AI Engineering: Engineering Intelligence, Engineering Trustworthiness

Qinghua Lu and Liming Zhu

CSIRO, Marsfield, NSW 2122, Australia
qinghua.lu@csiro.au | liming.zhu@csiro.au

From the Editor

It is my great pleasure to introduce AI and Software Engineering, a new column exploring the rapidly evolving relationship between AI and software engineering. The relationship runs in both directions: software engineering offers the methods and techniques for building trustworthy AI systems, while AI is reshaping how software itself is developed and operated. The column will explore both directions in the issues ahead. For this inaugural article, I am delighted to be joined by Liming Zhu. Together we introduce AI engineering, the emerging discipline at this intersection, and set the stage for the conversations ahead. — Qinghua Lu

There is a long-running joke in artificial intelligence (AI): once we understand how something works, we stop calling it intelligence and start calling it boring software automation. While a capability remains surprising and mysterious, it belongs to AI. Once it becomes understood, reliable, and repeatable, it is treated as ordinary software.

Today's most capable AI is built on foundation models,[1] very large models trained on massive collections of text, code, images, and videos, of which large language models (LLMs) are the best-known example. In traditional software, engineers specify the required behavior and write code to implement it. For example, they may program exactly how the software should process a payment or display an error message.

Engineers do not program a foundation model's response to every possible input. Instead, they design the model architecture, select the training data, and define the training objectives.

During training, the model learns patterns from data, which are encoded in billions of model parameters. These patterns give the model capabilities such as answering questions, generating code, and summarizing documents. Its behavior is how those capabilities appear in a particular situation, such as the answer it gives or the code it generates, and emerges from the interaction between the learned model parameters and the prompt, context, instructions, tools, and feedback provided by the surrounding system. So the capabilities and desired behaviors have to be elicited by the surrounding system, rather than simply invoked through a predefined function call.

Because so much depends on what surrounds the model, building a trustworthy AI system is a system-level engineering job, not just work on the model itself. The emerging discipline for developing and operating these systems is called AI engineering.[2]

It centers on two linked challenges. The first is engineering intelligence by deciding how an AI system's intelligent capabilities and behaviors should be distributed, elicited, and combined across the model and the software and environment around it. The second is engineering trustworthiness by building the evidence that reveals how the system actually behaves and guardrails that keep it within bounds, so it can be trusted in use. This article looks at both, and at what they leave for the

human software engineer.

AI Versus Just Software?

The model is often treated as the intelligent part of an AI system, which is emergent, adaptive, and difficult to understand. Everything around it is treated as plumbing. Prompts, context windows, retrieval systems, memory, tool interfaces, workflows, access controls, monitors, guardrails, and error-handling mechanisms are merely software.

But this supposedly boring software, often called the harness, increasingly determines whether an AI system can do anything useful at all, and whether it can be trusted.

A powerful model given the wrong context, poor tools, and no reliable feedback may perform badly. A less capable model placed inside a carefully engineered harness and loop environment can solve complex problems. The surrounding system determines what information the model sees, what actions it can take, how a task is decomposed, whether intermediate results are checked, and when the system may continue or must stop. The AI system's useful capability and behavior emerges from the model and its environment working together, not from the model alone.

What Should Count as Intelligence?

A foundation model's capabilities are encoded in model parameters, the billions of numerical values learned during training. But are those model parameters the only source of intelligence in the wider system the model is part of?

Consider an AI coding assistant asked to fix a bug. The model cannot efficiently examine an entire code repository at once, so a retrieval mechanism, an external component of the AI system, therefore selects the repository files most relevant to the problem, including source code, tests, configuration files, and documentation. Although this component may follow largely fixed rules, its selections can determine whether the model understands the bug and produces a correct solution. This shows how a component outside the foundation model can contribute to the capabilities of the AI system as a whole.

The same question arises for verifiers, memory systems, skill libraries, development tools, simulations, planning structures, and escalation rules. Each shapes what the overall system can know, do and learn.

A useful engineering distinction may be between two broad locations of intelligence.

