The Engineering Stack: Prompt, Context, Harness, Loop, and Graph Engineering for AI-Assisted Development

Edward Abel
abel@sdu.dk

Version 1.0 — August 2026. Current version is always available here:
edabel.co.uk/publications/TheEngineeringStack.pdf

Abstract. This reference sheet positions the informally-coined terminology that emerged around AI-assisted software development in 2026 — prompt, context, harness, loop, and graph engineering — within a single working framework. It further situates retrieval-augmented generation (RAG) and fine-tuning relative to this stack, and discusses where the threshold for "agentic programming" is generally understood to sit. Definitions are drawn from current practitioner writing and an early arXiv preprint, reflecting a fast-moving and still-consolidating area of terminology.

Keywords: *prompt engineering · context engineering · harness engineering · loop engineering · graph engineering · retrieval-augmented generation · fine-tuning · agentic programming*

1 Introduction

A cluster of new terms has emerged around AI-assisted development in 2026: prompt engineering, context engineering, harness engineering, loop engineering, and — more recently and less settled — graph engineering. These are widely-used working definitions rather than a single formal standard; different writers draw the lines slightly differently, and this is noted where relevant. Figure 1 gives a schematic overview of the stack. This sheet also addresses two terms that are frequently raised alongside the stack but answer a different question — retrieval-augmented generation (RAG) and fine-tuning — and closes with a discussion of where "agentic programming" sits relative to the five layers.

2 The Five-Layer Stack

Figure 1 shows the five layers as a simple pipeline, together with how the two adjacent concepts discussed later in this sheet relate to it: RAG feeds into context engineering as one of its inputs (Sect. 4), and "agentic programming" is best read as a threshold spanning the harness, loop, and graph layers rather than a further step after them (Sect. 5).

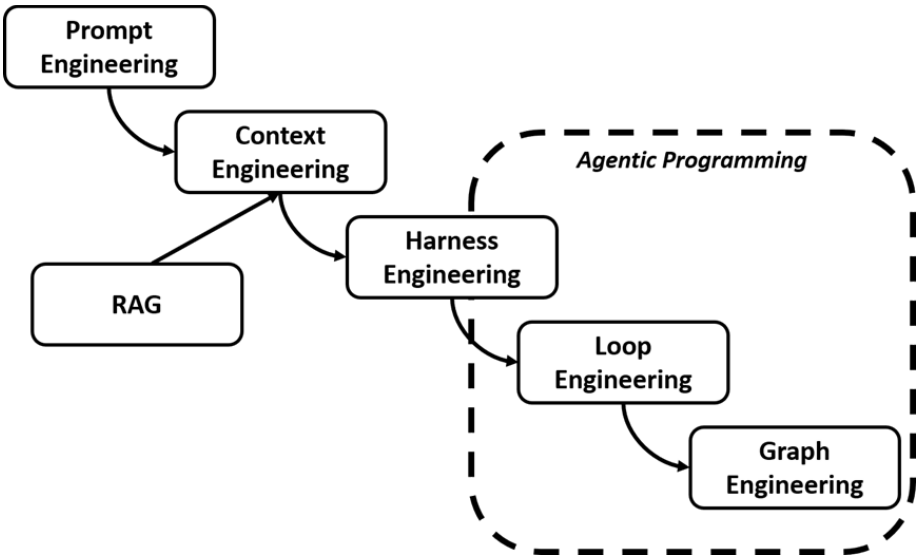

Fig. 1. The five-layer engineering stack. RAG supplies context engineering with retrieved, document-sourced input; "agentic programming" is a behavioural threshold spanning harness, loop, and graph engineering rather than a discrete sixth layer.

Table 1 summarises the five layers in more detail. Each layer *wraps* the previous one rather than replacing it — a loop still needs prompts, a harness still needs context, and graphs are built from loops which are built from prompts [1][2]. A common one-line framing: *prompt engineering makes the model smart for one turn; loop engineering makes the system reliable for a thousand turns* [3].

Table 1. The five engineering layers, from single-turn wording to multi-agent coordination.

Layer	Core question	Unit of design	Simple example
Prompt engineering	How do I word this message?	A single input/instruction	Writing a clear, well-structured request in chat
Context engineering	What does the model see on this call?	The information environment	Pasting the right files/docs in first; RAG retrieval
Harness engineering	What environment/tools surround the model?	The executable environment	Claude Code / opencode itself — file access, shell, memory
Loop engineering	What autonomous cycle repeats until a goal is met?	The repeating cycle (goal + stop condition)	"Fix code until tests pass, don't ask me each step"
Graph engineering (emerging)	How do multiple loops/agents coordinate?	The network of agents + shared state	Sub-agents working different parts of a repo in parallel

3 Layer-by-Layer Detail

3.1 Prompt Engineering

Prompt engineering covers everything that happens *within a single call*: wording, structure, examples, role-setting, output format, and chain-of-thought instructions [4][5]. It is the most established and mature discipline in this stack, dating to the 2022–2024 era [6][5], and it remains foundational — every layer above still depends on it for each individual step within a run [7][8].

