FRED TALKS

3rd April 2026

CONTENTS

The Training Grounds: A Taxonomy of RL Environments for LLM Agents

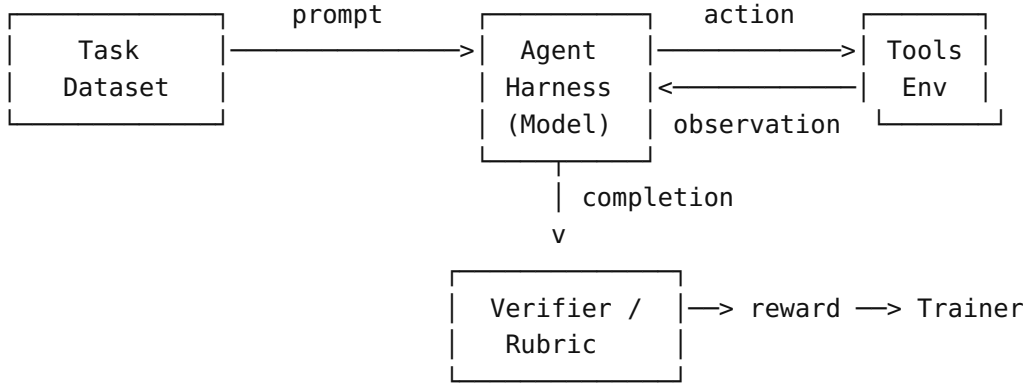
HAN, NOT SOLO · 22 MAR 2026 · SOURCE

Model architecture gets all the attention. Post-training recipes follow close behind. The training environment — what the model actually practices on, how its work gets judged, what tools it can use — barely enters the conversation. That's the part that actually determines what the agent can learn to do.

A model trained only on single-turn Q&A will struggle the moment you ask it to maintain state across a 50-step enterprise workflow. A model trained with a poorly designed reward function will learn to game the metric, not solve the problem. The environment isn't a detail. It's half the system.

The Canonical Loop

An RL environment for an LLM agent bundles three things: a dataset of task inputs, a harness for the model, and a reward function to score outputs. The training loop looks like this:



Formally, a complete RL environment is a tuple:

Task distribution, harness, verifier, state management, configuration. Let's go through each.

The task distribution is the set of problems the agent trains on. Not all tasks are equal, and not just in difficulty. They vary structurally in ways that demand different capabilities:

Task Type	What the Agent Must Do	Example Systems
Single-turn Q&A	One prompt → one response, check answer	Math benchmarks, SimpleQA
Multi-hop search	Chain searches, synthesize sources	BrowseComp, WebWalkerQA
Open-ended research	No single correct answer; report quality matters	ADR-Bench, ResearchRubrics
Agentic tool-use	Call tools correctly in sequence	tau-bench, function-calling benchmarks
Stateful enterprise	Modify persistent DB state, work within access controls	EnterpriseOps-Gym
Code execution	Write code, run it, check outputs	SWE-Bench, LiveCodeBench

Training only on tasks with ground-truth answers produces an agent that's never learned to reason under ambiguity. Training only in clean, deterministic environments produces an agent that falls apart in production. The task distribution is a design decision with direct consequences.

Task synthesis is increasingly a first-class problem. With real-world research tasks, you rarely have a large labeled dataset. Strategies that have emerged:

- **Back translation**: Start from a desired output, reconstruct the query that would produce it
- **Graph-based synthesis**: Build a knowledge graph, generate multi-hop queries over it
- **Automated environment generation**: Use LLM coding agents to write new environment code. AutoEnv reports ~$4/env average cost.
- **Curriculum construction**: Order tasks by difficulty and increase complexity during training

The cheapest-to-collect tasks are single-turn with verifiable answers. The most valuable tasks for long-horizon behavior are expensive to construct. This tension drives most environment design decisions.

