The “new thing” we have been doing for 50 years: What comes after code generation?

20 October 2026 · English translation, 2 November 2026

How I let an AI agent rewrite itself, why code generation is nothing new, and what programmers will do in the future.

I let an AI agent develop itself. Then everything got tangled up.

We are developing Promptbook (github.com/webgptorg/promptbook). Among other things, it is an AI coding agent. And because we are building a coding agent, a natural idea occurred to us: what if Promptbook developed itself?

It started innocently. Add a CLI flag. Fix a bug. Add support for another model. Gradually, we entrusted it with entire features and modules. Then we took another step: we stopped giving it tasks.

Promptbook could already identify work, create tasks, carry them out, and keep going. Not for an hour. For days or weeks. Almost continuously. And it worked. But while a person creates technical debt eight hours a day, an autonomous agent can work on it twenty-four hours a day.

New features. New abstractions. More dependencies. More tests. Fixing tests. Fixing the fixes to the tests. A slower build. A more complicated project. A merge conflict. Another conflict. Until, for the first time in my life, I was resolving a merge of three different commits at once.

Three branches of changes. One increasingly tangled project. … ENOUGH.

I did not want to spend another week untangling technical debt. I wanted to throw away the implementation and start again.

The old Promptbook created the new Promptbook

I had a specification created from the existing application. Not a list of classes or a copy of the architecture, but a description people could understand of what the software should do, and what it no longer needed to do. The result was a relatively short Markdown document.

Promptbook also has Book, our own language for defining agents, their goals, rules, and specifications: one attempt at programming at a higher level of abstraction. But the particular syntax does not matter for this experiment.

I ran the old, spaghetti-code Promptbook on a new project. It had no access to its original source code. It received the specification and an instruction: build an application from this. And it did. Without knowing it was writing its own successor.

In my experience, the resulting software had practically the same required functionality, while being roughly 20× faster and 100× smaller. This was not an independent benchmark or proof of full equivalence, but as an experiment it was convincing.

At first I thought: this is a completely new way to develop software. Then I realised we have known the principle for decades.

I invented something new. Or did I?

A programmer writes C. A compiler creates a lower-level representation and eventually machine instructions. The programmer does not usually deal with individual registers or instructions. They change the C, run the compiler, and can happily throw away the entire output. They can use a new version of GCC and get different machine code. Nobody finds this revolutionary. The source of truth is one level higher.

C → assembly → machine code. And in my experiment: specification → TypeScript → JavaScript → machine code.

For the pedants: yes, translating TypeScript into JavaScript is usually called transpilation. The actual chain also contains further intermediate steps. The point is the shift in the boundary between a higher-level representation and an artifact.

The old Promptbook creating its new version resembles compiler bootstrapping. GCC can compile a new GCC without needing to know it is compiling itself. This is not a technical equivalence: AI interprets much looser instructions. But it is a strong analogy.

Before source code: people and physical machines

When we say programming today, we picture text in an editor. But that picture is not inevitable. The word computer once described a person who performed calculations. An algorithm could be written on paper while a person executed it. Even before symbolic languages, there was a difference between describing a procedure and carrying it out; it just did not come wrapped in today's software infrastructure.

On some early electronic machines, a computational task was configured by physically connecting cables and setting switches. The program was largely the configuration of the machine. It is not accurate to say that people always “rewired the vacuum tubes”: the specific technologies varied. What matters is that a program was not yet routinely a portable text artifact, independent of a particular machine.

Then came machine code and symbolic instruction notation. Assembly gave people instruction names and symbolic addresses; an assembler converted them into a representation the processor understood.

Suddenly, there was a text-based program and a mechanism for converting it into executable instructions. The first major separation: a program was no longer just a machine's physical configuration.

Higher-level languages: when we stop describing instructions

Fortran, COBOL, C, and other languages moved the boundary higher still. Instead of deciding on every register, people began writing expressions, functions, data structures, and algorithms. The compiler's job was to decide how to realise that description on a particular processor.

