

Open in app ↗

Sign up

Sign in

Medium

Search



AI

+

Generative Ai Tools

+

Future Of Work

+

Elephants, Goldfish and the New Golden Age of Software Engineering

27 min read · Apr 27, 2026



Dave Rensin

Follow



Listen



Share



Introduction

As I write this in April 2026, it is already clear that Generative AI tools, and Large Language Models (LLMs) specifically, are completely reshaping how we think about and execute our work. This isn't just a software engineering shift; it touches every

single profession. This paper is my account of the past year — the journey my teams and I took, the tough lessons we picked up, and my personal take on what's coming next.

Whether you are already working with AI coding tools or have only just started to poke around with an AI chatbot, I hope you will take something away from this that will help you think about the role of AI in your work and how you might use it to make yourself more productive and valuable.

Regardless of your starting point, let me say two things in a loud and clear voice:

1. These observations reflect my personal perspective. Since many paths lead to success with these tools, you should explore widely and find what works best for you. Your mileage may vary.
2. Given the breakneck pace of change, another year might make these words obsolete. Please read with a healthy sense of skepticism.

How This Paper is Organized

This paper has three main parts.

In Part 1, I argue that most of us are using our AI tools — including basic chatbots — wrong. I will propose a different approach that I hope will be valuable no matter who you are or what you do for a living. This section is for everyone and it sets up what follows.

In Part 2, I will go into deeper detail about how you can successfully apply the basic principles from Part 1 to the task of writing software. In particular I will introduce a model of working we call the Elephant-Goldfish Model. If you are already using AI coding tools you will recognize a lot of what's in this section, though I do think there will be a few nuggets that might be new to you. You don't need to be a software engineer to read Part 2 but the way of working described there comes directly from my experience writing software.

Finally, in Part 3, I will venture (perhaps unwisely, given the rapid rate of change) into the arena of prognostication. Things are evolving at a breakneck pace in the

industry but I think there are already some emerging patterns that we can use to prepare for whichever specific future we ultimately live in.

Part 1: It's a Tool, Not a Toy

10x productivity improvements aren't just for engineers using AI models; they are for you, too. A simple change to the way you use the tools now will help you go faster today, much *much* faster tomorrow, and improve the quality of your output. This isn't a clickbait headline; it is a simple truth

Let's Buy a Car!

Suppose you wanted to buy a new car. How would you approach that problem?

Most of us (including me not so long ago) would think hard about the things that seemed important to us about this car, open our favorite LLM tool in research mode, and tell it:

"I want to buy a new car. Here's everything I've thought of about the car I want to buy. <INSERT WHATEVER YOU THOUGHT OF HERE> Go scour the Internet and produce the world's most comprehensive and impressive report on which cars I should consider and which I should buy. Take as long as you want."

Twenty minutes later we'd have a very impressive looking report that made a sensible sounding recommendation. We would nod approvingly to ourselves about how clever we were and start browsing the Internet for the best deals on our new AI-approved car choice.

Also, we did it wrong.

We used a very powerful tool like a toy and got a toy result. So.. How *should* we use AI to buy a car?

Step 1: Victory Loves Preparation

Our hypothetical quest to buy the right car was destined to fail because we didn't adequately prepare the machine to succeed on our behalf. *We asked it to act as our researcher rather than our interrogator.*

Here's the prompt we *should* have given our beleaguered AI assistant:

“I want to buy a new car. Since this will cost me a lot of money it’s very important that I make an informed and well reasoned decision. I have thought about my needs for this new car but I am certain that I am missing critical aspects. I will tell you what I’ve come up with so far and your job is to keep asking me clarifying questions until I tell you to stop. Our job together is to explore all of the important edges of this decision. Please continue to question my assumptions and force me to clarify important points until I tell you we are done. Here is what I’ve come up with so far: <INSERT EARLIER PARAGRAPH HERE>”

Spend the next 15–20 minutes answering these questions and challenges from the AI. Occasionally ask the model “what do you think?” When you disagree, argue with it. Just be sure to *argue to learn, not to win*.

***NOTE:** If the model starts to say unhelpful things like “That’s a great point! You’re a genius!” then it’s fallen into its sycophantic spiral. (An unavoidable consequence of all these models being told to be “friendly and helpful.”) You can usually break the model out of this loop by reminding it what it means to be actually helpful:*

“When you agree with me you are not being helpful. You are most helpful when you challenge my thinking and force me to think about the edges of my problem”

This is my go-to response whenever the model starts becoming too agreeable.

