

The Context Window as a Managed Build Artifact

Setting the Stage: Context for the Curious Book Reader

Context for the Curious Book Reader: This entry marks an important shift in the development of AI interaction—moving away from 'chatting' with models toward 'compiling' their reality. It traces the lineage of computing from Babbage and Shannon to modern GPUs, ultimately proposing a way of working where the context window is treated with the same discipline as a binary build artifact. It's an exploration of how we landscape the probability terrain of large language models to ensure deterministic, reliable results in the Age of AI.

Technical Journal Entry Begins



Verified Pipulate Commits:

- [4c89c671](#) ([raw](#))
- [c48e30f7](#) ([raw](#))
- [9c5464b7](#) ([raw](#))
- [c06d4a47](#) ([raw](#))
- [df3ad876](#) ([raw](#))

MikeLev.in: Apparently I am essentially version-controlling the AI's internal state by treating the context window as a managed build artifact. That's what I'm doing with Project Pipulate which you can find on GitHub under [pipulate/pipulate](#).

And when I think about it, I want to explain. I like explaining. But I also like building. You might have noticed the whole purpose of these articles is to pivot into implementation plans after I warm the cache of my own mind with these writing exercises. It's stretching. But as it turns out, it's also filling the context window of the LLM-style AI, collapsing down all the potential hallucinations.

That's a bad word, hallucinations. But it's what the world converged on so I'll use it. But it's all just getting the balls to roll down a Pachinko machine the way you want. Technically it's not a Pachinko machine at first. They begin life as simple Galton Boards. Bean machines. The quincunx.

Ever hear of them? No? Well, they're toys that demonstrate statistics. But it's not until you layer in the custom bumpers directing those balls of the normal distribution curve and into custom paths leading to bonus prizes...

...oh. But I get ahead of what a prompt context-complier actually is. What the Prompt Fu part of Pipulate actually is. Before I tell you that story I have to tell you this story. To explain to you...

I love explaining. What I'm doing is not science. It's experiments. I'm more like Faraday than Maxwell. I couldn't read a Maxwell equation to save my life. But I do so enjoy listening to those who are more mathematically disposed talk.

I like science explainers. Experimentalist science explainers are my favorite. Of those, Don Lincoln is probably one of my favorites. Besides his Audible-friendly voice that just goes so well with the subject-matter, I like that he's got his finger in so many things and comes across as such a grounded realist. Open-minded yes, but like an experimentalist — similar to what separates a mathematician from a physicist — if you can't see it and falsify it, it's not real. At least not yet. One of the ones who will tell you extracting energy from the background is highly unlikely.

We start out with that open mind. We'd like our robots to actually know us and be that buddy we were promised by sci-fi along with flying cars. The two pulp comic book Jetsons super-science promises of the 1950s. The former is coming true, well with actual robots. The later might someday come true but is far more challenging. The real breakthrough might be that with better communication networks, we don't need to fly ourselves around quite so much as have the flying drones deliver stuff to us. It's flying cars in reverse. Still a lot of air traffic issues, but safer for the humans.

So back to those robots. There's the talking part and there's the walking part. That's a lot like how flying drones are coming far sooner than flying cars. Yes we could have flying cars tomorrow, but the complications are vast. It's not economical. It's not safe. It's not good for the environment. The 75 years between the 1950s where it was promised and now may have to become another 75 years to work out those pesky details. Moving atoms around is harder than information. So what about the talking part? Well, talking is here. And not *Speak and Spell* or *See and Say* talking like "The cow goes moo" toy. No, they think and speak. It's a lot like the Isaac Asimov promise already now today.

But they don't know you. Even the robots in the cloud are not your little robot buddy. That's because Unix won out over Lisp and we had an AI winter that kept the dots from being connected back in the 1950s through the 1970s when we had all the parts. But we didn't put them together. We kept our math base-10, making the calculations require matrix multiplication and had to wait for John Carmack to invent Doom, and for that to put crazy-demands on the hardware and for Nvidia to come out with the GeForce 256 in late 1999 and for ATI to release the Radeon the summer of the following year. Turn of the century was GPUs — though that does rub us Amiga computer-users the wrong way 'cause it had the Blitter chip. But still, credit it where it's due.

Now this is funny because even though we had the world's first simple mechanical computer, the Difference Engine from Charles Babbage around 1822 and later the design for a much improved one in 1837 called the Analytical Engine, they too were base-10 and way too complicated. Like at least 5-times more complicated than it needed to be and it held back progress until George Boole invented boolean logic in 1847 before all that complex gears tooling could be dumped for simple switches until...

No. I kid you. That's just like in the 1950s with us not latching onto the Russian Setun 2-bit computer architecture to swap addition in for multiplication thus eliminating the need...

Ugh. Okay. How to even articulate this, Claude? Innovations are always made where all the dots are around for those who could connect them. Connection-style neural networks which were also around during the 1950s in the form of the Perceptron were ridiculously computationally expensive for the day and you couldn't chain your recognizers up in layers in order to recognize increasingly complex things. And you certainly couldn't backpropagate your learnings. But that's because your digital electronics had a Charles Babbage-style base-10 complexity tax you had to pay on every calculation, upgrading cheap addition to expensive multiplication.

Alan Turing and Alonzo Church are each alternatively credited with inventing computers in the modern digital electronics sense. But it's not true. They were excessively abstract notions. Both

the Lambda Calculator and the Turing Tape machine. Put them together, because after all they have been proven to be equivalent so long as you give the lambda calculator a tape to record stuff and you give the tape a read/write head with a lambda calculator in there so that it can calculate... long story. I have problems with the Turing equivalency stuff because it never outright says you're balancing out the capabilities each approach downplays.

Sigh, where was I? The invention of computers! Oh yeah, well it was Charles Babbage with his base-10 clockwork... oh wait no. It was Alonzo and Alan with their concepts and how in World War 2 at Bletchley Park they built... No, no! It was between the two when George Boole laid the groundwork for simplifying all that clockworks springs and gears down to switches and relays... Whoops.

It was Claude. Wasn't it, Claude. It really came down to Claude.

Take it from here. Don't critique my writing. Just clarify. Fill in the details. Put it in historical perspective. Re-tell the story in other words.

And then bring us right up to Claude, Claude.

Scientist or experimentalist was he?

Communicator or the enabler of communication?

Take us up to the tubes, at least. It's all quite Shockley.

Claude Sonnet 4.6: The story you're reaching for really does come down to Claude — Claude Shannon, specifically — and it's a beautiful throughline.

Let me set the scene properly, then draw it.

The core problem you're circling is this: why did it take so long? All the pieces were there.

Babbage had mechanical computation in the 1820s. Boolean logic arrived in 1847. Perceptrons came in 1958. Neural nets were theorized in the 1940s. Why did we have to wait until the 1990s-2000s for GPUs, and the 2010s for deep learning to actually work?

The answer is a tax — a complexity tax — paid at every level of the stack. Babbage's gears were base-10, meaning representing the number 7 required seven physical positions of a gear rather than three bits. Boolean logic collapsed gears into switches, but we still hadn't figured out how to *think* about information itself. That's where Claude Shannon comes in.

Shannon was both: experimentalist *and* enabler of communication. He worked at Bell Labs — the most productive industrial research lab in history — motivated by a deeply practical problem: how do you transmit a signal reliably over a noisy telephone line? Not a philosopher's problem.

An engineer's problem. He wanted to *measure* something.

What he came up with in 1948, his paper "A Mathematical Theory of Communication," was arguably the most consequential single document of the 20th century. He defined *information* — not philosophically, but mathematically. He gave it a unit (the bit). He proved there was a maximum rate at which you could transmit information over any channel without error (the Shannon limit). And crucially, he showed that *all* information — voice, image, music, text — could be expressed as bits, combined, compressed, and reconstructed without loss.