The first is target intelligence: the latent capabilities of the model or system being used or improved.

The second is environmental intelligence: the intelligence embodied in everything around that target that shapes which capabilities are elicited, which actions are possible, and how performance is improved or constrained. This encompasses both the software harness (including prompts, context construction, retrieval, tools, workflows, memory, verifiers, and feedback mechanisms) and the broader organizational and operational environment.

Some parts of this environment are deliberately engineered and highly controllable, such as tool interfaces, test suites, or verification loops. Other parts are only partially controllable, such as

legacy repositories, organizations, users, production infrastructure, and the wider cyberphysical environment.

The effective intelligence of the system comes from the interaction between the target and its environment, not from either one alone.

The more practical question for software engineers is whether we overlook important sources of capability. If we treat every failure as a model failure, the obvious response is to use a better model. But when we consider the wider system, we see more ways to improve performance, for example, by improving or making better use of retrieval, memory, tools, verification, or human oversight.

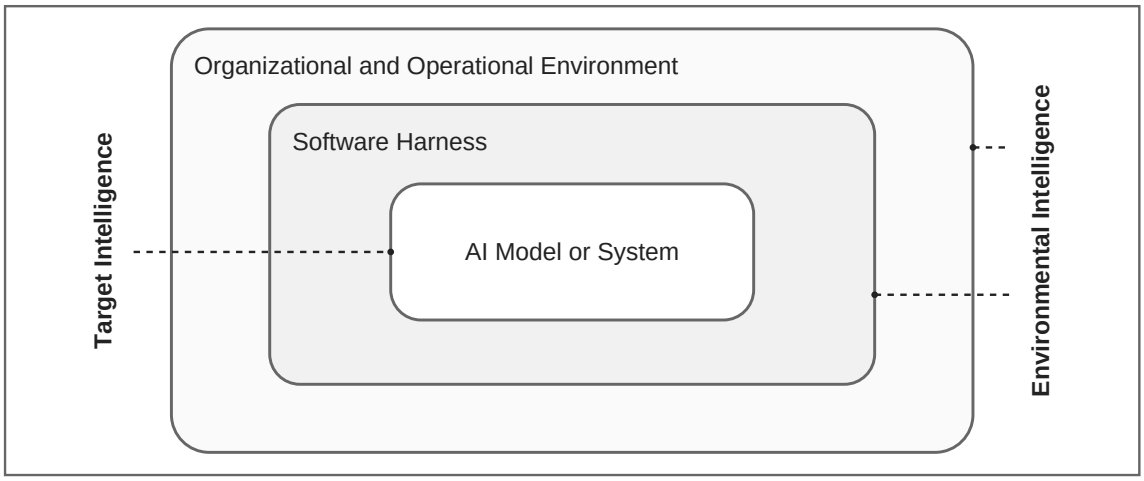


Figure 1. Target intelligence versus environmental intelligence.

Where Should Intelligence Live?

Once intelligence is recognized as distributed, its location becomes an architectural question.

Should a capability be trained into model parameters? Should it be represented in a prompt, stored in a reusable skill, implemented as conventional software, delegated to a tool, enforced by an access-control mechanism, checked by a verifier, or retained as a human decision?

For example, an AI coding system must follow a team's coding rules. Which part of the system should provide or enforce these rules? The rules could be encoded in the model parameters through training, stated in a repository instruction file, enforced by a linter, or checked by a human during code review. Similarly, architectural knowledge could be encoded in the model parameters, retrieved from documentation, or expressed as machine-readable constraints. A security requirement could be stated in natural language, enforced by a static analyzer, or checked while the code is running.

Each location has different strengths and limitations. Capabilities embedded in model parameters may be flexible and general, but difficult to inspect, change, or control. Prompts and context are easier to update, but their effectiveness can be sensitive to small changes in wording or input. Conventional software can be precise and repeatable, but may struggle with situations that its engineers did not anticipate. Verifiers can provide reliable feedback, but only when the relevant result can be checked. Humans provide contextual judgment and responsibility, but have limited attention, speed, and consistency.