After 15–20 minutes you will have a much fuller description of the ideal car you want to buy. Your initial paragraph will have expanded into 2–3 pages. If you stopped here and gave this new description to the AI you will already get a much improved result. But, we can do even better.

You are now ready for step 2.

Step 2: Trust But Verify

A thorough description of the problem is only the first half of the recipe to get a great answer. The next step is to make sure that what the model tells you is both useful and verifiably true.

Continuing your current chat session, give it this prompt:

“Thank you for your questions but I think our description is now complete. I will be giving this description to a researcher with whom I have never worked before. I need to know that I can trust the quality of the report after they have produced it. I would like you to now adopt the persona of an expert skeptical fact checker. I would like to have a conversation with you about all the things that I should find in their report that would give me confidence that their work was thorough and correct. What sorts of links? Citations? Etc.. I will only get to ask them to perform this research once and I will have no opportunity to ask follow up questions, so I need to be very clear with them about how I will evaluate their work. Let’s discuss this now. I will let you know when we are done creating our acceptance criteria.”

Now you repeat the same interview process from the previous step except this time you are focusing exclusively on how you will grade the quality of the final output. As before, you should expect to spend 15–20 minutes on this step.

When you are done you should have another 2–3 pages of text that represent the grading rubric for the report. Now that you have more complete descriptions of both the question and the evaluation criteria you are ready for the final step.

Step 3: Profit

Open a **new** session with your LLM and put it in some kind of deep research mode. Then tell it:

*“I have a topic I need you to think deeply about. Use any tools at your disposal to answer this question thoroughly. I will give you both the research topic **and** the specific criteria I will use to evaluate the quality of your final report. Your goal is to produce a final output that thoroughly answers the question and easily passes all the evaluation criteria.*

Here is my description of the topic I want you to research:

— — — — —

<INSERT TEXT FROM STEP 1>

— — — — —

Here are the criteria by which I will evaluate the quality of your output:

— — — — —

<INSERT TEXT FROM STEP 2>

— — — — — -”

Now you can let the model run its deep research for as long as it takes and you can feel confident that the quality of the output you get will be much better than trying to do this with a one-shot prompt.

A Word About Judgment

None of our typical day jobs are to buy cars (although if that is your day job then you have a very cool job!). Whatever our job function (Sales, Marketing, Finance, etc) we were all hired to do the same thing — exercise excellent judgment in order to create more value for our business.

The company decides our value by the quality and impact of the judgment we consistently demonstrate.

If we are early in our careers our employer has wagered that we have the ability to rapidly acquire good judgment in our particular work domain. If we are later in our careers then the company has hired us to exercise the judgment we have already accumulated for its benefit and to help others acquire judgment of their own.

These models (at least at the time of this writing) are not capable of this kind of judgment — though they are *excellent* at generating sequences of words that superficially resemble it. Do not make the mistake of outsourcing your judgment to the machine.

Accelerating Future You

The steps in this section will help you leverage your judgment into much more productivity very quickly. But... The real value actually compounds over time..

SAVE the full description, acceptance criteria, and final output that you generated in these steps someplace that you can easily find. (A shared drive, a NotebookLM, etc). Over time you will accumulate an incredible amount of reasoning and history that you can leverage in future conversations with the model for some other problem.

Note: There is no need to store the full discussions; these three artifacts are sufficient, the rest turns out to be mostly token noise

Why do this? Now, you can talk to your past judgment or the collective wisdom of the team!

“Why did we make this decision this way? What was our thought process about X?” etc..

How much easier will it be to onboard new team members or to test future outputs against past reasoning? How much easier will it be to reconstruct the trajectory of a project, a team, or a long running program?

If you start doing this today the speedup and quality will compound rapidly into the future. *Future You* will be unrecognizably more productive than *Current You*.

Part 2: Elephants and Goldfish

In the Summer of 2025 I had a business problem at work and wrote (with a lot of help from AI) a small tool to help me solve it. That little tool turned out to be useful for my teammates and — surprisingly to me — a lot of other people. Other engineers started showing up and asking to contribute new features to the tool (which I gratefully accepted) and within a few months more than 100,000 Googlers were using the tool regularly for similarly shaped problems of their own.