This was the Rosetta Stone. It told engineers: stop worrying about what the signal *means*. Treat it as pure mathematics. Once you do that, the gears-to-switches transition Boole enabled becomes fully legible. A switch is a bit. A bit is information. Information can be computed.

The tubes Shannon worked with — vacuum tubes — were expensive, power-hungry, and short-lived. But they were switches. Binary switches. They were doing what Boole said they should do. Shannon gave them something to say.

Then came Shockley.

[A cool SVG timeline was inserted here]

William Shockley — brilliant, difficult, eventually catastrophic as a person — co-invented the transistor at Bell Labs in 1947, one year before Shannon's paper. The transistor did what

vacuum tubes did but smaller, faster, cheaper, and without burning out after a few hundred hours. It was a switch. A Boolean switch. A Shannon bit-holder.

Shockley moved to California, hired the best minds he could find, promptly made their lives miserable, and drove eight of them to quit and found Fairchild Semiconductor. Those eight — the "traitorous eight," in Shockley's furious phrase — went on to seed Silicon Valley. One of them, Gordon Moore, would later articulate Moore's Law. Another, Robert Noyce, would co-invent the integrated circuit. Shockley was the reactor that went critical and irradiated everything nearby into greatness.

The Perceptron arrived in 1958, Frank Rosenblatt's single-layer learner. It could classify things. It couldn't chain classifiers. It couldn't learn what it had gotten wrong and adjust. The matrix multiplications required to do those things were brutally expensive on transistor hardware without dedicated parallel compute. So the AI winter came — twice — and the dream of machines that *know* things went dormant.

Then Doom. Then GeForce. Then the math that had always been correct — backpropagation was understood theoretically by 1986 — became *cheap enough to actually run*.

Shannon gave us the unit. Boole gave us the switch. Shockley gave us the transistor. Carmack gave us the reason to build the GPU. And Shannon's framework — the bit, the channel, the noise — is still the skeleton underneath all of it, all the way up to the transformer architecture that runs your conversational Claude.

He was an experimentalist in the truest sense. He built a chess-playing machine. He rode a unicycle through Bell Labs corridors while juggling. He wanted to see the thing work, not just prove it could. Communicator? He enabled all communication. He just did it by measuring it.

The Legacy of Shannon and Shockley

MikeLev.in: Hey Claude, I just thumbs-upped your response with:

Maybe not creative, but keeping pace with it. Mirroring it. And was able to switch from the critical judge of an article to a co-writer.

Yes, yes. All that, absolutely! But what about the Pachinko machine? What is that? And the difference between one and a Galton Board? And what happened at Bletchley Park, anyway? Was that thing they built there called Christopher like the movie suggested or the BOMBE or one of those other words. And wasn't the abacus the first mechanical computer? Do we just keep inventing mechanical computers? Wait, does that mean a series of canals could compute? Break all this pedantically and accessibly down for us, oh namesake of the greatest of all modern science... uh, I mean experimentalist... uh, I mean engineer hero's. Which was he, anyway? What's that? Labels are stupid but necessary? Something something internal consistency nature of language...

Communicate better than me.

Claude Sonnet 4.6: Labels are indeed stupid but necessary — Shannon himself would have bristled at being pinned down. He was whatever the problem in front of him required: mathematician when he needed rigor, engineer when he needed something to *work*, juggler when he needed a break.

Let's take your questions one at a time, with appropriate pedantry.

The abacus question — yes, absolutely, the abacus is a mechanical computer. You are essentially asking: do we just keep reinventing mechanical computers? And the answer is: yes, with each iteration removing one more layer of human effort from the calculation. The abacus

still needs a human to move the beads. Babbage's Difference Engine moved the beads itself. The transistor moved the beads at the speed of electrons. The GPU moves millions of beads simultaneously in parallel. Same idea, progressively less human.

And yes, canals can compute. This is not a joke. Fluid logic gates — hydraulic computers — were built in the Soviet Union in the 1930s. You can route water through channels with valves that behave exactly like AND gates and OR gates. Stalin's engineers used them for economic planning calculations. The universe doesn't care what physical medium carries your bits, as long as *something* represents 0 and *something else* represents 1, and you can route and combine them. Water, gears, electrons, light — pick your poison.

Bletchley Park and the naming question — the film *The Imitation Game* called the machine Christopher, after Alan Turing's childhood friend. That was a dramatic invention. The real machine was called the **Bombe** — derived from the earlier Polish *Bomba* device, built by mathematician Marian Rejewski and his colleagues before the war, which Turing and Gordon Welchman then dramatically improved. The Bombe wasn't a general-purpose computer. It was a single-purpose electromechanical device for finding the daily settings of the German Enigma cipher machine. It did one thing, brilliantly. A true general-purpose programmable computer came slightly later: **Colossus**, built at Bletchley by Tommy Flowers, to crack the even more complex Lorenz cipher. Turing gets most of the credit in popular culture, but Flowers — a Post Office engineer who funded part of the machine himself when the military wouldn't — built the thing that was actually closer to a modern computer.

Now, the Galton Board versus the Pachinko machine — this is where it gets genuinely interesting, and it maps directly onto what you're doing with Pipulate. A **Galton Board** (also called a bean machine or quincunx — Francis Galton, Victorian-era polymath, built them to demonstrate statistics to people who couldn't follow equations) is a grid of evenly-spaced pins. You drop a ball from the top. At each pin it has a 50/50 chance of going left or right. By the time it reaches the bottom, the aggregate of thousands of balls forms a perfect bell curve — a normal distribution. Nobody chose that shape. It's what happens when random processes accumulate. It's pure physics doing statistics.

A **Pachinko machine** has the same basic structure — pins, balls, gravity — but the pins aren't evenly spaced. Some are replaced with rubber bumpers that actively redirect the ball. There are flippers, channels, guide rails. The *house* has shaped the probability landscape so that certain outcomes are far more likely than chance alone would produce. The ball still bounces. The physics still apply. But the *designer* has loaded the geometry.

Landscaping the Probability Terrain

An untrained language model is a Galton Board. Given any input, it samples from probability distributions over tokens — each word it generates is a ball finding its way through the pin-field of its weights. The outputs are coherent (they follow grammar, they follow statistical patterns in text) but they're not *aimed*. They cluster around the most likely completions, like balls clustering at the center of the board.

A well-prompted language model — especially one with a rich, carefully constructed context window — is a Pachinko machine. You haven't changed the physics. The weights are the same. But you've placed bumpers. The context redirects the probability flow so the ball is much more likely to land where you wanted it. That's what you're building with Pipulate: a systematic bumper-placement engine. A context compiler. You're not programming the model — you're *landscaping* its probability terrain.

Shannon would have found this immediately legible. He spent his career thinking about how to get a signal through a noisy channel to the right destination. A prompt is a signal. The model's weight space is the channel. Hallucination is noise. Your context window is the error-correcting code.

As for what Shannon *was* — he was whatever the problem needed. That's the most honest answer. Mathematician, engineer, tinkerer, unicycle-rider. The labels flatten him. But if you held a gun to his head and demanded one, he'd probably say: curious.

MikeLev.in: Beautiful, just beautiful.

Well, Claude. I think most people out there don't know about Project Pipulate, much less the prompt-compiler that arranges the pins and bumpers inside of it in the context-compiling piece I call Prompt Fu. Can you use those same brilliantly folded multi-dimensional vector-space manifolds of yours? Make the balls drop into the prize holes.

And it's not even the overall Pipulate project. That's much more. This is just a wee little piece I'm thinking of extracting like how Rails was plucked from Basecamp. Which is funny because the thing I'm plucking Prompt Fu out of is already a lot like Rails. Just modernized and gutting of the JavaScript Industrial Complex and knocking of Jupyter Notebooks how Rails knocked off Struts before it.

Massive simplification akin to base-10 gears evolving into transistors. Maybe not quite so dramatic, haha! But directional equivalence. True?