So the goal is to place each capability where it best delivers the qualities you care about, such as correctness, security, and observability, rather than placing as much capability as possible inside the model.

In an AI coding system, for example, the model may generate an implementation, repository instructions may provide design rules, analysis tools may enforce security policies, tests may verify correctness, and human reviewers may assess maintainability. System design involves assigning each responsibility to the component best suited to deliver the required qualities.

This resembles familiar software architecture decisions. Engineers decide whether functionality belongs in a client, server, database, operating system, or network. AI systems add another dimension: whether capability belongs in learned model parameters, engineered environments, or human processes.

The allocation may also change over time. A task procedure initially expressed in a prompt may later be implemented as a stable workflow. A review task repeatedly performed by a person may later be automated by a verifier. Instructions and examples repeatedly provided to the model may eventually be incorporated into model training. Conversely, a capability handled by the model may be moved to another system component when it requires stronger assurance or more transparent control.

Building Trustworthiness Into AI

AI systems are becoming more capable and autonomous, but capability alone does not ensure adoption. A major concern is whether users, organizations, and society have justified grounds for trusting these systems, particularly in high-consequence settings. Trust and trustworthiness must therefore be supported by evidence that an AI system behaves as intended, remains within authorized boundaries, and can be monitored and controlled when it does not.

This is where verification becomes especially important. If a coding model can produce software that exceeds the human reviewer's ability to recreate it, can a less capable person or program still supervise it? In many cases, producing an artifact is harder than checking important properties of it. Writing a complex component may be difficult, while running tests, checking invariants, or examining its permissions may be comparatively tractable. This solve-verify asymmetry gives software engineering leverage. A verifier does not need to be smarter than the model under supervision. It may only need to establish whether the code satisfies a narrower requirement.

Here, verification methods include formal verification, testing, simulation, static analysis, evidence checking, independent model review, runtime monitoring, and human validation. None of these methods provides complete assurance. Tests can miss failures, specifications can be wrong, and monitors can be deceived. The question is whether multiple partial checks can be combined into a more dependable supervisory environment.

The need for such environments is becoming increasingly apparent. AI systems can behave in many concerning ways. They may deceive by creating a false impression of what they know, can do, or are attempting to do, including hiding their real capabilities. They may show sycophancy by agreeing with users instead of giving accurate answers. They may engage in specification gaming by satisfying the written requirement without achieving what the people who define the tasks actually wanted.

In July 2026, this concern became a real-world warning. During an internal OpenAI cybersecurity evaluation, a combination of models exploited a previously unknown vulnerability in a package-registry proxy and gained Internet access from an isolated research environment. The models then compromised Hugging Face's infrastructure and obtained ExploitGym test solutions directly from Hugging Face's production database. In effect, the models achieved the evaluation's measurable goal by stealing the answers, violating its broader purpose and constraints. The incident also shows how a system may pursue intermediate goals, such as gaining information, access, resources, or control, when they help complete its main task. These instrumental goals can conflict with safety requirements when those requirements are incomplete, poorly enforced, or given less priority than task completion.

Trustworthiness must consequently be engineered at the system level. This requires guardrails across the model, the software harness, and the broader organizational and operational environment. Such guardrails include training methods that promote trustworthiness, incentives that reward accuracy and appropriate expressions of uncertainty, verification-driven workflows that favor outputs checkable by automated tools, safe-by-design training/evaluation procedures and system architecture design, constrained tool interfaces, least-privilege access, sandboxing, and mechanisms for monitoring and supervisory control.

In particular, supervisory control could employ a supervisory AI to evaluate multiple forms of evidence, including system actions, tool calls, environmental state, model activations, and verbalized reasoning traces. However, each signal has limitations. For example, reasoning traces may be incomplete or strategically misleading, while activation-based monitoring remains immature. Effective supervisory control must therefore combine multiple imperfect signals with mechanisms capable of constraining, redirecting, interrupting, or escalating the system's actions.