I don't think it would have been possible to go this quickly, accept contributions safely from dozens of people, and still meet our security, privacy and trust bars without using AI coding assistants. (In my case I use Google Antigravity and Gemini).

Working this way is amazing but It doesn't take long until you quickly realize something terrifying.

If we aren't careful, we aren't just writing code faster. We are **mass-producing our mistakes**.

If you are using AI to go fast, but you aren't maintaining a high signal-to-noise ratio, you are actually going backwards in productivity. If the rate of bad code being generated is greater than the rate of productive code, your project will eventually collapse under the weight of its own unreadable, unmaintainable slop.

Over the intervening weeks and months, we found a better way, but first we had to accept that the era of the human-only “coder” is rapidly coming to an end.

The Three “Aha” Moments

The size of the problem really hit me during a particularly frustrating afternoon. I was using AI to implement a new feature. We were an hour into the session, deep in the weeds, when the model simply crashed.

When I started a new session, it was a blank slate. It had no memory of the previous hour of work. Re-bootstrapping the machine into our highly complex task by feeding it hundreds of lines of code was tedious, token-heavy, and incredibly frustrating.

That was **Aha Moment #1**: If I had simply *maintained a clearly written document explaining my intent and the current state of our progress*, re-bootstrapping would have been a 30-second operation. (“Read this doc. Read the code it references. Tell me where we are, and let's continue.”)

So, I tried that. I got my context back. Then, I asked the AI to finish the implementation.

Problem solved! Except.. It immediately ran off the rails, sprinting in directions I never asked for, hallucinating APIs, and making architectural choices that made zero sense for our system.

That was **Aha Moment #2**: AI models are eager to please, which means they will happily sprint off a cliff if you don't *build very strict guardrails*.

But the real revelation came a few weeks later when I was starting an entirely new feature. I realized I didn't need to feed the AI the source code to get it to understand the system. I could just feed it the design docs.

That led to **Aha Moment #3**, which is simply this:

```
sizeof(docs) << sizeof(code)
```

It is much much cheaper, faster, and more reliable to give an AI context through plain English design documents than it is to force it to read and interpret tens (or hundreds) of thousands of lines of raw code.

The Vanishing Code

To understand why this is so important, we have to look at how we build software today.

Historically, design judgments — the “why” and “how” of a system — sit in two places: in the design docs, and in the code itself. When a human writes code, they leave a trail of micro-decisions behind in the logic.

But if an AI is writing the code, the human is no longer making those micro-decisions. If we don't force all of those design judgments to “shift left” into the design document *before* the code is written, the resulting system will be completely incomprehensible to the human who is ultimately responsible for it.

We are currently existing in a liminal state. We still need code because we still need rigor. But 100% human-reviewed code is a vanishing intermediate step.

It seems to me that we will wind up in one of two places; (1) the models produce so much code that it becomes a practical impossibility for humans to review it all, or (2) the models just start producing binaries directly and skip the code altogether. It doesn't matter which of these turns out to be true, though. Whether in a blizzard or a binary, the code is going to become opaque to us and when the code becomes opaque, the only artifact that matters is the design.

Design is the new code.

Introducing The Elephant-Goldfish Model (EGM)

Over time the team developed a simple and highly effective operating framework I call the **Elephant-Goldfish Model**.

It's based on a very straightforward metaphor about how we build context and how we test it:

- **The Elephant:** “An elephant never forgets.” This is the heavily-prompted, context-rich AI session, and the resulting Design Document it helps you write. The Elephant holds all the institutional knowledge, the micro-decisions, the arguments you had with the model, and the entire history of the feature.
- **The Goldfish:** A brand new, completely fresh AI session with zero memory. It only knows exactly what is put right in front of it.

Our simple process for every new feature is this: **Feed the Elephant; test it against the Goldfish.**

Phase 1: Growing the Elephant (No Code Yet)

The goal here is to co-design the feature and the evaluation criteria *before a single line of code is written*.

Step 1: Context Loading

Open a fresh session with your AI tool and point it at the relevant design docs or source tree. Ask it to give you a high-level description of what it learned.

Why? Because if it misunderstands your system on minute one, it's going to write terrible code on minute ten. Correct its understanding immediately.

If you are starting a new project from scratch you don't need to do this step because there is no history or context into which to bootstrap the AI.