Imagine adding two numbers in C. The expression `a + b` is not a direct command to use a particular register and a specific machine instruction. It describes an operation in a language with defined semantics. The compiler can choose different implementations depending on the architecture, optimisations, and surrounding program. Yet the programmer still works with a single source expression.

A fundamental exchange took place here: we willingly gave up some low-level control so we could think in larger units. In everyday programming, we do not consider that loss of control a failure. We consider it an advantage of abstraction. Nobody wants to manually write hundreds of assembly instructions into every C function just to keep “everything under control”.

We find the same shift outside programming languages. SQL expresses which data we want, and the database optimiser chooses a query plan. CSS expresses rules for appearance, rather than a sequence of pixel operations on a graphics card. A higher-level representation gives the lower layer room to implement it. Abstraction is therefore more than convenient syntax; it is the delegation of decisions.

A second strand of history: how layers communicate

Alongside the history of languages runs another, equally interesting history: the relationship between higher and lower layers. The simplest model is compilation. I write source, compile it, and execute the result. But interpreters have long existed too, executing a higher-level representation at runtime, as have virtual machines that combine different intermediate stages.

Interpretation did not begin with Python, and JIT did not begin with Java. These techniques have long, overlapping histories. A modern JavaScript engine may first produce bytecode, interpret some parts, and later optimise frequently used paths into machine code. Likewise, different implementations of Java, C#, or Python choose different combinations of compilation and execution.

What matters is not assigning one label to every language. What matters is understanding that higher abstraction gave us more choices about when and how to turn it into something lower-level. Ahead-of-time compilation, runtime interpretation, just-in-time materialisation as needed: all these relationships can coexist.

A new layer: human intent as a program

Today, we may be adding another level. A person no longer has to start with TypeScript. They can start by describing the required behaviour, constraints, and goal. A model then creates a concrete implementation. Natural language, of course, does not have C's fixed formal semantics, and one specification may allow many correct implementations. That is why the result must be verified. But it can already work as a language in which a person programs at a higher level.

From this perspective, TypeScript may be to the author of a specification what assembly is to the author of a C program. Sometimes they read, debug, and fix it, but their main expression of intent is the higher layer. When a C programmer compiles a project, they do not usually say “I generated machine code”. They say “I wrote a program in C”. Perhaps one day we will not say “AI generated an application for me”, but “I programmed in a specification”.

This also changes what we expect from a specification. If we fill it with internal implementation details, we are not using a higher abstraction. We have simply rewritten lower-level code in words. A specification should define the important product invariants, public contracts, and required outcomes. It should not replace every function and every internal endpoint.

3 reasons AI is not a conventional compiler. Or perhaps it is?

1. “A compiler is deterministic. AI rolls the dice.”

When I run GCC on the same source code, I expect the same result. When I give AI the same specification, it may write two different programs. At first glance, a fundamental difference. But GCC's output is not determined by the program text alone: it depends on the compiler version, optimisations, target architecture, and build environment. Compilers also contain bugs, and their behaviour changes over time. This does not erase the fundamental difference, of course: C has defined semantics, whereas natural language admits many interpretations.

But randomness in generative models is not some metaphysical property of AI either. Sampling is a selection mechanism; in some environments, we can constrain it, set a seed, and fix the model and runtime configuration. A seed alone does not guarantee identical results across hardware or services, and reproducible output does not mean correct output. Still, the difference is less black and white than a simple “deterministic versus random”. It is a difference in predictability, guarantees, and how much we must hold fixed to reproduce a result. And those boundaries can change.

2. “Compilation is cheap. AI generation is an entire development sprint.”

A compiler today will commonly compile a small program in an instant. An agent may work on a specification for hours, running tests, fixing bugs, and consuming significant resources. That is a real difference. But perhaps we overestimate it because we live in an age of rudimentary natural-language “compilers”.