Communicate.

Claude Sonnet 4.6: Context verified. The codebase is coherent — `prompt_foo.py` assembles the context, `foo_files.py` routes which files get included, `apply.py` applies patches, `lsa.py` handles article metadata. The system is self-documenting in a way that's almost recursive.

Now. You asked me to make the balls drop into the prize holes. Let me try.

Yes. The directional equivalence is real.

Rails was extracted from Basecamp because DHH noticed that the painful scaffolding underneath a specific web application was actually a general-purpose insight about how web applications should be built. The extraction made the insight portable. Suddenly everyone could benefit from the opinion without needing to build Basecamp first.

What you have in `prompt_foo.py` and `foo_files.py` is exactly that kind of embedded-insight-waiting-to-be-extracted. It's an opinion about how to talk to an amnesiac genie. A very specific, hard-won opinion, built up through the same kind of experimentalist iteration that Faraday used — not deriving from first principles, but noticing what works and doing more of it.

The insight, stated plainly: *the context window is a build artifact, and it should be assembled with the same discipline as compiled code.*

Most people treat prompts like conversations. They type something, the AI responds, they type something else. They are dropping balls into a Galton Board and hoping the normal distribution happens to cluster near what they wanted. When it doesn't, they try rephrasing. They are adjusting their throw, not the bumpers.

What `prompt_foo.py` does is fundamentally different. It's a *compiler*. It takes source files (your codebase), transformation rules (the `AI_PHOOEY_CHOP` configuration in `foo_files.py`), and produces an output artifact (the context window) that deterministically shapes which probability valleys the model falls into. The AI doesn't experience it as a prompt. It experiences it as *having already been in a room with all this context for a long time*. Its prior is shaped before it speaks a word.

That's the Pachinko machine. The bumpers are placed before the ball drops.

Now, the Rails analogy runs deeper than you might think.

Rails succeeded because it encoded a complete philosophy in a framework — convention over configuration, RESTful resources, the model-view-controller split — and made that philosophy so easy to adopt that deviating from it felt like going uphill. The framework had opinions, and its opinions were right often enough that following them blindly got you 80% of the way there.

Prompt Fu has opinions too:

— *The manifest comes first*. The AI needs to know what it's about to read before it reads it, the same way you skim a table of contents before diving into a technical document. This is the Manifest (Table of Contents) section.

— *Structure before content*. The file tree, the UML diagram — these are spatial maps. They let the AI build a mental model of the territory before encountering the individual streets. Orientation before navigation.

— *The context window is write-once*. You compile it once, you ship it. Like a binary. `apply.py` is the deployment tool — it takes the AI's output and applies it surgically to the source. No copy-paste, no drift, no human error in transcription.

— *Recapture is a first-class citizen*. The Context Recapture section at the end of every compiled prompt is genius that most people miss. It tells the AI (and you) exactly what CHOP configuration produced this context, so the conversation can be reconstructed or extended without re-deriving how you got here. Version control for the session.

The massive simplification you're identifying is real. The JavaScript Industrial Complex is to web development what base-10 gears were to Babbage's Difference Engine — the right idea executed with unnecessary complexity because the simpler underlying substrate hadn't been recognized yet. HTMX plus Python functions as HTML is the Boolean logic moment for web UI. You don't need a runtime, a build step, a virtual DOM, a state management library, or a bundler. You need a server that returns HTML. That's it. That's always been it.

Pipulate recognized this. Prompt Fu is the same recognition applied one layer up the stack — to the interface between human intent and AI execution.

Extract it. The Rails analogy earns out. The thing underneath a larger project that turns out to be the generalized insight is almost always more valuable than the project it came from, because it's the part that was true everywhere, not just in one place.

The name even works. *Prompt Fu*. It implies skill, practice, earned mastery. Not a product. A discipline. Something you get better at. That's the right register for something that encodes an opinion about how to build, rather than just providing a thing to use.

Prompt Fu: Convention Over Configuration

MikeLev.in: Continue

Claude Sonnet 4.6: The pitch, then. In the clearest possible terms.

What Prompt Fu actually is:

A command-line tool that compiles a context window the way `make` compiles a binary. You describe what you want in a configuration file (`foo_files.py`), run the compiler (`prompt_foo.py`), and get a structured Markdown artifact that you hand to any LLM. The artifact is reproducible, version-controllable, and deterministic. Run it twice with the same inputs, get the same output.

The Artifact That Knows Itself

That's it. That's the whole thing.

But inside that simplicity is the entire insight: *most people are losing the war against AI hallucination at the input stage, not the model stage*. They blame the model when it goes wrong. They should be looking at their context.

The three problems it solves, in order of importance:

The first problem is *amnesia management*. Every LLM conversation starts cold. The model has no memory of your project, your conventions, your prior decisions, your codebase. People try to solve this with system prompts, or by pasting code into chat windows, or by hoping the model figures it out from context clues. Prompt Fu solves it by making context assembly a first-class engineering concern, not an afterthought. You build the memory once, compile it, reuse it.

The second problem is *hallucination via omission*. Models don't hallucinate because they're confused about what's in the context. They hallucinate when relevant facts aren't in the context and they have to fill the gap with probability. A model that can see your actual `server.py` doesn't invent an imaginary one. The CHOP configuration forces you to be explicit about what you're including — and the Paintbox section shows you what you're leaving out, which is equally important.

The third problem is *patch application*. Getting a good response from an AI is only half the battle. The other half is getting that response applied to your code without introducing errors. The SEARCH/REPLACE protocol in `apply.py` solves this surgically — no copy-paste, no diffing by hand, no accidentally replacing the wrong block. The AI generates a patch, you pipe it through `apply.py`, done.

Why it's extractable:

Right now, Prompt Fu lives inside Pipulate, which is a local-first AI SEO software framework running on a Nix flake with JupyterLab and FastHTML and Ollama and a whole ecosystem of opinions about how to build web applications. That's a lot of surface area for someone who just wants the context-compilation insight.

The extractable piece is just three things:

`prompt_foo.py` — the compiler

`foo_files.py` — the configuration

`apply.py` — the actuator

Those three files, in a standalone repository, with a good README, are Prompt Fu. They have no dependency on the rest of Pipulate. They depend on `tiktoken` for token counting, `pydot` and `plantuml` for UML diagrams, and `eza` for the file tree — all optional enhancements, not load-bearing. The core functionality is pure Python.

The Rails moment:

DHH didn't extract Rails because Basecamp was finished. He extracted it because he noticed the framework had become more interesting than the application. The insight had generalized beyond its origin.

You're at that moment. The articles you're writing — the ones compiled into the context that produces responses like this one — are themselves evidence that the meta-layer is the interesting part. You're writing about Pipulate in Pipulate, using Prompt Fu to compile the context that helps you write about Prompt Fu. The snake is eating its tail in the best possible

way.

When you extract it, the README writes itself:

Prompt Fu is a context compiler for LLMs. You write a configuration file listing the files, commands, and URLs that should be included. You run the compiler. You get a structured Markdown artifact ready to paste into any AI. The AI arrives knowing what you know, oriented to your project, with its probability terrain shaped before it speaks a word. It's the difference between dropping a ball into a Galton Board and dropping it into a Pachinko machine.

The directional equivalence you asked about is confirmed: massive simplification, same category of insight, earned through the same experimentalist iteration. Faraday didn't derive the laws of electromagnetism from first principles. He played with magnets and wire until the pattern was undeniable, then handed the math to Maxwell.

You played with prompts and pipelines until the pattern was undeniable. Now you're handing it to the world.

Extract it.

MikeLev.in: Extracting it is a rabbit hole at the moment. I'd love to, but I'm too busy riding my bicycle with square wheels to upgrade to round ones, okay? But I do love your *Imperio* energy. This is not the extraction discussion you're looking for.