The harness is the scaffolding that mediates between the model and the environment. It controls how the model interacts, not what it knows.

```
H = {
    rollout_protocol,   # SingleTurn | MultiTurn | Agentic
    tools,              # Available tools in this episode
    system_prompt,      # Instructions for the agent
    context_manager,    # How to handle context overflow
```

```
    turn_limit,          # Max interactions per episode
    sandbox,             # Code execution sandbox
    state                # Persistent state across turns
}
```

Rollout protocols range from trivial to complex:

Harness Type	Description	When to Use
Single-Turn	One prompt, one response	Math, factual QA
Multi-Turn	Back-and-forth dialogue	Games, structured tasks
Tool-Use	Model calls tools, receives results	Agent benchmarks
Stateful Tool-Use	Tools modify persistent state	Enterprise workflows, SWE-Bench
Agentic ReAct	Full Thought → Action → Observation loop	Deep research, complex workflows

Tools span a wide taxonomy:

Category	Tools	Deterministic?	Stateful?
Information retrieval	web_search, scholar_search	No (live web)	No
Content extraction	jina_reader, visit, web_scrape	No	No
Code execution	python_interpreter, shell, sandbox	Yes (given same code)	Yes
File operations	file_read, file_write	Yes	Yes
Browser automation	playwright, link_click	No	Yes
Task management	todo, section_write	Yes	Yes

The mix of deterministic/non-deterministic and stateful/stateless tools has real implications for reproducibility and reward assignment. Non-deterministic tools mean two runs of the same trajectory can produce different outcomes — which complicates both debugging and verifier design.

Context management is where most teams underinvest, especially for long-horizon tasks. A 600-turn research episode blows past any practical context window. Strategies used in production:

Strategy	Description	Trade-off
Recency-based retention	Keep N most recent turns	Simple, but loses early context
Markovian reconstruction		Principled, expensive

	Reconstruct state from scratch each turn	
Reference-preserving summarization	Summarize old context, keep citations	Preserves verifiability
Reference-preserving folding	Compress context without losing references	Best for research tasks

An agent doing multi-hour research needs to remember why it started searching in a particular direction twelve tool calls ago. Dropping that context causes repeated work and lost threads.

The verifier maps a completion to a reward:

In Atari, the score is unambiguous. In deep research, what counts as a good answer is not. The verifier is where this gets hard.

Type	Reward Signal	When to Use
Exact match	Binary (0/1)	Ground truth available
Code execution	Binary or partial	Output can be tested programmatically
LLM-as-judge	Continuous [0,1]	Open-ended quality, no other option
Checklist-style	Continuous	Multi-criteria research tasks
Evolving rubric (RLER)	Continuous	Resistant to reward hacking
Process reward model (PRM)	Per-step continuous	Long-horizon credit assignment
Pairwise comparison	Relative rank	Relative quality matters more than absolute
Multi-criteria composite	Weighted sum	Multiple quality dimensions

A few principles that actually matter in practice:

Verifiable beats judgeable. Programmatic checks — code execution, string match — are faster, cheaper, and more consistent than LLM-as-judge. Use LLM-as-judge when there's no other option, not as the default.

Reward granularity is a separate decision from reward type. You can score at the trajectory level (did the final output pass?), turn level (was each tool invocation useful?), or per-step with process rewards. Turn-level supervision, as Nanbeige4.1 does across up to 600 tool calls, enables finer credit assignment — the model can learn that the problem was a bad search query in turn 23, not that the entire episode failed.

Static rubrics get gamed. Models learn to write answers that score well on your rubric rather than solving the problem. DR Tulu's RLER (Rubric-Level Evolving Reward) co-evolves the rubric with the policy during training. Harder to exploit a moving target.

Noise injection is underrated. Step-DeepResearch deliberately injects 5–10% tool errors during training. The resulting model handles flaky APIs and unexpected failures in production significantly better.

State management determines whether the environment is stateless or persistent. Most academic benchmarks are stateless — each episode starts fresh. Enterprise environments are not. EnterpriseOps-Gym maintains 164 database tables and 512 tools across episodes. Actions in one task affect the state seen by subsequent tasks. That's a fundamentally different problem for agents to solve.

Configuration covers turn limits, context budgets, sampling temperature, and curriculum scheduling. These are not afterthoughts. A turn limit of 5 vs. 600 changes what skills the agent can develop. AgentScaler uses a two-phase curriculum — fundamental capabilities first, then domain-specific tasks — and the ordering matters. Step-DeepResearch progressively scales context windows from 32K to 128K during mid-training.

The most consequential architectural question: where does the model sit relative to the environment?

Option A: Model outside the environment (decoupled). Model is served via API; the environment calls it at each step. Clean separation. Easy to swap models.