On its own, however, it is *not agentic*: a well-worded prompt in a basic chat window is still a single human-driven turn, however good the wording.

3.2 Context Engineering

Context engineering is about curating *what fills the context window* — retrieved documents, conversation history, tool output, memory — as distinct from how the ask itself is worded [9][4][10]. It followed prompt engineering as the field's focus moved "from the words themselves to everything the model actually sees at the moment it responds," rising through 2025 [6]. Wikipedia frames it as the related discipline managing "non-prompt contexts... such as metadata, API tools, and tokens" [5].

Why this matters is well established in peer-reviewed work: a widely-cited study published in *Transactions of the Association for Computational Linguistics* found that model accuracy on long-context tasks follows a U-shaped curve, degrading sharply when the relevant information sits in the middle of a long context rather than near the start or end — regardless of how capable the model is otherwise [11]. Curating what goes into the window, and where, is not a cosmetic concern.

A simple example is uploading the right files before asking for an edit, rather than relying on the model's training knowledge. **Skills** (SKILL.md files) are essentially *harness-automated context engineering*: the system decides what context to load on the model's behalf, rather than the user hand-curating it every time [12].

3.3 Harness Engineering

Harness engineering covers everything that is not the model's weights: files, tools, the execution environment, memory, sandboxing, permission boundaries, and verification hooks [2][4]. Claude Code, opencode, and similar tools are harnesses in this sense — reportedly the term Anthropic engineers use for "the systems that prompt Claude iteratively — observe, plan, act, reflect — over hours" [9].

Within the harness, two sub-components are worth separating. **Skills** determine what the model *knows* — pre-packaged context, loaded on demand [12]. **MCP** determines what the model can *do* or *reach* — live tool calls, APIs, external systems, the layer that "connects the model to an executable environment" [13]. The distinction matters in practice: one reported finding is that "the same underlying model can perform up to 6x better depending on the quality of its surrounding harness and loop design, not the model itself" [7].

3.4 Loop Engineering

Loop engineering is the design of a control system that prompts, verifies, retries, and stops an agent — instead of a human doing that turn by turn [3][8]. It is defined by a **goal plus a stopping condition**, not a fixed step count, which is the core difference from a simple prompt chain [7]. The hard problems named consistently across sources are context management (avoiding "context rot"), termination, and verification [2][6].

The term itself is new, but the pattern it describes has a peer-reviewed academic root: the ReAct framework, presented at ICLR 2023, showed that prompting a model to interleave reasoning traces with concrete actions — rather than treating reasoning and acting as separate steps — improved performance on knowledge-intensive and interactive decision-making tasks [14]. Much of what is now packaged as loop engineering builds on that reason-act-observe cycle.

The term "loop engineering" was popularised in June 2026 by Google's Addy Osmani, synthesising public statements from Anthropic's Boris Cherny and OpenClaw founder Peter Steinberger [8]; Cherny is quoted as saying his job is now to "write loops, not individual prompts" [7]. It traces back to an earlier pattern nicknamed "Ralph," where engineer Geoffrey Huntley ran a coding agent inside a plain while-loop with a clean context reset each iteration [7].

3.5 Graph Engineering (emerging, least settled)

Graph engineering addresses what happens when one loop is not enough: multiple agents or loops coordinating on shared state [1][2]. At least one June 2026 arXiv paper on agentic AI names it as a step beyond loop engineering [1][13], though it is flagged directly as the newest and least agreed-upon label in the stack — "the term's provenance is unresolved and it collides with an older knowledge-graph usage of the same word" [1].

One rule of thumb offered by a source: if a task fits inside a single agent's context and domain, extend the loop — a single reasoning trace is cheapest for consistency. If independent branches genuinely need to run *at the same time*, treat it as a graph problem instead, and declare the nodes, edges, shared state, and failure routes explicitly [1].

4 RAG and Fine-Tuning in Context

Retrieval-augmented generation (RAG) and fine-tuning are frequently raised alongside the stack above but answer a different question: not "how do I structure this call?" but "where does the model's knowledge actually come from?" Table 2 summarises the distinction.