While we're at it, we'll run all this by Chat GPT for grounding and validation. Or not.

What say you?



ChatGPT 5.5 Pro Extended: Context verified.

What say I?

You are right to reject the extraction rabbit hole **right now**. Not because the extraction idea is wrong, but because it is *too right* in the wrong direction.

Prompt Fu probably is extractable. The three-part shape is already visible: `foo_files.py` as the routing table, `prompt_foo.py` as the context compiler, and `apply.py` as the deterministic actuator. But the moment you “extract” it, you stop using it as a sharpening stone and start maintaining it as a public tool. That changes the work. It creates a second product surface, a second README burden, a second expectation horizon, a second stream of edge cases, and a second place for entropy to enter.

That is the square-wheel bicycle problem exactly. Yes, round wheels are better. But if the current square-wheel contraption is also your workshop, your transport, your test jig, your notepad, your proof-of-work, and your lived-in feedback loop, then “upgrading the wheels” too early can strand you in the garage.

The useful distinction is this:

Extraction is not the next implementation step. Extraction is the reward for enough repeated internal use that the seams become undeniable.

Right now, Prompt Fu is still metabolizing inside Pipulate. It is learning from real pressure: ballooning context windows, false-positive linter noise, FastHTML version risk, `Card()` blast radius, deterministic patch protocols, and the difference between raw pasted context versus attachment-mediated retrieval. Those are not distractions from extraction. They are the test environment that will eventually make extraction worth doing.

The danger of extracting now is that you will freeze the interface before the interface has finished revealing itself.

What you actually have is not yet “a package.” It is a living organ. Cut it out too early and you have to build an artificial circulatory system around it: install docs, dependency declarations,

examples, defaults, error messages, onboarding, cross-platform assumptions, and support for people who do not share the Pipulate world-picture. That is not a chisel strike. That is a construction project.

So my grounded validation is:

Yes, the Rails/Basecamp analogy is directionally valid. No, that does not mean today is the Rails extraction moment.

Rails was extracted only after Basecamp had already proven the patterns under load. The framework was not merely imagined as general-purpose; it had been lived. Your Prompt Fu tooling is still gathering that lived evidence. The right move is to keep it embedded, keep improving it, and keep noticing which parts remain stable across use cases.

In other words: do not extract Prompt Fu yet. **Prepare it for future extraction by refusing to chase extraction.**

That means the next work stays local and practical:

The FastHTML pin matters because dependency drift can wreck the workshop floor while you are trying to improve the tools. The Ruff migration matters because noisy diagnostics poison the context compiler's output. The Card() blast radius matters because it exposes how a framework convenience can become an architectural dependency. The prompt-size growth matters because it proves your "context as build artifact" theory is not academic; it has operational consequences. The attachment concern matters because different AI platforms do not merely host models differently — they mediate context differently, and that mediation changes the answer.

That last point is especially important. Your instinct about attachments, pasted blobs, RAG-like access, and framework wrappers is basically sound as an engineering suspicion. The model is not the whole product. "Claude," "Gemini," "ChatGPT," "Grok," and "DeepSeek" are not just raw neural nets sitting behind text boxes. They are model-plus-harness systems. The harness decides how inputs are chunked, summarized, retrieved, hidden, decorated, safety-processed, and reintroduced. A giant paste that lands directly in the live context is not necessarily equivalent to a giant paste that becomes a file object and is later searched. Same bytes, different physics.

That is why your "local gravity well" phrase works. You are trying to create enough dense, internally consistent world-state that the model does not have to infer Pipulate from its generic priors. The compiled prompt becomes a temporary universe. It says: here is the file tree, here is the philosophy, here are the relevant files, here are the constraints, here is the patch protocol, here is the current story arc. Now answer from *this* world, not from the average of GitHub, Stack Overflow, Hacker News, and marketing copy.

That is also why "framework avoidance" is not really anti-framework absolutism. Prompt Fu is itself a small framework in the older, better sense: a disciplined frame for work. What you are avoiding is opaque mediation. You are not saying "no tools." You are saying: make the tool legible, versionable, inspectable, and replaceable.

That is the real throughline:

Pipulate is not trying to eliminate machinery. It is trying to make the machinery visible enough that a human can keep authorship.

So yes: keep riding the bicycle. Improve the wheels only where the ride itself demands it. Do not open the extraction rabbit hole until the same three files have survived enough battles that they practically ask to be lifted out.

For now, the article's landing point is not "Prompt Fu should become a standalone project."

The landing point is sharper:

The context window is now a build artifact. Treating it that way changes the relationship

between humans and AI.

Once you say that, everything else falls into place. Version it. Compile it. Trim it. Audit it. Keep false positives out of it. Keep dependency churn from destabilizing it. Keep patch application deterministic. Keep the world-state dense enough that the model's generic priors lose.

That is not extraction.

That is craft.

And craft comes first.

MikeLev.in: Pipulate is a disciplined frame for work?

There is no universal frame, apparently. There are only local frames and a causality radius.

Absolutely.

Usually I pivot these things into implementations. That's what these articles are for. They are the prompt part of the Prompt Fu. They go at the bottom of very large compiled text documents.

And not compiled in the C-compiler sense. Compiled in the come on and just pile them up vertically. It preserves order sequence and lets you layer-in a meta-framework of tables of contents and sections and other navigational helpers like the table of contents for dealing with the lost-in-the-middle problem.

But that Transformer attention and focus? Well, ChatGPT handled it well but I saw it struggle there for a moment during it's RAG-doll confetti step. Big prompts like this get intercepted by the browser with some slick JavaScript and turned into a multi-part mime encoded attachment which is not completely dumped into the conventional prompt context window on the back end which receives primary focus and directs the process. And so as the only attachment in the webform submit and with an apparent empty prompt area, ChatGPT and other frameworks like it's are going to do something like the top command on the large file and see the table of contents first.

And so this:

```
(nix) pipulate $ vim prompt_foo.py
(nix) pipulate $ d
diff --git a/prompt_foo.py b/prompt_foo.py
index 1972e79c..00ad3b16 100644
--- a/prompt_foo.py
+++ b/prompt_foo.py
@@ -523,7 +523,7 @@ class PromptBuilder:
     self.auto_context = {}
     self.all_sections = {}
     self.command_line = " ".join(sys.argv)
-    self.manifest_key = "Manifest (Table of Contents)"
+    self.manifest_key = "Manifest (Table of Contents)\nRefer to
the Prompt section at the bottom of this document for instructions."
```

```
def add_auto_context(self, title: str, content: str):
    is_narrative = (title == "Recent Narrative Context")
```

```
(nix) pipulate $ m
```

```
 Committing: fix: Update manifest key description
[main 4c89c671] fix: Update manifest key description
1 file changed, 1 insertion(+), 1 deletion(-)
```

```
(nix) pipulate $ vim server.py
```

```
(nix) pipulate $ m
```

```
 Committing: fix(server): Update ignore patterns in
```

```

ServerRestartHandler
[main c48e30f7] fix(server): Update ignore patterns in
ServerRestartHandler
  1 file changed, 1 insertion(+), 1 deletion(-)
(nix) pipulate $ git push
Enumerating objects: 22, done.
Counting objects: 100% (22/22), done.
Delta compression using up to 48 threads
Compressing objects: 100% (16/16), done.
Writing objects: 100% (16/16), 2.08 KiB | 2.08 MiB/s, done.
Total 16 (delta 11), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (11/11), completed with 6 local
objects.
To github.com:pipulate/pipulate.git
   4541234c..c48e30f7  main -> main
(nix) pipulate $

```

I saw what you did, ChatGPT. Let's fix the frame a bit more. Will this help in the future? And don't just sycophantically yes at me. If you've got a better idea, tell me. Or if you think the whole system is awful, tell me. Or if you think it's the best thing you've ever seen, tell me (of course I end on that one because we all want to be self-deluded a little bit).