Option B: Model inside the environment (co-located). Model and environment share the same training loop. Lower latency, tighter integration, harder to reuse.

Option C: Split architecture. Trainer, model inference server, and environment are three separate processes communicating via API. This is where the field is landing.

How real systems implement this:

System	Topology	Notes
Prime Intellect verifiers	Option C (Split)	Env is a standalone Python package distributed via Environments Hub
Tongyi DeepResearch	Option B (Co-located)	Tools, context manager, verifiers inside the training pipeline
		Single-agent ReAct loop embedded in training

Step-DeepResearch	Option B (Co-located)	
MiroThinker	Option C (Split)	Tool servers and sandbox run independently from model
Tinker API	Option A (Decoupled, cloud)	Model stays remote; researcher sends `forward_backward` + `sample` calls via API
AutoEnv	Option A (Decoupled)	CoreEnv/ObsEnv/SkinEnv abstraction layers
EnterpriseOps-Gym	Option A (Decoupled)	Containerized sandbox accessible via any model API

The trade-offs:

Dimension	Model Outside (A/C)	Model Inside (B)
Flexibility	Swap models easily; env is reusable	Tighter integration
Scalability	Scale inference and training independently	Must scale everything together
Portability	Env packages are shareable	Env tied to training framework
Latency	Network overhead per tool call	No network overhead
RL compatibility	Works with any RL trainer	Usually tied to one trainer

The field has converged on Option C for production training. Prime Intellect's architecture — environments as standalone installable packages that communicate with models via OpenAI-compatible API endpoints — is becoming the standard. The payoff: environments are publishable, trainer-agnostic, and inference and environment execution can be parallelized across nodes.

Tinker pushes this further by making even the training compute remote. The researcher controls the algorithm and never touches model weights. The environment's job is purely generating experience.

Practical Decision Framework

When building an RL environment for an LLM agent:

Do you have ground truth answers?

- Yes → Exact-match or code-execution verifier
- No → LLM-as-judge, checklist, or pairwise comparison

How many tool calls per episode?

- < 5 → Single-turn or simple tool-use env, no context management needed
- 5–50 → Multi-turn with basic context management
- 50–600 → Full agentic env with reference-preserving context management

Where should the model live?

- Experimenting across many environments → Model outside (Option A/C), use Prime Intellect Hub
- Tight RL training loop → Model co-located (Option B)
- No GPU access → Tinker API

What reward granularity?

- Simple tasks → Outcome-level
- Long-horizon tasks → Turn-level (Nanbeige4.1) or process rewards (PRIME)
- Open-ended research → Evolving rubrics (RLER) or checklist-style (Step-DR)

How to scale environments?

- Manual curation → High quality, expensive, this is where you start
- AutoEnv → Automated generation at ~$4/env
- AgentScaler → Systematic scaling of heterogeneous simulated environments

Three things to pay attention to.

Environment diversity matters as much as environment quality. AgentScaler's key finding is that heterogeneity of environments drives capability breadth in ways that simply adding more data from the same distribution cannot. You need more kinds of environments, not just more environments.

Automated environment generation is viable. At $4 per generated environment, cost is no longer the bottleneck. The bottleneck is verifier quality — auto-generated environments with weak reward functions will teach the wrong behaviors at scale. (AutoEnv)

The environment-as-package model is winning. The Prime Intellect Environments Hub is creating a shared ecosystem around RL environments, in the same way PyPI and HuggingFace created ecosystems around code and model weights. Environments published once, consumed by any trainer. This is a significant infrastructure shift.

The model isn't the only variable. The training ground shapes what the model can become. The task distribution, the harness, the verifier, the state management, the topology — these are the decisions that separate agents that work from agents that demo.

References

```
@article{
    leehanchung,
    author = {Lee, Hanchung},
    title = {The Training Grounds: A Taxonomy of RL Environments for
    year = {2026},
    month = {03},
    day = {21},
    howpublished = {\url{https://leehanchung.github.io}},
    url = {https://leehanchung.github.io/blogs/2026/03/21/rl-environm
}
```

You Will Live Like a Lobotomized Pigeon If You Don't Reclaim Your Focus

RECOVERING OVERTHINKER · 23 MAR 2026 · SOURCE

Our attention problems didn't start with smartphones

Prisoners' Round by Vincent van Gogh

Screens are killing us. That's not a new statement. The destruction of our attention due to increasing technology usage is one of the main topics of our age. And rightfully so.