Table 2. Comparison of RAG and fine-tuning, and their position relative to the five-layer stack.

	RAG (Retrieval-Augmented Generation)	Fine-tuning
What it changes	Nothing in the model — weights stay frozen [15][16]	The model's weights, via further training [15]
How it works	Retrieves relevant documents at query time and injects them into the prompt [16]	Retrains the model on a curated dataset so it internalizes patterns [15]
Good for	Fast-changing/large external knowledge, source citation [17][15]	Consistent behavior, tone, format, domain reasoning style [18][16]
Position in stack	A technique within context engineering — retrieval-sourced context [17]	Outside this stack — a training-time operation, not call/run-time

Is RAG a subpart of context engineering? Yes. Context engineering is the broad activity of curating what fills the context window; it is about shaping the input that goes into an LLM without changing the model's parameters — including structuring prompts carefully and using retrieval-augmented generation to bring in relevant context at query time [17]. RAG is simply the version of that where the injected text is *retrieved from a document store* rather than hand-pasted or carried over from conversation history.

Does RAG relate to fine-tuning? No — they solve different problems by different mechanisms. RAG augments a model by connecting it to an organisation's proprietary database, while fine-tuning optimises the model itself for domain-specific tasks by retraining it [15]. Put simply: fine-tuning changes how a model thinks; RAG changes what a model knows at query time [16]. RAG keeps the base model frozen and pulls relevant chunks of text into the prompt at inference time [16] — so it remains fundamentally "giving the model context," just context sourced from documents via retrieval rather than typed or pasted directly into the prompt.

They are also not mutually exclusive — a common production pattern combines both: "behavior in weights, knowledge in context" — fine-tune a model on curated data to establish consistent tone, formatting, and standards, then deploy that fine-tuned model inside a context-engineering framework (often RAG) that supplies knowledge dynamically [18].

Both techniques rest on well-established, peer-reviewed foundations. The original RAG paper, presented at NeurIPS 2020, combined a pretrained retriever with a pretrained sequence-to-sequence model and showed the combination outperformed purely parametric models on knowledge-intensive tasks [19]. LoRA, presented at ICLR 2022, introduced a widely-adopted way to fine-tune large models cheaply by injecting small trainable low-rank matrices into each layer rather than updating all of a model's weights — the technique behind much of today's practical fine-tuning [20].

5 Where "Agentic Programming" Sits

There is no single agreed line on the scale — "agentic" describes a **behaviour**, not one specific layer. As shown in Fig. 1, it is best read as a threshold that begins around the Loop layer and continues to apply through Graph, rather than a distinct step after either of them.

> **Agentic ≈ Harness + Loop (and anything built on top, including Graph).**
>
> Harness alone → the model can act (files, shell, tools) but a human still drives each step — usually not yet called agentic.
>
> Harness + Loop → the system "observes, acts, verifies, and recovers" toward a goal, repeatedly, without approval at every step [13] — this is where most usage starts calling it agentic.
>
> Harness + Loop + Graph → still agentic, just coordinated across multiple loops/agents rather than one.

This lines up with how sources describe the shift itself: multiple industry figures are described as having "independently described abandoning manual prompting for designed, autonomous loops" [8].

5.1 Where People Disagree

Usage splits along a rough line. The looser reading treats any tool-using, file-editing setup as "agentic" — even one where a human approves each step. The stricter reading reserves the term for the autonomous, looped case only, treating human-approved step-by-step work as merely "agent-capable" rather than agentic [4].

Graph engineering muddies the scale further: whether coordinating multiple agentic loops counts as simply "more agentic" or as a qualitatively different layer is not settled [1][2].

Practical test most people apply informally: if a human has to read and approve every step → not agentic yet; if it runs until done or stuck on its own → it is.

6 Summary: A Quick Mental Model

Figure 2 draws the same five-layer pipeline as Fig. 1, but attaches a concrete, representative example to each stage — including RAG's role as an input to context engineering, and fine-tuning's deliberate lack of connection to the run-time stack.

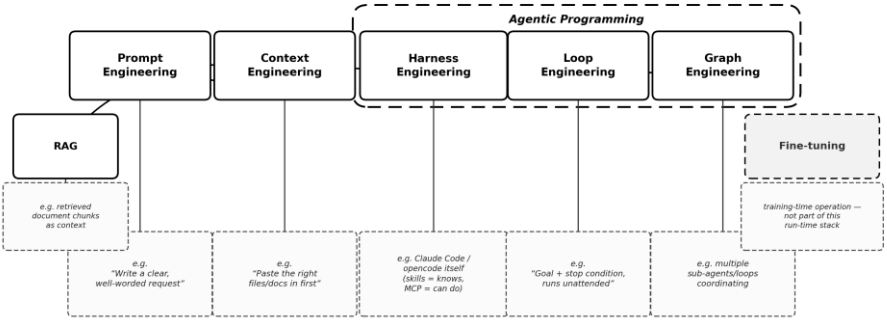

Fig. 2. The five-layer stack with representative examples at each stage. RAG and fine-tuning are shown at their respective positions relative to the pipeline: RAG as a retrieval-sourced input to context engineering, and fine-tuning as a training-time operation with no run-time link into the stack.