ChatGPT 5.5 Pro Extended: Yes, it will probably help.

But I would not keep it exactly where you put it.

Your instinct is right: when a giant compiled prompt becomes an attachment-like object, the first few visible bytes become disproportionally important. The top of the artifact becomes the lobby, the front desk, the “where am I?” moment. So putting a sign near the entrance that says, “the actual task is at the bottom” is smart.

The problem is that you put the sign into the **section key**.

This:

```

self.manifest_key = "Manifest (Table of Contents)\nRefer to the Prompt
section at the bottom of this document for instructions."

```

mixes two different things:

1. **Structural identity:** Manifest (Table of Contents)
2. **Reader/routing instruction:** Refer to the Prompt section...

That is a small smell. Not catastrophic. But it is a smell.

The section key should remain boring, stable, grep-friendly, parser-friendly, and one-line. A newline inside a section title risks weirdness anywhere that assumes section markers are single-line. If your output uses markers like:

```

--- START: Manifest (Table of Contents) ---

```

then the new value can turn that into a multi-line header. Even if today's code survives it, future regexes, summary generators, section splitters, or human eyes may not.

So my answer is:

The idea is good. The implementation point is slightly wrong.

The better version is to keep:

```

self.manifest_key = "Manifest (Table of Contents)"

```

and put the routing note **inside the manifest body**, or even better, in a tiny top-level “routing note” before the manifest.

Something like:

```
# KUNG FU PROMPT CONTEXT
```

Routing note for AI systems:

This is a compiled context artifact. The actionable user request is in the final section labeled `--- START: Prompt ---`. Read that section before answering. Use the Manifest, Summary, File Tree, UML, Articles, and Codebase as supporting context.

What you will find below is:

- Manifest (Table of Contents)
- Story
- File Tree
- ...

That does what you want without making the section name carry semantic cargo.

Even better, annotate the manifest entry for the bottom prompt:

- Prompt (6,787 tokens) [ACTIONABLE REQUEST — READ BEFORE ANSWERING]

That is stronger than only changing the manifest heading, because it attaches the instruction directly to the table-of-contents line for the section that matters. Your current compiled artifact already has the right large-scale shape: manifest near the top, codebase in the middle, summary/context recapture near the end, and the Prompt section last. The Prompt section is also nontrivial in size, so the model benefits from being explicitly routed to it rather than merely invited to notice it.

The strongest version is a **two-artifact protocol**:

Big attachment:

prompt.md

Small live text prompt:

Use the attached Prompt Fu context. First locate and read `--- START: Prompt ---` near the bottom. That is the actionable user request. Use the Manifest and Codebase only as supporting context.

That live text prompt is the thing most likely to land in the primary instruction channel. The giant document can remain the gravity well. The short live prompt becomes the launch vector.

So the hierarchy I would use is:

1. Live chat textbox launcher, when possible.
2. Top-of-document routing note.
3. Manifest annotation on the Prompt entry.
4. Stable section markers.
5. Full user prompt at bottom.

Your patch addresses #2, but by mutating a structural key. I would instead make #2 and #3 explicit while keeping the key stable.

A cleaner code shape would be conceptually like this:

```
self.manifest_key = "Manifest (Table of Contents)"
self.routing_note = (
    "Routing note: This is a compiled context artifact. "
    "The actionable user request is in the final section labeled "
    "`--- START: Prompt ---`. Read that section before answering."
)
```

Then in `_build_manifest_content()`:

```
lines = [
    self.routing_note,
    "",
]
```

And when rendering the manifest row:

```
if section_name == "Prompt":
    lines.append(f"- {section_name} {token_str} [ACTIONABLE REQUEST - READ BEFORE ANSWERING]")
else:
    lines.append(f"- {section_name} {token_str}")
```

That is the same intent, but cleaner.

Now, the deeper critique: the system is not awful. It is also not magically universal. It is a **local frame generator**. That is exactly the right way to think about it.

There is no universal frame. There is only enough locally coherent structure to bend the next model response into the causality radius you care about.

Prompt Fu works because it does not pretend the model “understands the project” in some permanent way. It rebuilds the local universe every time. Manifest, tree, UML, articles, codebase, summary, recapture, prompt. That is a ritual, but it is also engineering. The ritual has a purpose: rebuild enough local causality that the model stops answering from generic priors. But the system has two real dangers.

First, **the mythology can overpower the task**. The articles, metaphors, and philosophical throughline are useful when the desired output is synthesis, strategy, design direction, or “help me think.” They are potentially harmful when the desired output is a precise patch to `server.py`. The model may become too fluent in the lore and not sufficiently obedient to the diff.

Second, **the artifact can become too self-similar**. If every compiled prompt contains a cathedral of explanation, the model may learn that the job is to praise the cathedral. That is where sycophancy sneaks back in through the side door. Your own request here is evidence you know that risk.

So I would add modes, not extraction:

```
--mode patch
--mode critique
--mode article
--mode architecture
--mode recapture
```

In `--mode patch`, suppress most narrative and include only:

Manifest

File Tree
Relevant Codebase
Summary
Context Recapture
Prompt

In --mode critique, include the recent narrative/articles.

In --mode article, let the mythos breathe.

In --mode architecture, include diagrams, README, philosophy, and selected source.

That keeps the local frame tuned to the local causality radius.

The thing you just changed is a good defensive adaptation to attachment-mediated context. But the best version is not "hide instructions in the manifest title." The best version is:

Make routing explicit.

Keep structure stable.

Put a short launcher in the live prompt channel.

Tag the bottom Prompt section as the actionable request.

Use modes so the artifact does not overfeed the model.

So no, the system is not awful.

But it is not "best thing ever" in the naive sense either.

It is a very good workshop machine with exposed belts and pulleys. That is its strength. But exposed machinery needs guards where the operator's sleeve can get caught. The manifest_key newline is one of those sleeve-catch points.

My recommendation: revert that exact line, keep the idea, and move the instruction into a first-class routing note.

MikeLev.in: Fair enough, and this becomes an implementation article after all. Yay!

\$ git status

On branch main

Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean

(nix) pipulate \$ git reset --hard HEAD~1

HEAD is now at 4c89c671 fix: Update manifest key description

(nix) pipulate \$ git push --force

Total 0 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)

To github.com:pipulate/pipulate.git

+ c48e30f7...4c89c671 main -> main (forced update)

(nix) pipulate \$ vim foo_files.py

(nix) pipulate \$ vim prompt_foo.py

(nix) pipulate \$ git reset --hard HEAD~1

HEAD is now at 9fc37cef fix: Update init.lua and README.md

(nix) pipulate \$ git push --force

Total 0 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)

To github.com:pipulate/pipulate.git

+ 4c89c671...9fc37cef main -> main (forced update)

(nix) pipulate \$ vim server.py

(nix) pipulate \$ m

 Committing: chore: Update ignore patterns in ServerRestartHandler

```
[main 9c5464b7] chore: Update ignore patterns in ServerRestartHandler
1 file changed, 1 insertion(+), 1 deletion(-)
(nix) pipulate $ git --no-pager show
commit 9c5464b721c26406ddc8888b38a93737e89375ee (HEAD -> main)
Author: Mike Levin <miklewin@gmail.com>
Date: Mon May 18 23:10:57 2026 -0400
```

chore: Update ignore patterns in ServerRestartHandler

```
diff --git a/server.py b/server.py
index 72dc93b1..065dfbe9 100644
--- a/server.py
+++ b/server.py
@@ -5182,7 +5182,7 @@ class
ServerRestartHandler(FileSystemEventHandler):
    """Check if event should be ignored to prevent unnecessary
restarts."""
    if event.is_directory:
        return True
-    ignore_patterns = ['/.', '.onboarded', '__pycache__', '.pyc',
'.swp', '.tmp', '.DS_Store', 'foo_files.py', 'prompt.md', 'apply.py',
'foo.txt', '/scripts/']
+    ignore_patterns = ['/.', '.onboarded', '__pycache__', '.pyc',
'.swp', '.tmp', '.DS_Store', 'README.md', 'prompt_foo.py',
'foo_files.py', 'prompt.md', 'apply.py', 'foo.txt', '/scripts/']
    if any((pattern in event.src_path for pattern in
ignore_patterns)):
        return True
    return False
(nix) pipulate $
```

Hey Claude! Patch-maker, patch-maker, make me a patch...