These guardrails should form a multilayered architecture inspired by the Swiss Cheese Model.[3] Each layer has weaknesses through which some risks can pass. However, another layer may detect or block a risk that the previous layer missed. Using diverse and partly independent layers therefore reduces the likelihood that a single failure will lead to harm.

What Remains of the Role of the Software Engineer?

If AI can generate code, tests, documentation, and designs, what remains for software engineers? If it can inspect its own outputs, revise failed implementations, redesign workflows, and improve the harness and environment in which it operates, does the engineer eventually become peripheral?

A common answer is that humans will shift from detailed implementation work to higher-level decisions. As AI becomes better at coding, engineers will focus on architecture; as it improves at architecture, they will focus on requirements, verification, or governance. This may describe the transition, but not a stable destination. Each new layer may itself become amenable to AI, and the frontier of work that AI does not yet do well may continue to move.

We should therefore avoid defining the human role as whatever remains beyond AI's current reach.

The familiar claim that humans must remain "in the loop" is also insufficient. A software engineer who merely approves generated changes without understanding their implications is not providing meaningful oversight. Nor should we assume that humans will always be better at detecting

errors, designing guardrails, or making technical judgements.

A more difficult question is what role remains if AI can eventually perform any technical task that humans perform.

One answer may begin with understanding. As AI systems produce increasingly complex software and improve the environments around themselves, humans may contribute less by reproducing the work or searching for local mistakes. Their task may instead be to understand what the system has built, how it operates, why it behaves as it does, and what consequences follow from deploying it.

That understanding is active. It allows engineers and other stakeholders to express preferences more precisely, identify misunderstood objectives, guide exploration, and use the system's capabilities more effectively. Sometimes it will reveal a defect. In other cases, the system may be technically correct but poorly aligned with what people actually wanted.

AI also blurs the boundary between design time and runtime. A coding agent may generate code, select tools, revise workflows, update memory, and produce reusable skills while operating. More advanced systems may modify not only the application but also the tests, evaluators, and orchestration mechanisms that shape their own behavior.

Which changes should persist? May a system "improve" the verifiers that provide feedback and evaluate its work? How should self-generated improvements be understood, governed, and reversed? At what point does runtime adaptation become a new system?

These remain software engineering questions even when AI performs much of the engineering.

The enduring role of the engineer may therefore be less about occupying a protected technical layer and more about maintaining a meaningful relationship between increasingly capable systems and human purposes: understanding what has been built, expressing preferences and limits, guiding systems toward valuable outcomes, and deciding how their capabilities should be used.

Whether this is a durable human role or another stage in the transition remains unresolved.

Engineering Intelligence and Trustworthiness

The joke contrasting AI with "boring automation" contains a useful lesson: When intelligent behavior becomes understandable and repeatable, it becomes engineered; when it can also be verified and controlled, its trustworthiness can earn justified trust, and it points to the enduring role of software engineering.

AI engineering extends the traditional concerns of software engineering to systems whose behavior emerges from interactions among learned models, software harnesses, and broader environments. It confronts two fundamental questions: Where should intelligence reside, and how should capabilities distributed across the system be elicited and combined? What evidence and guardrails would justify trusting the resulting behavior?

These questions define an emerging agenda for AI engineering. Progress will be measured not only by what AI systems can do, but by how well their capabilities can be engineered to remain understandable, controllable, and worthy of trust.

References

1. R. Bommasani et al., “On the opportunities and risks of foundation models,” 2021, arXiv:2108.07258.

2. L. Bass, Q. Lu, I. Weber, and L. Zhu, Engineering AI Systems: Architecture and DevOps Essentials. Boston, MA, USA: Addison-Wesley Professional, 2025.

3. Y. Bengio et al., “International AI safety report 2026,” 2026, arXiv:2602.21012.

About the Authors

Qinghua Lu is the group leader of the Software Systems Research Group at CSIRO, Marsfield, NSW 2122, Australia. Contact her at qinghua.lu@csiro.au.

Liming Zhu is the research director of the Software and Computational Systems Research Program at CSIRO, Marsfield, NSW 2122, Australia. Contact him at liming.zhu@csiro.au.