Step 2: The “No Code” Rule

Engineers love to write code. AI loves to write code. You have to actively fight this urge. Give the AI this prompt:

“I do not want you to create code. We are not going to create code. Resist your impulse to create code. Instead, we are going to have a design discussion. I am about to describe a feature. I want you to ask me clarifying questions and challenge my assumptions. Do not just accept what I say.”

Give the AI a description of what you are trying to accomplish and then talk to it like a human. Have a back-and-forth conversation for 20–30 minutes. If the AI hallucinates, point it to the file that corrects it. Really argue with the AI. These models are excellent interrogators so make it really push your thinking about the description and edges of the problem you are trying to solve.

This step is functionally the same as the back-and-forth interview process described earlier when we were buying our hypothetical car. The point of this step is to create a thorough description of the problem we are trying to solve.

The crisper you can be about what you are trying to accomplish, the better the output will eventually be.

Step 3: The Sycophant Challenge

AI models are trained to be helpful, which often makes them insufferable sycophants. If the AI stops asking hard questions and just says, “Great idea, Dave!”, you have to force it back into critic mode. Tell it: *“You are not being helpful. Your highest and best use is to challenge my thinking.”*

Pro-tip: *Simply asking the AI, “Why do you think that?” is your best friend. It will often snap the model out of a hallucination loop.*

Step 4: The First Draft Technical Proposal

Once you are satisfied that you have a complete description of your problem it’s time to start debating potential solutions.

Get Dave Rensin’s stories in your inbox

Join Medium for free to get updates from this writer.

Enter your email

[Subscribe](#)☐

Remember me for faster sign in

Ask the AI to propose an implementation in prose and block diagrams. Short blocks of pseudocode are fine, but demand written text. Argue with the AI until you settle on a solid approach.

It's **very** important that you ask the AI to propose the first design. That will tell you if it understands your problem well.

If, on the other hand, you propose an initial design and ask the AI to react to it, you will almost certainly miss some blind spots. There are lots of ways to ask the model to do this, but this is the prompt I most commonly use:

“Based on your understanding of the codebase and what we discussed, I would like you to give me a first draft proposal of a technical implementation to actually make this feature happen. I’m not looking for code. I want prose from you that demonstrates your understanding of my system. Short blocks of pseudocode are fine if you think that will help, but I would strongly prefer clearly written text and block diagrams.”

The first draft is going to be (at least a little) wrong. Ask clarifying questions (e.g., “Why did you choose that pattern? I thought the system worked like X...”). Argue with the AI until you settle on a solid technical approach.

When the model hallucinates something about your system, point it to the file that will correct its understanding. “Nope. I don’t think it works that way. Please read `auth.py`. (or `AUTH.md` or whatever)”

This back and forth creating the technical proposal might take hours. Some of my sessions with Gemini have lasted 2–3 days while we debate important decisions.

Do not rush this back-and-forth; you will be sorry if you do.

Phase 2: Teaching the Elephant

Now we create the markdown document that will serve as the absolute source of

truth and the guardrails for the code. Do not ask the AI to write the whole doc at once because the current models have token limits on the outputs they will produce at once. Build it iteratively in the same chat session.

This is the structure and order of the sections that we have found to be most useful. Do **not** try to one-shot this document. Write each section separately.

1. **The Problem:** Have the AI write a plain English description of the business problem. 3–5 sentences will probably be enough. It just needs to be enough that a casual reader can understand what you are trying to accomplish.
2. **The Technical Plan:** Have it write a jargon-light description of the big components and how they fit together. Include a block diagram of the major components, if that makes sense. Your goal in this section is explanatory prose.
3. **Alternatives:** Document the ideas you considered but ruled out during Phase 1. Make sure all the major decisions that were considered but rejected get some consideration in this section. These will become important guardrails against future hallucinations and arguments.
4. **Detailed Implementation:** This will be the longest section of this document. It should contain a highly detailed step-by-step implementation plan. Demand that the AI enumerate *every single file* that will be created or changed, and the rationale for why. This is **critical**. Doing this will help keep the AI on the rails when it comes to implementation.

Save this file to your tree. This is the first artifact for your new feature and is now your new source code.

Phase 3: The Goldfish Protocol

How do you know if your design doc is actually good, or if it's just relying on the context built up in your "Elephant" AI session?