Conventional compilers have gone through decades of optimisation, incremental builds, caching, and parallelisation. The first compilers also appeared back in the 1950s, when computing resources were incomparably scarcer. Even today, building a large project is not always cheap. We cannot infer from this that AI generation will ever reach the cost of compiling C: it solves a more open-ended problem and has to search, verify, and decide. But its current slowness does not prove that the processes are unrelated in principle. It may partly reflect the technology's present stage.

3. “Generated code is not edited. Yet with AI, we edit it all the time.”

With C, we typically return to the source file rather than the resulting assembly. With AI, we often open the generated TypeScript and continue directly in it. But this boundary is not absolute either. C has inline assembly, intrinsics, pragmas, and other ways to influence the lower-level representation from a higher layer. And conversely, with AI we can fix a public interface, reference implementation, or test while continuing to generate the other parts.

The difference, however, is not just whether we ever touch the lower layer. What matters is how much authority we give it. Must we describe everything in the top-level specification? Can we preserve something in the lower layer without elevating that entire layer into an untouchable source of truth? This question will lead us to a fourth relationship between abstractions.

A two-hundred-page specification can be a bad specification

I see a trend towards ever-longer specifications. Fifty pages. A hundred. Two hundred. Sometimes AI generates the whole thing, as if the number of pages proved its quality. In my view, it is often the other way round.

If a specification describes every internal function, endpoint, and implementation decision, perhaps we have not created a higher abstraction. We have just rewritten the source code in English.

Each level of abstraction should describe what belongs at that level. In C, I do not want to assign processor registers by hand. In TypeScript, I do not want to manage memory manually. In a product specification, I do not want to dictate internal endpoints unless they are themselves an important contract. One example of an interface can be useful, but systematically transferring the implementation into the specification defeats the point of abstraction.

Fear is often behind this: I want everything under control. Yet we created higher-level languages precisely so we would not have to control everything. A good specification defines the important things precisely and delegates the rest.

And what if something valuable already exists in the lower layer, something I do not want to lose?

3 familiar relationships between higher and lower abstractions

Two parallel strands in the history of programming interest me. The first is the growth of abstraction: from physically configuring a machine, through instructions and higher-level languages, to describing human intent. The second is the changing relationship between these layers. There is more than one way for a higher-level representation to become a lower-level one. And in the world of AI, the differences between these approaches are becoming surprisingly practical.

Before we add something new, it is worth walking through three familiar models. They are not exclusive categories of languages: a single runtime can use all three. For each, let us look at traditional programming first and its AI analogy second.

1. Compilation: translate a higher-level expression into a lower-level representation

The conventional world. I write C, and a compiler produces a machine program, perhaps via assembly. Or I write TypeScript, and a tool turns it into JavaScript (for the pedants: we usually call that transpilation). The author works with a higher-level representation, while the resulting program runs on a lower-level representation. I change the source and build a new result; I can throw away the old output.

The AI world. I write an application specification in natural language. An agent creates TypeScript, tests, and other files. After verification, I run the resulting application without involving the original generator. This is the clearest, most straightforward form of spec-driven development: the specification acts as the source, and the code as generated output. The difference from GCC is in the weaker semantic guarantees; that is why generated software must be tested and verified.

```
Specification → AI build → TypeScript → JavaScript → execution
```

2. Interpretation: execute the higher-level expression directly

The conventional world. An interpreter reads a program and carries out its operations at runtime. A simple BASIC interpreter or a shell evaluating commands one by one makes a useful teaching example. With modern Python, bytecode is commonly generated first and then executed by a virtual machine; even “interpreted language” does not mean there is no intermediate representation. It is a method of execution, not an immutable property of the syntax.

