From Layers to Agents:
Rethinking Software Architecture for the AI Era

Enrico Zimuel

unibz
Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

SDU
Syddansk Universitet

GraceX

About me

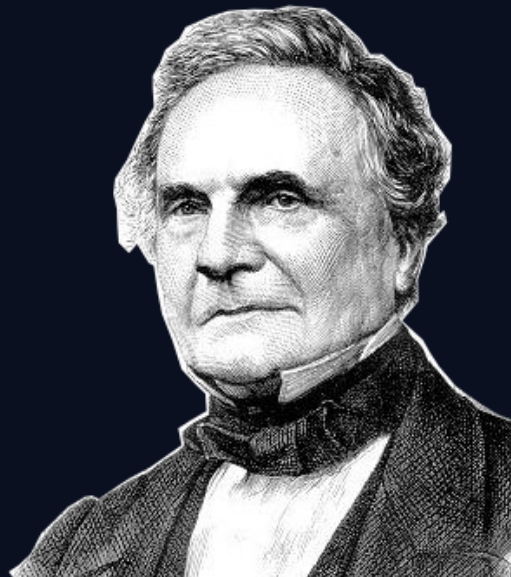
- Developer since 1996
- Adj. Prof. of Machine Learning at University of Turin
- Lecturer of Agentic AI at University of Roma Tre
- VP of Engineering & Tech Lead at GraceX
- 15+ years in Silicon Valley companies (Elastic, Zend)
- Research Programmer at Amsterdam University
- TEDx and international speaker in 140+ conferences
- Author of books and articles
- More info: www.zimuel.it

What is this?

Accademia delle Scienze (Torino), Photo by Enrico Zimuel

The Analytical Engine of Charles Babbage

- **Charles Babbage** (1791 - 1871) often called as the “father of computer” designed the **Analytical Engine** in 1837, a mechanical machines that contained the basic ideas used in modern computers
- Babbage explicitly drew on the punched-card mechanism of the **Jacquard loom** (1804)
- He never completed the Analytical Engine



The “father” of software architecture?

- **Computation was not the breakthrough. Control was.**

- The remarkable idea was that the machine would not have one fixed behavior
- Its behavior would be determined by a sequence of instructions outside the machine (punch cards)

- **Separate control by its execution**

190 years later

- **1837**
  - **Machine → Program → Machine → Program → …**
- **2026**
  - **Agent → Environment → Agent → Environment → …**

- Almost two centuries later, we are facing another control-flow transition

Cycle / Loop in Note G (1843)

- **Ada Lovelace** wrote a program to calculate Bernoulli numbers for the Analytical Engine (Note G)
- She wrote also in a note of the algorithm:
  - ***“The engine might compose elaborate and scientific pieces of music of any degree of complexity or extent”***
  - ***“The Analytical Engine has no pretensions whatever to originate anything”***

AI today

- 1843
  - Machines might compose music, but can they originate anything?
- 2026
  - Almost two centuries later, we are still arguing about that question

So, what does this have to do with software architecture?

Why this talk now?

Agentic AI is changing the unit of software work

- Generative AI moved from autocomplete to autonomous task execution
- Developers increasingly delegate implementation loops to tools
- The bottleneck shifts from:
  “Can we code it?”
  to:
  “Did we specify it well?”

