Sequoia Ascent 2026 — Andrej Karpathy

From Vibe Coding to Agentic Engineering

What Karpathy's Talk at Sequoia Capital Means for Engineers Building Production AI Systems

Source: https://karpathy.bearblog.dev/sequoia-ascent-2026/

Sequoia Capital — April 29, 2026

The December Inflection Point

December 2025 was the moment agentic coding crossed from useful-but-unreliable to trustworthy end-to-end — and most engineers missed it. [1]

"I can't remember the last time I corrected it."

— Andrej Karpathy on trusting agent output for complete coding tasks

- The shift was **not incremental**: snippet help and correction loops gave way to coherent agentic workflows that run to completion
- The break period gave Karpathy uninterrupted time to notice the step-change — engineers on daily deadlines largely missed it
- **Implication:** if your mental model of LLM coding capability is pre-December 2025, your instincts are wrong

Software 1.0 → 2.0 → 3.0

LLMs are not faster compilers — they are a new programming substrate where the context window is your lever and natural language is your source code. [1]

Stage 1

Software 1.0

- Explicit rules
- Branching logic
- Hand-written code

Output: Deterministic execution

Stage 2

Software 2.0

- Datasets
- Objectives
- Neural architectures

Output: Learned weights

Stage 3

Software 3.0

- Prompts + context
- Tools + memory
- Instructions

Output: LLM as interpreter

The context window is the primary programming surface — you are not writing to a compiler, you are writing to an interpreter with its own intelligence

The MenuGen Lesson — Some Code Shouldn't Exist

The right Software 3.0 answer often eliminates entire application layers that Software 1.0 thinking assumed were necessary. [1]

Software 1.0 MenuGen

Hundreds of lines

1. **Photo upload** — user submits menu image to app
2. **OCR pipeline** — extract text from image
3. **Image generation API** — create overlay assets
4. **Re-render + Vercel deploy** — stitch and ship

⚠ Brittle orchestration — every step is a failure point

Software 3.0 MenuGen

Single model call

1. **Pass photo to Gemini** — no pre-processing, no storage step
2. **Single prompt:** "use Nana Banana to overlay items onto the menu"

```
pass photo to Gemini +
"use Nana Banana to overlay items onto the
menu"
```

✓ **Annotated image returned — pipeline eliminated**

> *"All of my MenuGen is spurious. That app shouldn't exist."*
>
> — Andrej Karpathy · Before scaffolding a service, ask: what is the text I paste to the model to get the output directly?

Engineering habit to break: reaching for a framework before asking whether a direct model transformation makes the service layer redundant.

Vibe Coding vs. Agentic Engineering

Vibe coding raises the floor for everyone; agentic engineering preserves the quality bar of professional software while going dramatically faster. [1]

Vibe Coding

Natural language → working prototype

- No accountability required — no production guarantees
- Democratizes access — anyone can build a working prototype
- Speed gains come without professional obligation
- Raises the floor: more people shipping more things, faster

Lowers barriers, but security and correctness are not guaranteed

Agentic Engineering

Same speed — professional standard maintained

- Engineer remains responsible for security, correctness, and maintainability
- Agents are stochastic and fallible — coordination and oversight are the job
- "You are not allowed to introduce vulnerabilities due to vibe coding."
- Speed ceiling is unknown — Karpathy: "10x is not the speedup you gain"