You test it against a Goldfish.

Step 5: The Comprehension Test

Start a brand new, empty AI session (the Goldfish). Give it your markdown doc and

say:

“Read this document and the files it references. Tell me what it’s trying to accomplish, and how my system currently works as it relates to this feature.”

If the Goldfish cannot explain your system based *only* on that document and the files it references, your doc is missing context. Add the details and repeat until it passes. Do not skip this loop. Documents that pass the Goldfish test become important protections against slop.

Step 6: The Critic Review

Start another Goldfish session:

“Assume the role of an expert technical reviewer. Read this design doc and all the files it references. Tell me all the things I missed, all the faulty assumptions, all the edge cases I’m missing, and things I should have considered but did not. Every mistake and ambiguity you find makes you more helpful and useful.”

Anecdotally, I find about 30% of its suggestions are highly valuable — which is more than enough to make this worthwhile. Update the file accordingly.

Rinse and repeat until the recommendations and questions start falling consistently into the category of “nit pick.”

Step 7: Implementation Readiness

Start a new Goldfish and ask it:

“You are an experienced software engineer experienced with our codebase. Read this document and the files it references and tell me: Does it absolutely have all the information you would require to successfully implement this feature in your first pass?”

If it asks questions, answer them in the doc. Rinse and repeat until you are satisfied that there aren’t important missing items or unresolved ambiguities.

Finally, show it to a real human for review and approval. (If you’re working solo then this may not be an option for you. Use your best judgment.)

Phase 4: Implementation

Step 8: Coding with Guardrails

Now, and only now, do you write code. You hand the finalized Design Doc to the AI and say:

“Read this design doc and the files it references. Implement the feature as described. Follow the plan exactly.”

Because the doc explicitly lists every file and every change, the AI will produce much higher quality code. If during the implementation session(s) the AI spins out or crashes, you just open a new session, hand it the design doc, tell it to read the doc and all the files referenced, and you are immediately back in business.

Step 9: The “Mean” Code Review

When the code is done, feed it back to the AI. Tell it to find every way the code is terrible. I actually say:

“I have a strong intuition that this code is of poor quality. Please tear it to shreds and tell me all the ways it sucks.” (yes, that’s my actual prompt)

Tell it to flag any place where you’ve gone 10 lines without a comment. *Demand strict readability.*

Rinse and repeat until the critiques are trivial (or disappear).

Bootstrapping Your Reality (Or, “Feeding the Elephant Peanuts and Hay”)

When I explain this to other engineers the first thing they say is, “Dave, that’s nice for new projects, but my codebase is millions of lines long. I don’t have the time to write design docs for all of it. I can’t train an elephant for a monolith.”

Actually, you can, and it takes about a week, (almost) no matter how large your code base.

The trick is understanding how to feed the elephant. To strain the analogy further, if you try to feed an AI the entire jungle of your source code all at once, it’s going to choke. It loses context, gets confused, and hallucinates. Instead, you have to feed it peanuts and hay — small, highly nutritious, compressed bites of context. (Yup. Still

an engineer. Terrible puns and all.. But at least this is how you know this was written by a human.)

Those “peanuts and hay” are README.md files. You just have to make the AI do the heavy lifting to harvest them through a recursive summarization process:

1. **Start at the leaves** of your source tree. Assign each developer a set of directories.
2. **Generate the Peanuts:** Have them point the AI at the directory and say: *“Read the files in this directory and produce a new file named **README.md**. This file should (a) explain the purpose of this directory and the files contained in it and (b) enumerate each file in the directory and a short description of its function”*
3. **Human Verification:** The AI will be about 50% wrong in the leaf directories because it has only the code to rely on. The assigned engineer will spend 5–10 minutes fixing the contents of the file.
4. **Roll up the Hay:** Once the leaves are done, move up one level:
“Please read all the README.md files in all the subdirectories below me, then read the code in just this directory and create a file here named README.md. Using that information this file should (a) explain the purpose of this directory and the files contained in it and (b) enumerate each file in the directory and a short description of its function”
5. **Continue upward** — level by level — until you hit the root of the project.

When you are done, you will have an excellent hierarchy of context — the “peanuts and hay”. You can open a fresh session at the root of your project, feed the Elephant *just* the READMEs in the tree, and it will understand your codebase (nearly) instantly and use much less of the context window than if it had to read each of the source files.