The future skill is not less engineering, it is more explicit engineering

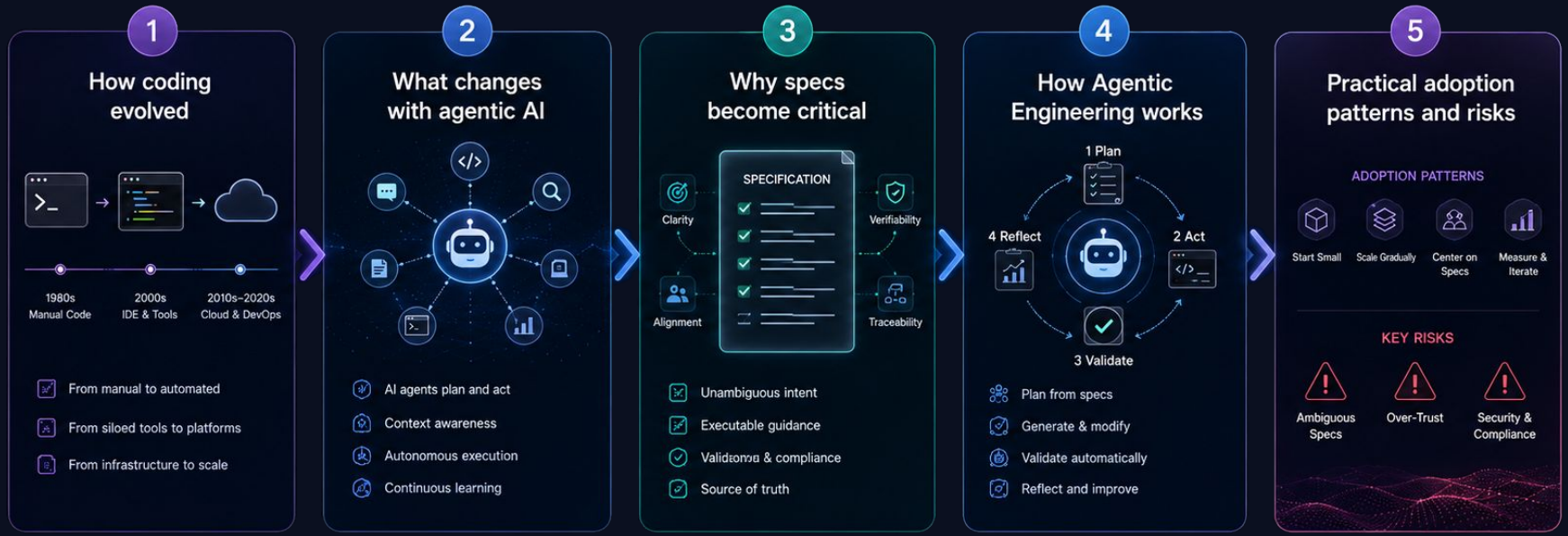
Talk thesis

- AI-assisted coding improves speed at the line and function level
- Agentic engineering changes the workflow, feedback loops, and accountability model
- Spec-Driven Development provides the control surface for agentic systems

Specs become the interface between human intent and AI execution

Agenda

What we'll cover today



01

How coding evolved

From machine instructions to reusable abstractions to AI-mediated development

A short history of abstraction

Every coding era raised the level of intent

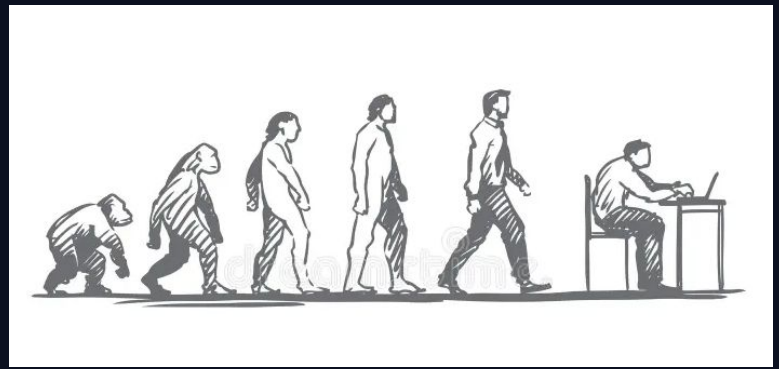
- **Assembly**: tell the machine exactly what to do
- **Compilers:** use an high level language to be translated in assembly
- **Languages and frameworks**: express patterns at higher levels
- **Cloud and DevOps:** define systems, environments, and pipelines
- **AI:** describe outcomes and constraints, then supervise execution

Software development has always been an abstraction story

The developer role has kept moving upward

The work shifts as tooling improves

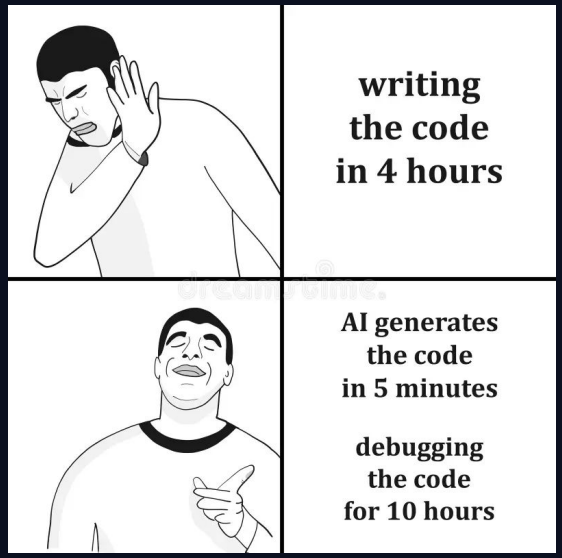
- From handcrafting every instruction
- To composing libraries and services
- To orchestrating delivery systems
- To designing verifiable intent for AI agents