The AI world. I have an agent for customer emails and tell it: “Read new messages, answer routine questions, and hand complicated cases to a person.” No permanent email application necessarily emerges. The agent keeps evaluating the instruction against each incoming message, calls APIs and tools, and may briefly create a shell command or a little JavaScript that it discards afterwards. The user maintains the instruction, not a permanently generated codebase. In practical terms, that is the interpretation of a natural-language program.

```
Instruction + incoming email → agent → tool calls → response
```

3. Just-in-time: materialise the lower layer only when it is needed

The conventional world. Java is a familiar example, not the inventor of JIT. The Java Virtual Machine can begin by executing bytecode and compile a repeatedly used part of the program into optimised machine code. Profiling identifies a “hot” path, and the runtime invests in compilation because subsequent runs will be cheaper. The history of interpretation and JIT predates Java; the important thing is the economic principle: I prepare something more expensive only when it pays off.

The AI world. An agent processes data and notices that it keeps cleaning the same kind of CSV. Instead of reasoning about each row again and again, it creates a script and saves it for later runs. Or it creates a helper library or algorithm. A user asks to compare business scenarios, and the agent builds generative UI on demand: an interactive chart and controls for this particular task. These applications and utilities are not built in advance; they materialise according to the current need. They may live in a hidden runtime working directory without the user ever seeing their source code.

When an agent needs to sleep: JIT as an alarm clock

Long-running agents are an interesting example. Imagine an agent that should react to an incoming email or a change on a web page. The naive solution is to wake the entire language model every minute and have it check again whether anything has happened. It works, but consumes tokens and computing resources even when there is nothing to do.

The agent can instead generate a small low-level watcher: a cron job, webhook handler, mailbox-watching script, or heuristic for detecting a page change. It puts itself to sleep and lets the cheap script watch for a particular condition. When the event occurs, the script wakes the expensive agent. This resembles hardware interrupts and low-power mechanisms that wake the main processor only when a relevant event occurs.

In Promptbook, an agent's long-term goal is one example of such a higher-level instruction. Some wake-up conditions can be derived and materialised as a simple script; others require more complex assessment by the model itself. Deciding whether to turn something into conventional code is an optimisation decision. As with JIT in Java, we invest once in a lower-level representation to save costs on repeated execution. I do not consider the implementations identical: the JVM optimises an already defined program, while the agent may still be designing the watching algorithm.

```
Agent goal → watcher created once → cheap waiting
                          ↓ event
                       agent wakes
```

And this raises another question. What if the user wants to see that generated script, component, or interface, and wants to preserve some of its properties? That takes us beyond a hidden JIT artifact.

But wait. Does AI have a 4th relationship too?

Anchoring: the lower layer gains limited authority

Now for what I think complements those three models. An agent has generated a UI. It is beautiful and polished. Or it has created an API that customers already depend on. Or an algorithm that finally handles a difficult edge case correctly. At the next generation, I do not want to throw absolutely everything away, but I do not want to declare the entire existing codebase untouchable either.

So I choose particular invariants and anchor them. Sometimes the invariant is an exact file. At other times, it is a public interface, visual identity, behaviour, or the result of a regression test. The generated artifact does not become the highest source of truth, but it gains limited authority over subsequent transformations. That is anchoring: selectively preserving value in the lower layer.

Compilation asks: “What should we create from the higher-level instruction?” Interpretation: “How should we execute the instruction now?” JIT: “What is worth creating only when it is needed?” Anchoring adds: “What that already exists must survive the next change?” This is why I see it as more than another generation method: it is a fourth relationship between layers of abstraction. I am

not claiming that invariants, reference implementations, or partial regeneration did not exist before AI. What is new is mainly how central these boundaries may become in generative development.

```
Specification + selected anchors + new task
                    ↓
             AI transformation
                    ↓
       New code, preserved invariants
```

All four relationships can coexist in one project. An agent can interpret a long-term goal, create a utility just in time, compile a new application from a specification, and respect an anchored API throughout. These are not four competing types of AI, but four ways in which higher and lower representations work together.