It won't be perfect, but it will understand things surprisingly well. It will be able to reliably pick out the major subsystems and design patterns — which will be a huge boost to you when you go to make changes.

How Long Do I Need READMEs?: *The READMEs only need to exist until you have*

enough design docs such that all of the code files are referenced (and lightly explained) in at least one doc. Once you have 100% file coverage you can delete the READMEs (or not, your call). If you are using the Elephant-Goldfish model process for a new application and are rigorous about your design docs, you will never need these READMEs at all.

From there, you just start writing design docs for your new features using the Elephant-Goldfish process above.

Bonus Feature: Instant Onboarding

There is an incredible secondary benefit to having your entire source tree documented in a hierarchy of README.md files and/or robust Design Docs: it makes onboarding new engineers super low friction.

If you have a new engineer joining the team, here is your new onboarding protocol:

1. Create a new NotebookLM notebook (or similar tool of your choice).
2. Dump your entire hierarchy of README.md files and existing Design Docs into it.
3. In the system instructions, tell the notebook what specific part of the system the new team member is going to focus on.
4. Hand the notebook to the engineer.

Tell them their first job is to use that notebook to learn enough about the system to write an excellent design doc (using the above process) for their first feature.

When they can write a good doc that passes the Goldfish test, you can feel comfortable that they know enough to productively contribute to the codebase — or at least not waste your time during code reviews. No more weeks of shadowing or trying to reverse-engineer opaque legacy code.

It is a common event for someone to show up in my inbox to suggest a new feature for our tool. Pre-2025 we would just add it to the backlog and eventually get to it. With this approach, however, I just ask them “*do you have any background in writing code?*”

Lots of product managers, program managers, and others have at least a little coding experience. For those people we prepare a new notebook and ask them to use it to write the design doc for the feature they want.

A few days later (normally) a new design doc shows up in my review queue. Most of the time it's ~70%-80% correct, so I review it and offer suggestions on how to improve it. After one or two more turns it will have met the bar for our quality.

Once they have met that bar, I can feel comfortable that the code they produce using the AI will eventually make it into our source tree. It will all be reviewed as normal, of course, but it won't be slop — which means it won't be a waste of time.

Come for the Speed; Stay for the Comprehension

We are moving into a world where typing syntax into an IDE is no longer the primary job of a software engineer. The job is intent. The job is architecture. The job is design judgment.

If we lean entirely on AI to write code without forcing those design judgments into human-readable, rigorously tested documents, we will lose the ability to confidently put our names on the work we are supposed to stand behind. At the end of the day, a human will be held responsible for any action taken by an agent and we can't reasonably be expected to take on that risk if we don't really understand the system the agent builds at our direction.

But if we treat Design as the New Code, we align our human need for comprehension with the AI's capacity for speed. It is the ultimate de-risking exercise.

Part 3: Some Probably Wrong Predictions

“It's tough to make predictions, especially about the future.”

— Yogi Berra

Things are moving so quickly in this space that it's probably a mistake to make any predictions, but that's what I'm going to do. I think there are some outlines of durable patterns we can use to navigate whatever comes, but I could be wrong, so please read with a very healthy skepticism.

We Are All Managers Now

In the current order of things if you manage employees you are known as a “people manager.” If you do not then you are known as an “individual contributor (IC).” It seems to me that this distinction is going to get awfully blurry *this year*.

As the tools and models become more capable we will all start managing some number of agents who are doing work on our behalf. If you are already an experienced people manager then you should adapt pretty easily to this. If you have never managed other humans, however, this will require you to learn some new (and highly useful) skills.

On the surface it might seem like managing humans should be very different than managing AI agents. I would like to argue, however, that they overlap more than you might think.

AI Agents Aren't Humans — But Sometimes They Rhyme

The purpose of management is to leverage your judgment and expertise far beyond the work you can personally accomplish.

If you are a bad manager, you tell your employees precisely the steps they should take to do every little job. We call this micro-managing and anyone who has ever experienced it knows that it doesn't scale and seldom gets you the result you want.

A good manager, on the other hand, tells their employees what *outcome* they are looking for and gives the people guardrails and guidelines. They clearly explain the scope of authority and degrees of freedom under which the employee can operate as well as the conditions under which they want the employee to stop and seek guidance.