AI-assisted coding: the first wave

The assistant lives inside the implementation loop

- Suggests code, tests, snippets, refactors, and explanations
- Works best when the developer owns the plan and validation
- Optimizes local productivity rather than system-level outcomes



The evolution of AI code generation

In less than 2 years we moved from 33% to 88% of accuracy

SWE-bench scores from Aug 2024 to June 2026 (source)

What AI-assisted coding still leaves unresolved

Fast code is not automatically correct code

- Ambiguous intent can generate plausible but wrong implementations
- Local suggestions can miss architecture, policy, or domain constraints
- Validation often remains manual, late, or implicit

Speed without specification can amplify uncertainty

02

What changes with Agentic AI

Agentic AI changes the loop: planning, acting, testing, and revising

What makes AI “agentic”?

An harness for a LLM with a defined context

- It can decompose goals into tasks
- It can use tools: code, shell, tests, docs, issue trackers, APIs
- It can evaluate results and iterate
- It can maintain context across a workflow

An agent is not just a model, it is a model embedded in an action loop

AI-assisted coding vs. Agentic Engineering

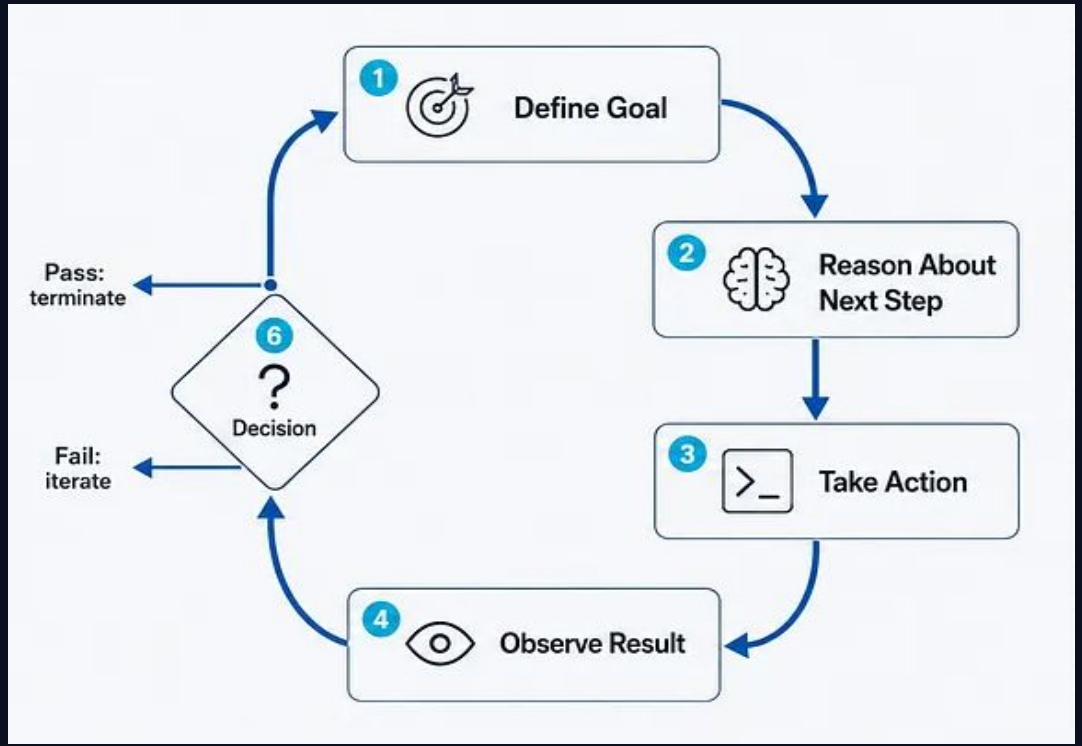
The core shift you wanted to discuss

- **AI-assisted coding**: human drives, AI suggests
- **Agentic Engineering**: human specifies, agent executes, evidence decides
- The deliverable expands from code to code plus tests, traces, decisions, and verification

The center of gravity moves from prompt-to-code to spec-to-system

The new engineering loop

Agentic workflows are iterative and evidence-based



The loop is only trustworthy when the success condition is explicit

Source of the image

Where agents create leverage

Tasks that benefit from autonomy

- Feature scaffolding across files and services
- Migration and refactoring work
- Test generation and failure triage
- Documentation, API clients, and repetitive integration work

Agents are strongest when goals are bounded and verification is available

Where agents create new risk

Autonomy increases the need for guardrails

- They can confidently satisfy the wrong objective
- They can change too much surface area
- They can pass shallow tests while violating hidden requirements
- They can make undocumented architectural decisions

Autonomy without specification becomes uncontrolled delegation

03

Why specs become critical

The specification becomes the source of intent, constraint, and verification

What is a “spec” in this context?

