalate only on insufficient answer quality report up to **98 % cost reduction** at comparable quality; learned routers trained on human preference data cut cost by **more than a factor of two** without quality loss, generalising to unseen model pairs (figures from deck 6 – here placed where they live: in the gateway)

AI Lens: Model routing is a classical tactic in new clothes

Part I defined **tactics** as the atomic units of architectural design. Routing traffic across a cheap and an expensive resource depending on demand is the ancient *resource-arbitration* tactic – the FrugalGPT cascade is its token-economics incarnation. Note where it lives in the figure: **in the gateway, as infrastructure, invisible to domain logic**. A tactic that leaks into the domain layer stops being a tactic and starts being coupling.

The elements justified (4/4): stability, ontology as contract

- **Stability patterns** transfer directly from the classical catalogue: *timeouts* (an LLM call without one blocks a thread for minutes); *retries with exponential backoff* – only for idempotent calls and with cost awareness, since every retry burns tokens; *circuit breakers* per provider and model; *fallback chains* – alternative model → alternative provider → cached or rule-based answer → honest degradation ("analysis currently unavailable"); *bulkheads* separating interactive from batch quotas
- Only the failure semantics are new: a **semantically unusable** answer – schema violation, suspected hallucination – must trigger the error path exactly like an HTTP 500
- **Ontology and schema as contract** – the most effective systematic hallucination defence is layered: (1) **structured outputs** enforce syntax; (2) every extracted **entity** (account number, ISIN, customer name, key figure) is resolved against the deterministic data store – unresolvable references are *rejected*, not passed on; (3) **domain axioms** hold as invariants – sums add up, weights lie in $[0, 1]$, cited passages exist in the source document; (4) **grounding** via RAG makes citations mandatory
- The schema becomes a *contract* in the design-by-contract sense, and the gateway is the contract checker – this is the **ontology guard** your project activates on all insights this week

The eval harness – definition and course thesis

Non-determinism breaks the classical test idiom: `assert expected == actual` presupposes that equal inputs produce equal outputs. When that falls, correctness must be redefined *statistically* – "correct in at least 95 % of the evaluation cases" – carried by a test artefact of the first rank.

Definition: Eval harness

A **versioned suite of test cases, scoring logic, and statistical thresholds** for a non-deterministic component, executed in the CI/CD pipeline like a test suite. It gates every prompt change, model update, and provider migration. Its thresholds are the *response measures* of the AI-related quality attribute scenarios (Assumption A4), and its pass rate is a *fitness function* in the measurement contract of Part IV.

Key Concept

The course thesis on testing AI. The eval harness is to AI components what the test pyramid is to deterministic code – the artefact that converts "it seems to work" into a *falsifiable, continuously executed claim*. Without it, every model migration is a blind flight – and with deprecation windows of months, migrations are not hypothetical. **Statistical acceptance replaces exact assertion; the thresholds are architecture decisions and belong in the measurement contract.**

Four complementary evaluation strategies make a complete harness

1. **Regression against labelled references**: a curated *golden set* of input/expectation pairs from the domain, scored with task-appropriate metrics (exact match or F1 on extracted fields, rubric scores for generated text); every prompt change, model update, and migration runs against this suite – the direct counterpart of the regression test
2. **LLM-as-judge**: strong LLM judges agree with human preference judgements in over 80 % of cases – the level of human–human agreement – a scalable scoring instrument; biases to control for: position, verbosity, self-enhancement, weak reasoning grading. Conclusion: the judge is a measurement instrument that must itself be calibrated against human labels – *the judge needs its own eval*
3. **Domain axioms and property-based testing**: instead of exact expected values, the harness checks *properties* that must hold for all valid outputs – schema validity, referential integrity against the ontology, metamorphic relations (a paraphrased input must yield a semantically equivalent output), domain monotonicities; axioms catch failure classes that no finite golden set covers
4. **Online evaluation**: sampled human review, user feedback signals, drift monitoring of the eval metrics in production – the LLM counterpart of model monitoring in the ML workflow