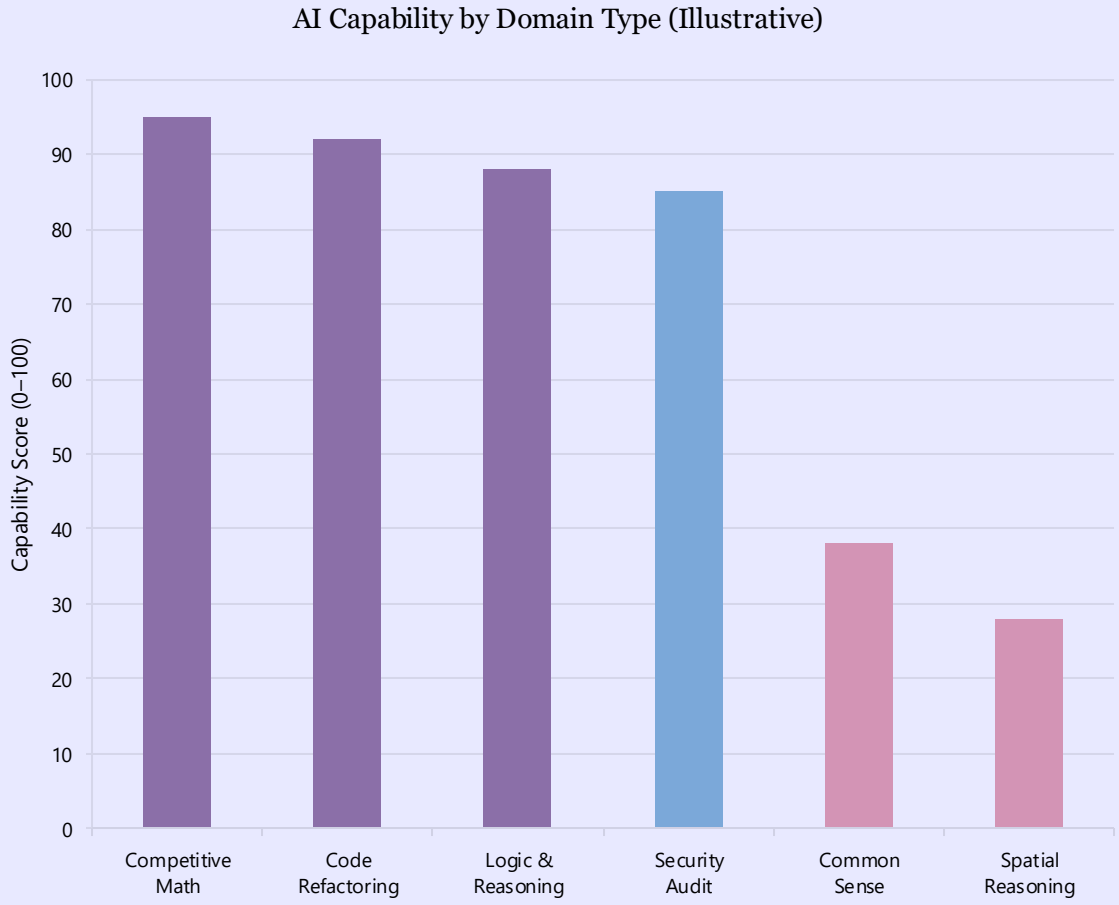
Practitioners who master agentic coordination peak far above 10x

Sequoia Ascent 2026 — Andrej Karpathy

Verifiability — Why AI Capability Is Jagged

AI automates fastest where outputs can be verified; understanding the shape of that jaggedness is now a core engineering skill. [1]

- Frontier labs train via **RL with verifiable reward signals** — models peak in math, code, and adjacent verifiable domains
- Jaggedness example: Opus 4.7 can refactor a 100,000-line codebase — but recommends walking 50m to a car wash
- Chess capability jumped GPT-3.5 → GPT-4 via a **deliberate data decision**, not emergent scaling alone
- Untargeted domains: expect rough edges — diagnose before deploying, consider fine-tuning to reshape RL distribution
- **Strategic white space:** valuable verifiable domains labs haven't targeted yet are the differentiation opportunity



Source: andrej-karpathy-from-vibe-coding-to-agentic-engineeri...

The Human-Agent Division of Labor

Agents handle recall and fill-in-the-blanks; engineers own spec design, taste, oversight, and conceptual understanding — that split is the new job description. [1]

What Agents Handle Well

- **API surface recall:** knowing `reshape` vs `permute`, `keepdims` vs `keep_dim` across frameworks
- **Boilerplate generation:** scaffolding, repetitive patterns, iterative refinement within a defined spec
- **Exhaustive search:** filling in a large, well-specified solution space quickly and without fatigue
- **Code aesthetics — not yet:** agent output tends toward bloat, copy-paste, and brittle abstractions — currently outside the RL reward signal

What Engineers Must Own

- **Identity & data design:** unique user ID semantics, data ownership, security model — email is not a stable foreign key
- **Abstract architecture:** which layers to build, which to eliminate — questions that require judgment, not lookup
- **Mental models that matter:** knowing tensor views share storage vs. copy prevents subtle perf bugs even when implementation is delegated
- **Simplicity & taste:** prompting models to minimize produces poor results — the push for elegance must come from the engineer

MenuGen bug: the agent cross-correlated Stripe and Google accounts by email address — a correct-looking implementation with a fundamental identity design flaw that only an engineer who understands the domain would catch.