Not a document for a shelf; a machine-usable contract

- **Intent**: what outcome should exist
- **Scope**: what is included and excluded
- **Constraints**: architecture, security, performance, compliance
- **Acceptance criteria**: how success is proven

A spec is executable intent, even before it becomes executable code

Why specs matter more with AI agents

Agents need a stable target and a verifiable finish line

- They reduce ambiguity in planning and implementation
- They support automated test creation and review
- They preserve human intent across long workflows
- They provide auditability for decisions and changes

The better the spec, the smaller the trust gap

Example: spec-kit

https://github.com/github/spec-kit

- An open source toolkit from Github that allows you to focus on product scenarios and predictable outcomes instead of vibe coding every piece from scratch
- **Spec-Driven Development** flips the script on traditional software development
- For decades, code has been king — specifications were just scaffolding we built and discarded once the "real work" of coding began
- **Driven Development** changes this: specifications become executable, directly generating working implementations rather than just guiding them

A typical spec-kit workflow

1. **/speckit.constitution**
   Define project principles, constraints, quality rules, coding standards.

2. **/speckit.specify**
   Describe the feature in terms of intent, users, behavior, and outcomes.

3. **/speckit.clarify**
   Let the agent find ambiguities and ask questions.

4. **/speckit.checklist**
   Generate checks to validate the quality and completeness of the spec.

5. **/speckit.plan**
   Define architecture, technology choices, data model, integration strategy.

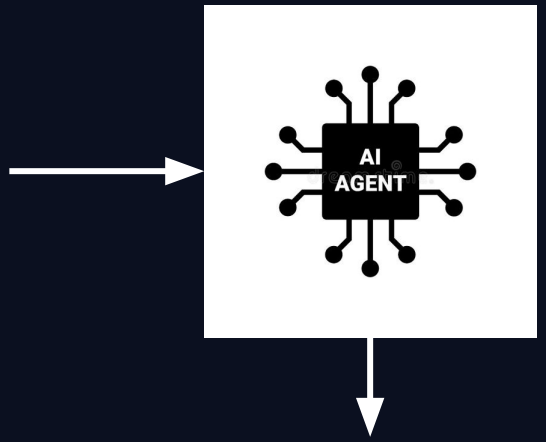
6. **/speckit.tasks**
   Break the plan into actionable implementation tasks.

7. **/speckit.analyze**
   Check consistency across constitution, spec, plan, and tasks.

8. **/speckit.implement**
   Let the agent implement the tasks.

Spec-kit outcome

- A markdown that contains:
  - **Feature**
  - **User goal**
  - **Functional requirements**
  - **Constraints**
  - **Acceptance criteria**



- The Agent can derive:
  - **Plan**
  - **Tasks**

From prompt engineering to spec engineering

The durable artifact is not the prompt

- Prompts are often conversational, temporary, and under-specified
- Specs are versioned, reviewable, testable, and reusable
- The prompt becomes a delivery mechanism for the spec, not the source of truth

Prompting is interaction; specification is engineering

04

How Agentic Engineering works

How teams structure work when agents become collaborators

Agentic Engineering workflow

A practical end-to-end flow

- Human creates or reviews the spec
- Agent proposes plan, assumptions, and affected files
- Agent implements in small, reviewable increments
- Automated checks produce evidence
- Human approves decisions and merges with traceability

Humans own intent and accountability; agents own execution loops

New artifacts in the workflow

Code is only one part of the output

- Spec files and decision records
- Generated test cases and evaluation reports
- Agent plans, logs, and change summaries
- Policy checks and security findings
- Human review comments linked to acceptance criteria

The engineering record expands beyond commits

Human-in-the-loop control points

Where judgment must remain explicit

- Approve scope and non-goals before implementation
- Review assumptions before large changes
- Require evidence for acceptance criteria
- Escalate ambiguous requirements to humans
- Merge only when traceability is clear

Human oversight must be part of the design, not a fallback when things go wrong

Metrics that matter

How to measure spec-driven agentic work

- Cycle time from accepted spec to verified change
- Defect leakage from ambiguous or missing criteria
- Review effort per change size
- Automated test coverage of acceptance criteria
- Rework caused by incorrect assumptions