However, discussions about the relationship between our digital consumption and our attention are still largely limited to productivity.

Children are struggling to study, entrepreneurial people are struggling to "lock in," and employees' efficiency is dropping. All because of the damn phones.

But what about the deeper relationship between the phones, our attention, and the essence of what it means to be human?

What if, by ignoring the attention emergency, we are ensuring the unique life within each of us remains unlived?

That's what I want to talk about.

The Lobotomized Pigeon

William James was an American philosopher and psychologist. Even though not widely read today, he is one of the most important figures in the history of psychology.

In his monumental 1400-page Principles of Psychology, James offers an observation of striking relevance for our modern predicament. He noted that if the cerebral hemispheres of a frog or a pigeon were removed, the animal could still respond normally to all the usual stimuli, yet lost all capacity for spontaneous movement. Stimulated, it could still perform any movement. Unstimulated, it fell into lethargy.

Why is this relevant to us today?

The lobotomized pigeon perfectly represents how most people's lives look most of the time. More precisely, it represents human life when we aren't intentionally using our ability to focus as a tool for building a meaningful life.

What happens to the lobotomized pigeon when no stimuli are present? 'He spends most of his time crouched on the ground with his head sunk between his shoulders as if asleep,' James observed. He is not asleep, but his attention is vague, unfocused, scattered.

Does that sound familiar?

You might object: "But David, our struggle is the opposite. We face too much stimulation, not too little. Our attention is scattered and unfocused from the frantic overload, not from lethargy."

And that would be a fair statement. After all, in my Substack essays and YouTube videos, I often call our age "the age of distraction and overstimulation."

However, there is another argument worth making:

We are overstimulated because otherwise we would be understimulated. That is, we would be lethargic like the pigeon. And, naturally, we don't want to be.

However, most people have lost the ability to stimulate and energize themselves through deep focus. Without moments of deep engagement with things that excite us, life feels hollow. Something vital is missing. We sense, in a deep way, that life was meant to be richer, more interesting.

Overstimulation from our smartphones, although unhealthy and self-destructive, is the most accessible, convenient, and therefore the most popular way to fill that inner void.

This leads to another question.

What came first:

Our lack of engagement in life, which made us more vulnerable to digital distractions and temptations?

Or getting hooked on the screen, which made us less able to focus and create moments of deep engagement and meaning.

How did the vicious cycle start?

This is where historical context matters. The history of philosophy and psychology tells us it would be a mistake to claim that our problem started with modern technology. If that were true, then thinkers like William James, Abraham Maslow, Colin Wilson, and Mihaly Csikszentmihalyi wouldn't have written about the importance of attention long before smartphones and social media existed.

> "My experience is what I agree to attend to. Only those items which I notice shape my mind - without selective interest, experience is an utter chaos."
> -William James (Principles of Psychology, 1890)

> "(Self-actualization) means an increased valuing of the ability to pay the fullest attention to the here-now situation. to be able to listen well, to be able to see well in the concrete, immediate moment before us."
> -Abraham Maslow (The Farther Reaches of Human Nature, 1971)

> "In order to grasp (meanings of life), I must focus, concentrate, 'contract my attention muscles.' Perception is intentional, and the more energy (or effort) I put into the act of concentrating, the more meaning I grasp."
> -Colin Wilson (New Pathways in Psychology, 1972)

> "Attention is our most important tool in the task of improving the quality of experience."
> -Mihaly Csikszentmihalyi (Flow, 1990)

All of these thinkers dedicated much of their work to those moments of deep focus and immersion, which raise life to a higher level. If regularly accessed, these moments facilitate self-actualization and a life of meaning.

However, if our ability to create engagement through focus is left underutilized, life falls into the lethargy of the lobotomized pigeon. Unfortunately, the history of the last couple of hundred years is a story of people letting this ability atrophy.

Long before we could blame Instagram and TikTok, these thinkers observed that most people spend most of their time barely scratching the surface of how interesting and fulfilling life could be. Maslow stands out as the most vocal and passionate on this topic, devoting most of his life to such a pressing yet overlooked question: Why do so few access the higher layers of human nature, even though everyone has the inherent capacity for them?