Example: an eval harness for the portfolio platform

Thresholds = response measures: mean F1 ≥ 0.92 on the golden set · zero axiom violations · judge–human agreement $\kappa \geq 0.70$ (Cohen's chance-corrected measure) – changing any of them is an architecture decision requiring an ADR; note what is *absent*: no assertion demands an exact output string.

```
GOLDEN = load_cases("evals/portfolio_extraction_v3.jsonl")
def test_extraction_regression(gateway):
    scores = [f1(gateway.extract(c.report), c.expected) for c in GOLDEN]
    assert mean(scores) >= 0.92   # statistical threshold
def test_domain_axioms(gateway, ontology):
    answer = gateway.advise(sample_portfolio())
    for pos in answer.positions:
        assert ontology.resolves(pos.isin), f"unknown:  {pos.isin}"
    total = sum(p.weight for p in answer.positions)
    assert abs(total - 1.0) < 1e-6
    for cit in answer.citations:
        assert cit.passage in source_text(cit.doc_id)
def test_judge_is_calibrated(judge, human_labels):
    agreement = cohens_kappa(judge.score(GOLDEN), human_labels)
    assert agreement >= 0.70   # the judge's own eval
```

Runs in CI on every change to prompts, models, or the gateway, alongside the deterministic test suite.

This week's exercise: AdvisorAgent + sub-agents behind the gateway

Project Link: Portfolio Intelligence Platform

Coaching slot (1 lesson). Milestone M5 – Multi-Agent Orchestration, Evaluation, and Hardening (weeks 12–13):

1. **Mandatory this week**: the AdvisorAgent orchestrates 2–3 sub-agents *through contracts* – every LLM call through the gateway port (ADR-011); no provider SDK import in domain code
2. **Ontology guard active on all insights**: entity resolution against the deterministic store (valid ticker, sector, event type), domain axioms, citation check – unresolvable references are *rejected*, not passed on
3. **LLM agents propose; deterministic services decide and book** – keep the deterministic core free of LLM calls *(the line that is graded)*

Axis A discipline as on the project-link slide; week 13 turns the harness into a CI gate.

The reference architecture of today is the topology you are wiring: typed port → gateway → queue → guard.

Summary

1. **A6 falls due – two axes, one method**: Axis A changes *how* systems are built, Axis B *what* they contain; independent, kept apart, analysed with the apparatus of Parts I–IV
2. **Copilot** $+55.8\,\%$ **vs. METR** $-19\,\%$: neither wrong – five moderators reconcile the record; the perception gap ($+24$ / -19 / $+20$) is the didactic core
3. **System level**: DORA – throughput positive, instability persists, AI is an amplifier; GitClear – duplication $4\times$, refactoring signature collapsed; adoption up, trust down
4. **Verification bottleneck, Maxim 7**: generation cheap, specification/verification/architecture binding; D7 and D9 gain weight in every profile
5. **Axis A compact**: documentation as control interface, fitness functions as operating licence, who merges answers; patterns examinable, products expire; MCP $=$ ports-and-adapters
6. **Axis B**: the sentiment call fails on D3, D10, D12, D7, D6 – properties of the wiring; three types $=$ a per-component matching problem; CACE $\rightarrow$ decompose *around* the component
7. **Reference architecture**: typed port $\rightarrow$ gateway (routing, cache, stability, cost telemetry) $\rightarrow$ queue $\rightarrow$ ontology guard; eval harness $=$ statistical acceptance, thresholds in the measurement contract

Next week

Lecture 13 – Part V closes: security and law, the matrix shift, the eighth pattern, synthesis

- OWASP LLM Top 10 and prompt injection; the EU AI Act as hard constraint
- how AI shifts the matrix: the C10 row cell by cell, D12 across the seven patterns, which cells shift, MLOps maturity
- agent orchestration as the emergent eighth pattern: topologies, the economics of autonomy, capability-profile sketch
- synthesis: one theory, five parts; exam orientation

Reading

- this week: Part V, sections 40–41, 42.1–42.5
- ahead: Part V, sections 42.6–42.7, 43–45

Exercise / deliverable

- coaching; eval harness as CI gate; cost/latency observability; hardening; distinction work
- **milestone: eval harness in CI + guard + cost observability**

Thank you!

Dr. Florian Herzog

Fachhochschule Graubünden, Chur

AISE502 – AI in Software Engineering II