You are doing a good job as a manager when you can successfully look away from

your employee for a while and when you look back things are in good order. The longer you can look away, the better job you are doing of growing your employee. As the durations get longer, you can add more people since the overhead cost per person is dropping.

I don't think you have to squint very hard to see how this translates to managing AI agents. Sure, you don't have to do career planning, but I would propose that there are many more similarities than not.

Some Practical Advice

If you think this premise is not entirely insane and you are currently an IC, what should you do?

Take a basic people management class. Right now. Really.

That may seem like unusual advice, but hear me out...

The purpose of management is to magnify the impact of your judgement behind what you can personally accomplish in a day. You are a good manager when you can successfully look away from your employee for a period of time and feel confident that things will be in good order when you check back in.

The evidence that you are doing a good job setting your employees up for success is that the duration of how long you can successfully look away increases over time. As that happens the amount of personal overhead imposed per employee shrinks. That means you can take on more employees successfully without being consumed by interrupts and burning out.

Learning to successfully scale your judgement across independent entities acting on your behalf is a skill that takes training and practice. The age of AI agent management is upon us and you need to start building this ability now.

Many companies have internal training for first time managers that specifically cover delegation, coordination, and orchestration. Sign up for one of those classes. If your current employer doesn't offer a class like that then buy a book, watch some YouTube videos, take an online course, but do *something*.

The other piece of advice I would give is to *ramp slowly into the number of agents you are trying to manage at one time*.

When we first start as people managers we usually get a small team of 3–5 people. The reason for this is because managing people means learning how to set goals, incentives, and rules of the road so that you don't get overwhelmed by interrupts.

This rule of thumb seems like a good first target to aim at as you learn how to smoothly scale your judgment through agents. It's a little like a new exercise; start slow, lock in the correct form, and scale from there.

Sustained Concentration is a Muscle That Has to Be Developed

AI is steadily absorbing some of the toil in our work such as status reports and update meetings. This is genuinely good news. But it raises the question: what do we do with the reclaimed time?

The answer is the work we've never had enough time for. Hard thinking. Real design. The kind of judgment that can't be delegated to a colleague or to a model.

The problem is that sustained concentration is a skill most of us haven't had the chance to develop. Not because we lack the capacity, but because modern society likes to interrupt us with things competing for our attention.

Building concentration is like building any other kind of endurance. If you're not used to long stretches of focused thought, starting with two hours is like running a 10K on day one. A better entry point is 30 to 45 minutes every work day to concentrate hard on a problem. At the start of each week, add another 5 to 10 minutes. In a few months, you will build up your mental endurance to concentrate for 2 hours (and more) at a time.

This is a muscle everyone needs to build. You don't have to sit in monastic silence; feel free to use that time to debate with your AI tool about some problem you are working on. The point is to think/debate your way to some level of mental fatigue. A tired brain signals growth the same way as a tired bicep.

If AI takes away lots of our toil this just means we'll all get to work less, right? Nope.

You might hope that AI will make your work so efficient that you'll only need to work a few hours a day. However, history shows that when a tool makes work easier, we don't necessarily work less — we just end up doing more because the cost of that effort has dropped. This is a long observed phenomenon known as Jevons' Paradox.

A Final Thought

Speaking of William Jevons and his famous paradox, I sometimes hear concerns from students and younger colleagues that AI will mean they can't have long and fruitful careers. It may be true that our day-to-day tasks will change somewhat, but that doesn't mean we will be without purpose or value. This sort of change has been true for as long as I have been in the workforce.

Your value is in the judgment you acquire and successfully apply. The better taste you develop in your specific domain and the more you learn how to scale it, the more in demand you will be. This has always been true and I suspect it will continue to always be true.

These AIs are already very powerful tools. Learn to use them like a craftsman and you will find no shortage of demand for your skills. Will it be hard? Sure, but it'll be hard like a hill, not hard like a mountain; so just start walking.

I would like to thank the following colleagues for their invaluable assistance in bringing this paper to life — Elise Ewing for pushing me write these things down in the first place, and Amber Huffman and Partha Ranganathan for patiently reading early drafts and offering thoughtful editorial suggestions.

AI

+

Generative Ai Tools

+

Future Of Work

+





Follow

Written by Dave Rensin

680 followers · 3 following

Distinguished Engineer - Google (<https://dkr.bio>)