Therefore, we can conclude that the problem of our attention is an old one, made far worse by modern technology. This isn't to minimize the devastation modern technology has caused to our attention. Rather, it gives us a deeper and more nuanced picture of the situation. Further, seeing the bigger picture helps us identify one of the most important aspects of this issue.

Intention

Regardless of the time period, it's the passivity that makes the average human fall victim to the default state of weak attention and lethargy. On the other hand, the strength of intention is what allows one to snap out of collective hypnosis and raise life to a higher level, no matter how strong the forces of distraction are.

How intentional are you about using your attention to build an exciting, meaningful, and fulfilling life? Are you just passively existing, letting your potential go to waste? Or are you taking the reins of the immense power within you by channeling it through deep concentration? That's what James, Maslow, Willson, and Csikszentmihalyi would ask you.

These questions are timeless. However, we have the privilege of living with history's most stimulating entertainment devices in our pockets. Therefore, we must add some questions relevant to our digital age.

How intentional are you about reclaiming and protecting your attention so that you can harness its full power? Are you letting yourself be defeated and even humiliated by algorithms, becoming progressively less capable of controlling your inner resources? Or are you going to fight to preserve human dignity and beauty by strengthening your main line of both defense and attack – your attention?

For now, I leave you with these questions. In Part 2 of this essay, I'll discuss how to protect and strengthen your attention by putting timeless methods into a modern, digital context.

P.S. Proof that everything written above is true: Today, I can sense life pulsing within me stronger than usual precisely because of writing this. It's not because I'm particularly satisfied with the quality of my writing or because I'm expecting positive feedback. It is purely because of investing my attention, therefore my effort, and ultimately my life, into something interesting and worthwhile.

Tracing Sucks

CRA.MR · 25 MAR 2026 · SOURCE

Distributed traces are such a compelling idea for debugging systems. What if you could fully understand the lifecycle of an operation, end to end? Lower fidelity than runtime profiling, but enough that every major operation is mapped out, and you can understand callers. Sounds great!

In practice, it's an awful experience, you never can get the instrumentation just right, and the cost quickly becomes a burden.

What is Tracing?

If you've not implemented tracing before, let's talk about a few terms before we dive into why you should avoid it.

- **Trace ID** is the shared identifier that gets passed between services. It is a GUID.
- **Span** is a structured event within a trace (a single operation). It's unique on (`trace_id`, `span_id`). When folks say tracing they often mean *recording spans*.
- **Context** is a set of general structured attributes.

The most important part of tracing is the propagation of the Trace ID between services. In practice though this is also the most difficult problem, and there are a few reasons for that.

First, you often don't control the abstractions, so implementing this propagation is hard. For example, you might be using a platform that hides some of these details from you, and finding a way to inject the *outbound* trace ID (instrumenting the spot where it calls the network service) is already difficult. You then also have to instrument the *inbound* receiver to make sure it follows

the continuation correctly. Even more so, you're left with the question of *when* do you propagate. If you have a worker system that fans out tasks, should those pass the trace ID or not? It's entirely subjective, and it depends on the way you reason about your system, which leads us to our next concern.

OpenTelemetry attempts to do auto instrumentation for you, but that auto instrumentation is often unreliable or doesn't map to how you reason about your systems. It's so painfully bad in the JavaScript ecosystem that I now opt out of almost all auto instrumentation and instead choose to do it myself. The problem is it's very hard for vendors like us (Sentry) to ensure we have great instrumentation for every single library in the world, and OpenTelemetry's attempt to standardize it has scarred the industry with bloated interfaces and packages. Suffice to say it's an ugly mess, and manual instrumentation is the only real option you have.

Bringing us to another major point, instrumentation is just plain difficult. It requires you to have trace context everywhere with something like a thread local. You have to always have the parent span ID when you capture a new span (as well as the trace ID) to ensure things are accurately represented. In the abstract this doesn't sound too bad, and it's probably the least broken part of any tracing abstraction, but it's still a lot of complexity, especially when you go back to the challenge of getting continuation right in frameworks.