Spec-First Development and Agent-Native Workflows

Working with agents at production quality requires spec-first thinking — write the detailed design doc first, then let agents fill it in under your oversight. [1]

1

Define

Context

What does the agent need to know about the system, constraints, and non-negotiables?

2

Define

Tools

What actions can the agent take, and what are the explicit boundaries on those actions?

3

Define

Feedback Loop

How will the agent know if its output is correct? Evals, tests, linting, type checks.

4

Define

Guardrails

What must never happen regardless of agent reasoning? Write these as hard constraints — not soft suggestions.

> *Work with your agent to design a spec that is very detailed — maybe the docs — and then get the agents to write them. Agent-native infrastructure is the next build surface: llms.txt, agent-readable APIs, prompt-driven deployment.*

Hiring and Evaluating Agentic Engineers

Traditional coding puzzles screen for skills that agents now supply; the signal has shifted to coordinating agents on large, adversarially tested projects. [1]

Old Signal

Solve an algorithmic puzzle in 45 minutes

New Signal

Build a Twitter clone — make it secure, deploy it, then withstand 10 parallel Codex 5.4 agents trying to break it

> *Most people have still not refactored their hiring process for agentic engineer capability.*

What to Observe in Candidates

- **Setup investment** — does the candidate scaffold a robust environment before delegating?
- **Feature utilization** — breadth and intentionality in using agentic tooling
- **Spec authorship** — ability to write, enforce, and iterate on a detailed specification
- **Override judgment** — knowing when to intercept the agent and correct its trajectory
- **Security baseline** — evaluation environment must itself be adversarial; security under automated attack is now baseline, not advanced

Self-Evaluation Prompt

Can you brief an agent on a large unfamiliar codebase and ship a correct, secure feature without touching every line yourself?

Ghosts, Not Animals — A Mental Model for LLM Behavior

LLMs are summoned statistical circuits shaped by data and reward, not intrinsic motivation — building an accurate mental model of what they are changes how you deploy and trust them. [1]

> *"We are not building animals. We are summoning ghosts."*
>
> *Jagged entities shaped by pre-training statistics and RL reward, not evolution or curiosity.*

- Negative prompting and frustration-driven iteration have no effect — LLMs have no intrinsic drive to satisfy you
- Inside RL circuits: behavior is near-magical; outside: you're pulling teeth — and that friction is itself a diagnostic signal
- When you hit friction: ask if the task maps onto a verifiable domain — if not, fine-tune, add examples, or restructure
- Maintain enough human understanding to catch the "car-wash class" of errors — invisible to the model, embarrassing in production

The Future Stack — Agent-Native Infrastructure and Personal Knowledge

The next platform shift is from human-readable software infrastructure to agent-first infrastructure, with personal knowledge bases as the new personal compute layer. [1]

Today's Gap

Frameworks, docs, and deployment UIs are designed for human eyes and hands — every settings menu and DNS config screen is friction for agents.

Target State

Give a prompt, the model builds the app and deploys it without human touch — Karpathy's test for truly agent-native infrastructure.

The Agent-Native Design Pattern

1. **Define Context**
   Sensors + actuators model: decompose workloads into what agents can observe and what they can change. Design APIs accordingly.
2. **Define Tools**
   LLM knowledge bases — a dynamic wiki compiled from raw documents, reordered and reframed by the model. A new primitive that couldn't exist before.
3. **Define Feedback Loop**
   Agent representation layer: agents handle inter-agent negotiation, scheduling, and coordination on behalf of people and organizations.
4. **Define Guardrails**
   Preserve human understanding: let agents work, but maintain enough comprehension to catch the errors that are invisible to the model.

The pattern: *define context → define tools → define feedback loop → define guardrails → let agents work → preserve human understanding*

What Still Requires Human Understanding

Thinking can be outsourced; understanding cannot — and understanding is what makes you an effective director of agents rather than a passive passenger. [1]

"You can outsource your thinking, but you can't outsource your understanding."

- **Agents simulate, not understand** — direction, judgment, and goal-setting remain irreducibly human
- **Bottleneck is comprehension**: knowing why something is worth building and what the system must guarantee
- **Learn to brief, evaluate, and override** — not to outperform agents line by line
- New core skills: **evals, guardrail design, spec authorship, taste**, and the systems understanding that catches what agents miss
- Knowledge base as a tool for preserving understanding — synthetic insight generation across a personal wiki corpus