Measure the quality of intent, not only the speed of output

Adoption roadmap

A staged path for teams

- Start with bounded tasks: tests, docs, refactors, small features
- Standardize spec templates and acceptance criteria
- Add automated checks and policy gates
- Introduce agent execution with traceability
- Scale to larger workflows after feedback loops are reliable

Before granting autonomy, define how outcomes will be verified

05

Practical adoptions, patterns and risks

From Promise to Practice: Scaling with Control

What changes for developers

The skills that become more valuable

- Requirement decomposition and domain modeling
- Writing precise acceptance criteria
- Designing testable interfaces and constraints
- Reviewing generated plans and evidence
- Maintaining architecture and socio-technical context

The best developers become intent designers and evidence reviewers

Engineers are becoming managers of AI agents

The new workflow

- Adoption Patterns: The **Agentic Engineering Loop**
  - Human approval before implementation
  - Small, reversible changes
  - Test-first or acceptance-criteria-first workflows
  - Traceability from requirement → task → code → test
  - Agent output reviewed like a pull request, not accepted as truth

Key Risks: Where Agentic Engineering Break

Agentic AI does not remove engineering risk

1. Ambiguous specs
2. Over-trust
3. Security & Compliance
4. Loss of human-in-the-loop

Source:
https://www.youtube.com/watch?v=m8I2p3nQtR4

Key Risks: Ambiguous specs

- **Ambiguous specs:**
  - Vague requirements produce plausible but wrong implementations
  - Missing constraints become hidden assumptions
  - Edge cases are often invented or ignored by the agent
- **Mitigation:**
  - Use clarification steps, acceptance criteria, examples, non-goals, and explicit constraints

Key Risks: Over-trust

- **Over-trust:**
  - AI-generated code can appear correct but be subtly wrong
  - Teams may skip review because output is fast and confident
  - Hallucinated APIs, libraries, or architecture choices can enter production
  - Errors scale quickly when agents automate repetitive changes
- **Mitigation:**
  - Require validation loops: tests, code review, static analysis, benchmarks, and explicit confidence checks

Key Risks: Security & Compliance

- **Security & Compliance**
  - Agents may expose secrets, logs, or sensitive data
  - Generated code can introduce insecure dependencies or patterns
  - Compliance requirements may be missed without explicit constraints
  - Automated changes can bypass normal governance controls
- **Mitigation**
  - Embed security policies into the workflow: least-privilege access, secret scanning, dependency checks, audit logs, and compliance gates

Key Risks: Loss of human-in-the-loop

- **Loss of human-in-the-loop**
  - Humans may become reviewers of outcomes only, not decisions
  - Critical trade-offs can be made without product or engineering judgment
  - Accountability becomes unclear when agents act autonomously
  - Team learning may decline if agents hide implementation reasoning
- **Mitigation**
  - Keep humans in control of intent, approval, and exceptions: require checkpoints for architecture, risk, production deploys, and irreversible actions

Back to Babbage and conclusion

- **Jacquard**:
  - The punch card controls the loom.
- **Babbage**:
  - The card controls the computation.
- **Software**:
  - The program controls the machine.
- **AI Agents**:
  - What controls the agent?

- In the AI era, the **specification becomes the architecture**

References

- Tim Robinson, *The Analytical Engine: 28 Plans and Counting,* Computer History Museum
- Raúl Rojas, *How Charles Babbage Invented the Computer,* 2023
- Simone Daniotti et al., *Who is using AI to code? Global diffusion and impact of generative AI*, Science, 22 Jan 2026, Vol 391, Issue 6787, pp. 831-835
- Alessio Bucaioni et al., *A Functional Software Reference Architecture for LLM-Integrated Systems*, IEEE International Conference on Software Architecture (ICSA), 2025
- Esposito, M., Li, X., Moreschini, S., et al. *Generative AI for software architecture: Applications, challenges, and future directions*. Journal of Systems and Software, 231, 2026
- Burak Gülmez , *Code generation with large language models: a survey from neural program synthesis to autonomous software development*, Appl Intell 56, 2026
- Christos Hadjichristofi et al., *An Empirical Evaluation of Large Language Models Applying Software Architectural Patterns*, AI. 2026; 7(6):195.
- Enrico Zimuel, *Software Engineering in the Age of Generative AI. Managing Uncertainty and Agentic Design*, 2025

Thanks!

Contacts:
Email: enrico@zimuel.it
Linkedin: https://www.linkedin.com/in/ezimuel/
Github: https://github.com/ezimuel

Linkedin