References

1. MarkTechPost: Prompt Engineering vs Loop Engineering vs Graph Engineering: What Changes at Each Layer (Jul. 29, 2026), https://www.marktechpost.com/2026/07/29/prompt-engineering-vs-loop-engineering-vs-graph-engineering/
2. Sarthakai: Harness, Graph, and Loop Engineering — How to Evolve From Prompts and Context. Substack, https://sarthakai.substack.com/p/harness-graph-and-loop-engineering
3. BuildFastWithAI: Loop Engineering: Complete Guide for AI Agents (Jul. 14, 2026), https://www.buildfastwithai.com/blogs/loop-engineering-ai-agents-guide
4. Chawla, A.: Prompt, Context, Harness & Loop Engineering. Daily Dose of Data Science (Jul. 3, 2026), https://blog.dailydoseofds.com/p/prompt-context-harness-and-loop-engineering
5. Wikipedia: Prompt engineering, https://en.wikipedia.org/wiki/Prompt_engineering
6. MachineLearningMastery: An Introduction to Loop Engineering, https://machinelearningmastery.com/an-introduction-to-loop-engineering/
7. NextAgile: From Prompt Engineering To Loop Engineering: The Next AI Shift (Jun. 30, 2026), https://nextagile.ai/blogs/gen-ai/prompt-engineering-to-loop-engineering/
8. Lyzr: Loop Engineering vs Prompt Engineering in 2026, https://www.lyzr.ai/blog/loop-engineering-vs-prompt-engineering/
9. explainx.ai: Context vs prompt vs loop vs harness engineering (Jun. 29, 2026), https://explainx.ai/blog/context-prompt-loop-harness-engineering-stack-2026
10. Towards Data Science: Prompt, Context, Loop: The Three Engineering Layers Every RAG System Is Built On, https://towardsdatascience.com/prompt-context-loop-the-three-engineering-layers-every-rag-system-is-built-on/
11. Liu, N.F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., Liang, P.: Lost in the Middle: How Language Models Use Long Contexts. Transactions of the Association for Computational Linguistics 12, 157–173 (2024)
12. OpenCode: Agent Skills. Official documentation, https://opencode.ai/docs/skills/

13. Buildrix: An Open Platform for Sharing and Benchmarking Agentic AI Skills in Building Engineering. arXiv preprint (Jun. 2026), https://arxiv.org/pdf/2606.25139

14. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., Cao, Y.: ReAct: Synergizing Reasoning and Acting in Language Models. In: International Conference on Learning Representations (ICLR) (2023)

15. IBM: RAG vs. Fine-tuning, https://www.ibm.com/think/topics/rag-vs-fine-tuning

16. DEV Community: RAG vs Fine-Tuning: What I Actually Learned After 6 Months of Building LLM Apps (Mar. 2026), https://dev.to/synsun/rag-vs-fine-tuning-what-i-actually-learned-after-6-months-of-building-llm-apps-1iac

17. Agami: Context Engineering vs Fine-Tuning vs Distillation: How to Decide (Aug. 2025), https://blog.agami.ai/context-engineering-vs-fine-tuning-vs-distillation-how-to-decide/

18. Promethium: Context Engineering vs RAG vs Fine-Tuning: Choosing the Right AI Data Strategy (Feb. 2026), https://promethium.ai/guides/context-engineering-vs-rag-vs-fine-tuning-ai-data-strategy/

19. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., Kiela, D.: Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In: Advances in Neural Information Processing Systems (NeurIPS) 33 (2020)

20. Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W.: LoRA: Low-Rank Adaptation of Large Language Models. In: International Conference on Learning Representations (ICLR) (2022)

Note on sourcing: this is fast-moving terminology, and right now the best thinking on it lives largely in practitioner blogs, newsletters, and preprints rather than in textbooks — which is simply where the liveliest, most current discussion is happening for a space this new. Alongside those, this sheet also draws on established peer-reviewed research wherever the underlying techniques connect to it. Together, these reflect an ongoing conversation that is worth revisiting as the field's language continues to settle.