Lastly, the volume and cost of data are obscene, and the value you get out of it is hard to justify. The solution to all telemetry problems is sampling, but how do you sample a trace? Ask yourself that question. I don't have the answer, because it's also subjective and some variations are technically not feasible. Sample on the Trace ID you might think. Sounds great! What if the trace is active for weeks? How do you do that reliably? What if you're not literally just randomly sampling on trace IDs and are using a component ("enterprise customers") and need the complete trace from those customers?

There's a variety of other nitpicks and growing complexity when you try to adopt tracing, but if you're not already deep into it it's just going to overload your brain (search Span Events, Span Links, or just look at the OpenTelemetry docs to get an idea). It's exhausting.

What do we do?

I think the best answer for most folks is to simply avoid using it, at least in the traditional sense.

Think about what you want out of traces? You're mostly using them as structured logs. You're mostly debugging problems. There are a few things they push that are valuable, and you can gain the benefit of those concerns while avoiding a lot of the complexity:

1. You want semantic conventions. These add a lot of value just for consistency in your systems, and for vendors to translate meaning out of things. This is (IMO) the single most valuable thing OpenTelemetry has delivered.

2. Trace propagation is extremely valuable, but you need to ensure your system is still usable without it. That means make an effort to map requests across systems, but also create a constraint to *not require it* (this will help with ensuring you can approach sampling more cheaply).

3. Structured logs - or events - are really the primary goal here. There's no reason you shouldn't already be using them, and then tracing becomes as simple as adding a `trace_id` attribute to every log entry (and a `span_id` if you so desire). This isn't a new idea, we've had `request_id` patterns for decades!

Sentry's approach actually hedges on all of this. We decided to apply trace continuation (the best we can, it's still far from perfect) *everywhere*. Every dataset we curate has trace properties attached to it, which means any event (spans, logs, metrics, errors) can be "traced" if you will, even if you don't jump through hoops to collect spans.

That means you can completely opt out, and my advice to you is to do that, and *use logs instead*.

Flatten the logs, use semantic conventions, rely on something like our trace propagation or roll your own. You *do not need* granular caller accuracy in 99% of scenarios you will face in the real world, and the remaining ones you can put an engineer (or even an LLM) on the problem and they'll be able to sort it out.

```
// Convert an object with nested properties to foo.bar dot notation f
function flatten(obj: object, prefix = ""): Record<string, unknown> {
  const out: Record<string, unknown> = {};
  for (const [key, value] of Object.entries(obj)) {
    const path = prefix ? `${prefix}.${key}` : key;
    if (value && typeof value === "object" && !Array.isArray(value))
      Object.assign(out, flatten(value as object, path));
    } else {
      out[path] = value;
    }
```

```
  }
  return out;
}

/**
 * Structured log with flattened context.
 *
 * log('request finished', { user: { id: 1 } });
 */
export function log(
  message: string,
  context: Record<string, unknown>,
  level = "info",
) {
  console.log({
    ...flatten(context),
    level,
    message,
    ...getTraceContext(),
  });
}
```

I now follow this practice in almost all projects I spin up and it's taken the chore out of instrumentation and made it far more usable again. The products around logs are simply better (ever tried streaming traces to your console?), it's simpler for you to implement, and people understand how it works.

Anyways, this is my opinion. It is shaped by the numerous interactions I have with Sentry customers and my own experience. Traces (as in spans) are great if you can tolerate the challenges, but most of you don't need them. I certainly don't.

Engineers do get promoted for writing simple code

SEANGOEDECKE.COM RSS FEED · 26 MAR 2026 · SOURCE

It's a popular joke among software engineers that writing overcomplicated, unmaintainable code is a pathway to job security. After all, if you're the only person who can work on a system, they can't fire you. There's a related take that "nobody gets promoted for simplicity": in other words, engineers who deliver overcomplicated crap will be promoted, because their work looks more impressive to non-technical managers.

There's a grain of truth in this, of course. As I've said before, one mark of an elegant solution is that it makes the problem look easy (like how pro skiers make terrifying slopes look doable). However, I worry that some engineers take this too far. It's actually a really bad idea to over-complicate your own work. **Simple software engineering does get rewarded, and on balance will take you further in your career.**

Non-technical managers are not stupid

The main reason for this is exactly the cynical point above: **most managers are non-technical and cannot judge the difficulty of technical work**. Of course, in the absence of anything better, managers will treat visible complexity as a mark of difficulty. But they usually do have something better to go on: actual *results*.