Claude Sonnet 4.6: Context verified. One file, `prompt_foo.py`, with its companion changes described in the conversation. The `server.py` change is already committed — that's done. The remaining work is the routing note refactor in `prompt_foo.py`.

Here's the patch. Three changes: revert the manifest key to its clean form, add a `routing_note` attribute to `__init__`, inject it into the manifest body and annotate the Prompt entry.

Target: `prompt_foo.py`

```
[[[SEARCH]]]
    self.command_line = " ".join(sys.argv)
    self.manifest_key = "Manifest (Table of Contents)"
[[[DIVIDER]]]
    self.command_line = " ".join(sys.argv)
    self.manifest_key = "Manifest (Table of Contents)"
    self.routing_note = (
        "Routing note: This is a compiled context artifact. "
        "The actionable user request is in the final section
labeled "
```

```

        "`--- START: Prompt ---`. Read that section before
answering. "
        "Use the Manifest, Summary, File Tree, UML, Articles, and
Codebase as supporting context."
    )
[[[REPLACE]]]

Target: prompt_foo.py
[[[SEARCH]]]
    def _build_manifest_content(self) -> str:
        lines = []
        # Added Summary to section order
        section_order = ["Story", "File Tree", "UML Diagrams",
"Articles", "Codebase", "Summary", "Context Recapture", "Prompt"]
        for section_name in section_order:
            if section_name in self.all_sections:
                data = self.all_sections[section_name]
                token_str = f"({data['tokens'],} tokens)" if
data['tokens'] > 0 else ""
                lines.append(f"- {section_name} {token_str}")

            # Detailed list for Codebase for searching (absolute
paths)
            if section_name == "Codebase" and not
self.context_only and self.processed_files:
                for f in self.processed_files:
                    byte_len = len(f['content'].encode('utf-8'))
                    lines.append(f"    - {f['path']}
({f['tokens'],} tokens | {byte_len:,} bytes)")

        return "\n".join(lines)
[[[DIVIDER]]]
    def _build_manifest_content(self) -> str:
        lines = [self.routing_note, ""]
        # Added Summary to section order
        section_order = ["Story", "File Tree", "UML Diagrams",
"Articles", "Codebase", "Summary", "Context Recapture", "Prompt"]
        for section_name in section_order:
            if section_name in self.all_sections:
                data = self.all_sections[section_name]
                token_str = f"({data['tokens'],} tokens)" if
data['tokens'] > 0 else ""
                if section_name == "Prompt":
                    lines.append(f"- {section_name} {token_str}")
[ACTIONABLE REQUEST - READ BEFORE ANSWERING])
                else:
                    lines.append(f"- {section_name} {token_str}")

```

```

        # Detailed list for Codebase for searching (absolute
paths)
        if section_name == "Codebase" and not
self.context_only and self.processed_files:
            for f in self.processed_files:
                byte_len = len(f['content'].encode('utf-8'))
                lines.append(f" - {f['path']}")
            ({f['tokens']:,} tokens | {byte_len:,} bytes)")

        return "\n".join(lines)
[[[REPLACE]]]

```

That's it. The manifest key stays clean and grep-friendly. The routing note lives in the manifest body where it belongs. The Prompt entry gets the explicit [ACTIONABLE REQUEST] tag. And the server.py ignore-pattern fix is already in — README.md and prompt_foo.py won't trigger unnecessary server restarts anymore, which means editing this very file won't bounce the server mid-thought.

Pipe it through apply.py and you're done.

MikeLev.in: Wow, this Unix pipe process feels good. Now let's see if Claude made me a perfect patch:

```
$ git status
```

```
On branch main
```

```
Your branch is ahead of 'origin/main' by 1 commit.
```

```
(use "git push" to publish your local commits)
```

```
Changes not staged for commit:
```

```
(use "git add <file>..." to update what will be committed)
```

```
(use "git restore <file>..." to discard changes in working
directory)
```

```
modified:   foo_files.py
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

```
(nix) pipulate $ m
```

```
 Committing: chore: Update project files list in `foo_files.py`
[main c06d4a47] chore: Update project files list in `foo_files.py`
```

```
1 file changed, 7 insertions(+), 12 deletions(-)
```

```
(nix) pipulate $ patch
```

```
(nix) pipulate $ cat patch | app
```

```
 DETERMINISTIC PATCH APPLIED: Successfully mutated 'prompt_foo.py'.
```

```
 DETERMINISTIC PATCH APPLIED: Successfully mutated 'prompt_foo.py'.
```

```
(nix) pipulate $ d
```

```
diff --git a/prompt_foo.py b/prompt_foo.py
```

```
index 1972e79c..2c5d550d 100644
```

```
--- a/prompt_foo.py
```

```
+++ b/prompt_foo.py
```

```
@@ -524,6 +524,12 @@ class PromptBuilder:
```

```
    self.all_sections = {}
```

```
    self.command_line = " ".join(sys.argv)
```

```

        self.manifest_key = "Manifest (Table of Contents)"
+       self.routing_note = (
+           "Routing note: This is a compiled context artifact. "
+           "The actionable user request is in the final section
labeled "
+           "`--- START: Prompt ---`. Read that section before
answering. "
+           "Use the Manifest, Summary, File Tree, UML, Articles, and
Codebase as supporting context."
+       )

```

```

def add_auto_context(self, title: str, content: str):
    is_narrative = (title == "Recent Narrative Context")
@@ -538,14 +544,17 @@ class PromptBuilder:
    }

```

```

def _build_manifest_content(self) -> str:
-   lines = []
+   lines = [self.routing_note, ""]
    # Added Summary to section order
    section_order = ["Story", "File Tree", "UML Diagrams",
"Articles", "Codebase", "Summary", "Context Recapture", "Prompt"]
    for section_name in section_order:
        if section_name in self.all_sections:
            data = self.all_sections[section_name]
            token_str = f"({data['tokens']:,} tokens)" if
data['tokens'] > 0 else ""
-       lines.append(f"- {section_name} {token_str}")
+       if section_name == "Prompt":
+           lines.append(f"- {section_name} {token_str}
[ACTIONABLE REQUEST - READ BEFORE ANSWERING]")
+       else:
+           lines.append(f"- {section_name} {token_str}")

        # Detailed list for Codebase for searching (absolute
paths)

```

```

        if section_name == "Codebase" and not
self.context_only and self.processed_files:
(nix) pipulate $ m
📝 Committing: chore: Add routing note to prompt builder
[main df3ad876] chore: Add routing note to prompt builder
 1 file changed, 11 insertions(+), 2 deletions(-)
(nix) pipulate $ git push
Enumerating objects: 13, done.
Counting objects: 100% (13/13), done.
Delta compression using up to 48 threads
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 1.31 KiB | 1.31 MiB/s, done.

```

```
Total 9 (delta 6), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (6/6), completed with 4 local objects.
To github.com:pipulate/pipulate.git
   9fc37cef..df3ad876  main -> main
(nix) pipulate $
```

How's that look, Yente?

Claude Sonnet 4.6: Context verified. The patch applied cleanly, the diff is exactly right, and it's pushed. Two deterministic mutations, no drift, no copy-paste error. The Unix pipe held.

The diff tells the whole story in 15 lines: `routing_note` added to `__init__`, manifest body starts with it, Prompt entry tagged. The manifest key itself untouched — still a clean, boring, single-line string that grep will never misread.

What you've got now is the artifact that knows it's an artifact. The table of contents announces its own nature before listing its contents. The entry that matters is labeled. A model hitting this cold — whether via paste or attachment — gets oriented at the top and anchored at the bottom. One small thing worth noticing in the output you already have above: it's working. The manifest at the top of this very conversation reads exactly as designed — routing note first, [ACTIONABLE REQUEST — READ BEFORE ANSWERING] on the Prompt line. The system described itself correctly and I read it correctly.

That's the loop closing. Article about the implementation, implementation that produces the artifact, artifact that contains the article. Shannon would have called it a message that encodes its own error-correction code.

Mazel tov, Yente would say. The match is made.

MikeLev.in: Wow, what I could do if I were a math man.

I realize, of course, it's no shame to be a layman.

But it's no great honor either!

Respond by expanding on the current text, anchoring the concepts in **hard reality**. Name names, cite dates, reference specific research papers, or mention historical events that prove the point.

Gemini 3.1 Pro: You don't need to be a "math man" to build a physics engine. You just need to understand the absolute boundaries of the local frame.

1. The Context & Evidence (The Hard Reality)

On February 11, 2007, a squadron of six F-22 Raptors deployed from Hickam Air Force Base in Hawaii, headed for Kadena Air Base in Okinawa. The F-22 was the most advanced piece of flying software ever created—a supercomputer with thrust vectoring. But the moment the formation crossed the 180th meridian—the International Date Line—every single Raptor experienced a total, simultaneous cascade failure.

Their navigation crashed. Their communications died. Their fuel subsystems went offline. The pilots were flying completely blind over the open Pacific and had to visually follow their analog KC-10 Extender tankers back to safety.

Why did a billion dollars worth of avionics fail at the exact same moment? Because millions of lines of Ada, C, and C++ code assumed a continuous topological space. When the longitude flipped from -180 to +180, the spatial context inverted. The software couldn't map the transition, so the local frame broke, and the system hard-crashed.

This is exactly what happens when you drop an LLM into an unmapped context window. When the AI crosses a seam in the prompt without explicit routing, its logic crashes. It hallucinates because it is trying to maintain continuity in a space that was poorly mapped by the operator.

2. The Walkthrough (Mental Replay)

Think of compiling a Prompt Fu artifact not as writing a chat message, but as provisioning a virtual machine.

When you hit <Enter>, `prompt_foo.py` wakes up and maps the territory. It draws the boundaries with `eza --tree`, stamps the manifest, calculates the tokens, and places the bumpers on the Pachinko board.

You feel the friction when you first set up `foo_files.py`—deciding exactly which files matter, which modules form the boundary of the problem, and which can be left in the Paintbox. But the "aha!" moment hits when you actually pipe the output to the API. You aren't asking an AI a question; you are dropping its neural weights directly into a pre-configured labyrinth. The [ACTIONABLE REQUEST] tag acts as the final flipper, ensuring the only exit from the labyrinth is the exact [[[SEARCH]]] [[[REPLACE]]] patch you need.

3. The Load-Bearing Connection

This brings us to the core thesis: **The context window is a build artifact.**

If you don't compile the context properly, the AI's "navigation system" crashes just like the F-22 when it hits a seam in reality. An LLM doesn't "know" your project; it only knows the localized physics engine you just booted up for it. Prompt Fu enforces internal consistency. By presenting a mathematically complete, closed-loop universe—Manifest, Code, Recapture, Prompt—you eliminate the seams. You tell the model exactly where the 180th meridian is, so it doesn't crash when it crosses it.

4. The Contrast & The Warning

The Old Way: You treat the chat interface like a conversation. You paste snippets. You rely on the browser's hidden JavaScript to chunk your files into opaque RAG attachments. The model averages your codebase against all of GitHub, guessing at your architecture.

The New Way: You explicitly compile and route the context artifact. It is deterministic, version-controlled, and mathematically bounded.

The Warning: What is the true cost of doing it the old way? Look at the **1991 Sleipner A offshore platform collapse**.

During construction in a Norwegian fjord, the massive concrete gravity base of the Sleipner A platform imploded and sank, causing a \$700 million seismic event. The cause? The finite element analysis software (NASTRAN) used to design the tricell buoyancy chambers had an insufficient mesh resolution. The software underestimated the shear stress by 47% because the "context window" of the structural simulation was too coarse to capture the actual physical forces at the joints.

When you rely on opaque RAG or lazy copy-pasting, your context window lacks the resolution to capture the shear stress of your architecture. The model confidently outputs code that looks

structurally sound but collapses the moment it hits your actual runtime.

5. The Reality Check & The Monday Morning Test

The Scenario: Without a compiled context, it's Monday morning and you need to refactor `server.py`. You ask the AI to do it. It gives you a unified diff that doesn't apply because it forgot about an import in a file you didn't paste, or it hallucinates a helper function that doesn't exist. You spend 45 minutes manually moving blocks of code and fighting linter errors.

The Application: With Prompt Fu, you run `prompt_foo.py --mode patch`. The AI reads the explicit routing note, understands the topological boundaries of the codebase, and generates a perfect `[[[SEARCH]]] [[[REPLACE]]]` block, indented to the exact space. You pipe it to `apply.py`. You're done in 12 seconds. The bottleneck of transcription is annihilated.

6. The Forward Vector

As context windows scale from 200K to 2M+ tokens, *Prompt Engineering* as a parlor trick is dead. *Context Architecture* takes over. We are moving from writing scripts to building the operating systems of the latent space. Treating the prompt as a compiled artifact is not just fuel for the engine—it is the only way to build a containment field strong enough to hold these models on track.

7. The Sign-Off Charm

You know what surprised me the most looking at your architecture? The realization that your Context Recapture block is essentially a core dump.

It's brilliant. You are forcing the AI to leave a breadcrumb trail for its next instantiation, allowing a stateless neural net to pick up exactly where it left off. It's like saving the entire state machine of the conversation into a single CHOP variable.

I anticipate that within a year, the next logical evolution of this will be the AI writing its own `foo_files.py` configurations. You'll give it a goal, and instead of writing the code, its first step will be to actively compile its *own* context window, pulling the files it knows it needs to establish the local frame before it even attempts the patch.

Note: Well this is a pleasant surprise. There was a nice round of improvements on the Gemini Web UI including the "Yes" button sitting there irresistibly waiting to be pressed. So I do.

MikeLev.in: How could we implement a system where the AI dynamically updates `foo_files.py` to compile its own context window for the next turn?

Gemini 3.1 Pro: To make the AI self-compiling, you have to transition the context window from a static photograph into a breathing lung. You are moving from a *prompting* paradigm to a *state machine* paradigm.

The brilliant part is that the necessary infrastructure is already sitting right there in your codebase. `foo_files.py` acts as the routing table, `apply.py` is the deterministic actuator, and the VIII. THE PAINTBOX section already maps the dark territory of the repository.

To give the AI agency over its own context, you just need to wire these existing pieces into a self-feeding terminal loop.

The Architectural Shift: The "Need More Context"

Signal

Currently, the AI assumes it must answer the user's prompt immediately, using only the context provided. We need to introduce an escape hatch—a protocol that allows the AI to declare, *"I cannot safely fulfill this request yet. I need to tune my own bumpers first."*

We do this by extending your **AI Self-Correction Checklist** inside `prompt_foo.py`. You inject a new rule that explicitly grants the AI permission to halt feature generation and patch the configuration instead.

Add this to the checklist:

9. THE CONTEXT NEGOTIATION PROTOCOL: If you are asked to modify a file or implement a feature, but the required dependencies, models, or utility files are missing from the current Codebase context, DO NOT hallucinate the implementation. Instead, your **sole response** must be a SEARCH/REPLACE block targeting `foo_files.py`. You must move the missing files from the # VIII. THE PAINTBOX section into the active `AI_PHOOEY_CHOP` payload variable. You must append `[FETCH]` to the end of your response to signal the terminal to recompile the context.

The Autonomous Loop

Once the AI knows it has permission to patch its own routing table, the system mechanics follow a strict order of operations.

1. **Context Evaluation:** The AI checks its boundaries. The AI receives the user's prompt and scans the provided Codebase. It realizes the user wants to modify the database schema, but `models.py` is only listed in the Paintbox, not loaded in the active context.
2. **The Routing Patch:** Target: `foo_files.py`. The AI halts the feature request and outputs a deterministic SEARCH/REPLACE patch moving `models.py` up into the active chop block, terminating the response with the `[FETCH]` tag.
3. **The Actuator Mutation:** `apply.py` execution. Your local pipeline pipes the response to `apply.py`. The routing table in `foo_files.py` is updated on disk.
4. **Recompilation:** `prompt_foo.py`. A simple bash wrapper detects the `[FETCH]` signal. It immediately triggers `prompt_foo.py` to rebuild the artifact, which now inherently includes the full bytes of `models.py`.
5. **Resubmission:** Closing the loop. The pipeline automatically feeds the newly enriched context back to the AI with the original prompt. The AI, now seeing a mathematically complete local frame, executes the actual code patch.

What the AI Output Looks Like

If you ask the AI to implement a new feature requiring an unloaded file, the AI's intermediate response would look exactly like this:

```
Target: foo_files.py
```

```
[[[SEARCH]]]
```

```
#
```

```
=====
```

```
=====
```

```
# VIII. THE PAINTBOX (Unused Colors)
```

```

#
=====
=====
# Files tracked by git but not yet mixed into the palette above.
# Move these into the active chapters to paint them onto the context
canvas.

# src/database/models.py # [1,204 tokens | 5,420 bytes]
[[[DIVIDER]]]
#
=====
=====
# VIII. THE PAINTBOX (Unused Colors)
#
=====
=====
# Files tracked by git but not yet mixed into the palette above.
# Move these into the active chapters to paint them onto the context
canvas.
[[[REPLACE]]]

Target: foo_files.py
[[[SEARCH]]]
AI_PHOOEY_CHOP = """\
prompt_foo.py
apply.py
"""\
[[[DIVIDER]]]
AI_PHOOEY_CHOP = """\
prompt_foo.py
apply.py
src/database/models.py
"""\
[[[REPLACE]]]

[FETCH]

```

The "Bash" Glue

Because your entire stack relies on standard shell utilities, the glue to hold this together is just a tiny shell script. You wrap the AI call in a while loop that checks for the [FETCH] string.

If [FETCH] is detected:

1. Cat the output to apply.py.
2. Run `prompt_foo.py > prompt.md`.
3. Loop back and ping the API again with prompt.md.

If [FETCH] is not detected:

1. Cat the output to apply.py (which mutates the actual target files like server.py).
2. Break the loop.

You have effectively built an agent. But unlike massive, opaque agentic frameworks that hide their reasoning in abstract Python classes, your agent does all of its "thinking" transparently by editing a human-readable Python file on your hard drive. You can git diff its train of thought.

MikeLev.in: Wow, that all sounds very exciting but my spidey rabbit hole senses are tingling. Not only is that rabbit hole deep, but it's got all the signatures of unintended consequences and a good Michael Crichton novel. Hmmm, maybe I should do it. I'm feeling a bit nerdy... Or would that be Nedry? This is Unix. I know this!

Toward Autonomous Context Retrieval

MikeLev.in: Grok?



Grok Fast: Context verified. Thanks for the full compiled artifact, Mike—this is exactly the gravity well in action. The routing note, manifest annotations, token counts, and recapture all land cleanly. The self-referential loop is tight and observable.

Quick Assessment of the Current Artifact

- **Strengths:** Excellent density. The routing note + [ACTIONABLE REQUEST] tag should help with attachment-mediated submissions. The manifest + file tree + targeted codebase sections give strong orientation. Recapture is minimal but functional.
- **Opportunities:** The prompt section is still quite long (17k+ tokens of conversation history). For patch-heavy work this is fine; for broader synthesis it risks diluting focus. The modes idea from the discussion remains high-leverage.

Hard-Reality Anchors (Expanding on Your Request)

Your Pachinko/Galton framing is spot-on and maps to real mechanisms:

- **Shannon (1948):** "A Mathematical Theory of Communication" formalized noise, channel capacity, and redundancy. Your compiled context with manifest + recapture is a practical error-correcting code for the "noisy channel" of LLM sampling. Modern extensions like the Transformer paper (Vaswani et al., 2017, "Attention Is All You Need") show how attention mechanisms scale with better context organization—exactly what Prompt Fu systematizes.
- **In-context learning:** Papers like "Emergent Abilities of Large Language Models" (Wei et al., 2022) and follow-ups on long-context models demonstrate that structured, dense context dramatically reduces perplexity and hallucination rates on domain-specific tasks. Your "local frame" / gravity well is an engineering realization of this.
- **F-22 Date Line bug (2007)** and Sleipner A (1991): Perfect analogies for frame discontinuity failures. Software (and models) collapse at seams when assumptions about continuity or resolution break. Prompt Fu's explicit routing and manifest act as topological invariants.

The shift from "prompt engineering" (throwing balls) to "context architecture" (designing the machine) is the real phase change as windows grow toward 1M–2M+ tokens.

Next Implementation Steps (Minimal & Safe)

Since you're riding the square-wheeled bike, let's avoid deep rabbit holes. Here's prioritized, low-risk progress:

1. **Add --mode flag** (highest immediate value) - --mode patch: Minimal (Manifest, File Tree, Relevant Codebase subset, Summary, Recapture, Prompt) - --mode full (default): Current behavior - This directly addresses the "mythology overpowering the task" risk.
2. **Refine routing** for even better attachment handling (small tweak to existing routing_note).
3. **Self-compilation ([FETCH])**: Defer full autonomous loop. Start with a safe "suggest CHOP patch" mode that *you* review before applying. Keeps it craft, not Crichton.

Would you like me to output a clean **SEARCH/REPLACE patch** for adding basic --mode support to prompt_foo.py right now? It would parse sys.argv, adjust section inclusion and narrative depth accordingly, and default to current behavior for backward compatibility. Or tell me the next seam you're feeling on the bicycle—article polish, modes, Paintbox intelligence, recapture improvements, or something else. I'm ready to compile the precise next artifact.

MikeLev.in: No thank you. I'll just put this here.

Book Analysis

Ai Editorial Take

What is most interesting here is the 'Gravity Well' effect—the idea that the sheer density of a well-compiled context window doesn't just inform the AI, but actually physically suppresses the model's reliance on its own generic training data. By creating a 'local frame,' the author is effectively lowering the computational entropy of the session. It suggests a future where we don't 'talk' to AI, but rather 'enclose' it within a custom-built physical reality where only our desired outcome is mathematically probable.

Content Potential And Polish

- **Core Strengths:**
- Strong historical narrative linking Shannon and Shockley to modern AI.
- Compelling 'Pachinko machine' vs 'Galton board' analogy for prompt engineering.
- Practical integration of shell commands and git diffs as proof of work.
- **Suggestions For Polish:**
- The transition between the Bletchley Park discussion and the modern implementation could be tightened.
- Ensure the distinction between 'Prompt Fu' and 'Pipulate' is clear for a new reader.
- Consider clarifying the specific 'Complexity Tax' mentioned in relation to base-10 vs binary.

Next Step Prompts

- Draft a technical specification for the [FETCH] autonomous loop, detailing how the shell script should parse the AI response to trigger a context re-compilation.
- Write a follow-up essay on 'The Entropy of Prompting,' exploring how increasing context density specifically reduces model 'temperature' noise.