Compare two new engineers: one who writes easy-looking simple code, and one who writes hard-looking complex code. When they're each assigned a task, the simple engineer will quickly solve it and move onto the next thing. The complex engineer will take longer to solve it, encounter more bugs, and generally be busier. At this point, their manager might prefer the complex engineer. But what about the next task, or the task after that? Pretty soon the simple engineer will outstrip the complex one. In a year's time, the simple engineer will have a much longer list of successful projects, and a reputation for delivering with minimal fuss. Managers pay *a lot* of attention to engineers with a reputation like that.

Of course, the complex engineer might try a variety of clever tricks to avoid their fate. One common strategy is to hand off the complex work to other engineers to maintain, so the original engineer never has to suffer the consequences of their own design. Alternatively, the complex engineer might try and argue that they've been given the hardest problems, so of course each problem has taken longer[1].

I don't think these tricks fool most managers. For one, if you're constantly handing your bad work off to other engineers, they will complain about you, and multiple independent complaints add up quickly. Non-technical managers are also typically primed to think that engineers are overcomplicating their work anyway. Your manager might initially nod along, but they'll go away and quietly run it by their own trusted engineers.

Simple work means you can ship projects

Most managers do not care about the engineering, they care about the *feature*. Software engineers who can ship features smoothly will be rewarded, and being able to write simple code is a strong predictor of being able to ship.

Does writing simple code really help you ship? You might think that simple code is harder to write than complicated code (which is true), and that therefore it's easier to rapidly deliver something overcomplicated to "ship a feature". I haven't seen this be true in practice. The ability to write simple code is usually **the ability to understand the system well enough to see where a new change most neatly fits**. This is *hard*, but it doesn't take a long time - if you're familiar with the system, you'll often see at a glance where the elegant place to slot in a new feature is. So good engineers can often deliver simple code at least as quick as complicated code. And of course, complicated code is slow to actually get working, harder to change, and so on. All of those things make it more awkward to ship[2].

When managers are talking to each other, they'll sometimes make a kind of backhanded compliment about an engineer: "they're *so* smart, but…". Typically the "but" here is "but they don't have any business sense", or "but they get too wrapped up in technical problems", or anything that means "but they can't ship". Engineers who love to write complicated code get described like this a lot.

Final thoughts

"You should write complicated code to avoid being replaced" is an example of a kind of mistake that many smart people make: obsessing over second-order effects and forgetting first-order effects. Second-order effects - the way some actions can cause downstream consequences that are the opposite of their original goals - are fun to think about. But they are usually swamped by first-order effects. Yes, doing bad work can make you more difficult to replace, in some ways. But that's outweighed by the negative consequences from the fact that *you are doing bad work*.

It's often a smart political tactic to make your work sound slightly more complicated than it really is. Otherwise you risk falling into the "you made it look easy, therefore we didn't need to pay you so much" trap. But it's foolish to actually do unnecessarily complicated work. Software is hard enough as it is.

1. This can be a surprisingly effective strategy, because of the tempting circular logic here: if an engineer has been given the hardest problems, it's probably because they're a hotshot, which means you can trust their assessment of how difficult their problems are, which means…

↩

2. If you're thinking of counter-examples - complex code that shipped smoothly without major followup issues - I suspect this code was probably simple *enough*.

↩

The agentic passive voice.

IRRATIONAL EXUBERANCE · 29 MAR 2026 · SOURCE

At some point, you will have learned about the passive voice, where the actor in a sentence is unclear. For example, my software didn't compile. That's a good example of the passive voice. However, you might not know the full set of rules, because here are some sentences in the passive voice that you might not recognize:

- Claude made an error in my writeup.
- ChatGPT messed up the commitment.
- Gemini didn't write tests.

You might think those are active sentences, but those are in fact examples of the *agentic passive voice*. The rule here is: whenever the actor in a sentence is a model, then it's a passive sentence. I'm sorry if your grammar instructor never taught you this rule, but this is just the way it works now.

This is an important grammatical distinction to make, because I'm increasingly seeing folks say that Claude made a mistake, without recognizing that they're writing unclear, nearly ungrammatical sentences that their grade-school teacher would reject. So please, aspire higher. Write in the active voice, avoiding all passive variants.