Anchoring: when a generated artifact is more than waste

Imagine AI creates a web application. It has a beautiful component: a precise design, animations, and polished behaviour. You want to add another feature. You can regenerate everything and hope the component comes out the same. Or keep the entire project and simply add code. The first path loses details. The second preserves technical debt.

We need a third option: anchor only what matters.

Anchoring is the principle of recognising value in particular parts of a generated or previously existing artifact that we want to preserve through later transformations. They are not the primary source of truth. But they are not entirely disposable either.

We can talk about code anchoring, UI anchoring, design anchoring, behaviour anchoring, or feature anchoring. An anchor can be a component, interaction, visual detail, API contract, or distinctive behaviour.

Anchoring does not necessarily mean preserving identical bytes. A design can survive a new implementation. Behaviour can be anchored by a test. Code by a particular module. It is more than a cache, a Git commit, or an ordinary build artifact: it is a decision about how much authority part of the result should have over a future change.

The specification says what should be created. Anchors say which parts of what already exists are worth preserving.

Neither source of truth nor disposable artifact

In a traditional build, the boundary is usually fairly clear: we keep the source code, while the build output can be recreated. AI-native development introduces a spectrum. Some parts are the highest-level specification. Others can be regenerated. And some are anchored: they have less authority than the specification but still contain value.

Anchoring is therefore more than another name for iterative generation. We can iterate on entirely disposable output too. Anchoring describes selective continuity. And it applies to older projects that never had an original specification as well.

What exactly can be an anchor?

Let us start with a visual design. An agent creates a dashboard. The specification says it should be clear, but the particular typography, proportions, and animations only work well on the third attempt. I can anchor the whole design, a particular component, or just selected visual invariants. I am not saying every CSS line is untouchable. I am saying I do not want to lose a particular result at the next generation.

The second example is a public API. A product specification might say “customers can retrieve invoices through an API”. But the real system already uses particular endpoints, field names, and error behaviour. If external clients depend on them, they have value even if a new implementation could be more elegant. The anchor is not necessarily the old controller; it is the compatible contract.

The third example is an algorithm. A model generates an optimisation that, after extensive tuning, handles a particular edge case. If I described it in the higher-level specification down to every implementation detail, I would defeat the purpose of abstraction. But I can preserve a reference implementation, a set of regression tests, or a property the new implementation must demonstrably satisfy.

The fourth example is content. An agent prepares a brand's tone of voice, an illustration, a video, or a particular onboarding sequence. The result is partly generated, but becomes valuable once approved. I do not want to rewrite it on every new run just because it can be recreated. Here we are talking about design anchoring, content anchoring, or behaviour anchoring, not only code anchoring.

The fifth example is an older project that never had a higher-level specification. Its source code may be the only surviving record of certain decisions. Only gradually do I extract the higher-level intent from it and distinguish which existing properties are binding and which have merely survived by accident. Anchoring does not therefore have to begin by generating an artifact. It can also be a way of selectively carrying value over from an existing system.

The key word is invariant. Sometimes it will be the exact contents of a file. At other times, an interface, behaviour, or result that can survive a complete implementation rewrite. An anchor should therefore express its scope, reason, verification method, and, where relevant, rules for releasing it. Otherwise, “anchoring” is just a nicer name for being afraid to change anything.

And what does Git have to say?

Today's Git versions files very well, but typically does not record what they mean for regeneration. In practice, we often distinguish committed source from ignored build output, yet that pair is incomplete: even a generated lockfile, approved design, or reference component may have a reason to remain in the repository. Nor does `.gitignore` alone decide what is the source of truth.

Perhaps we will need a third semantic category: anchor. A file or property that can technically be regenerated, but whose selected invariants should survive the next transformation. This need not be a new feature in Git itself; metadata, agent rules, and validation tests could do the job. What

matters is that version control can eventually capture not only what we keep, but also why we do not want to lose it.

Is the age of frameworks ending?

Libraries and frameworks are special cases of abstraction. They are often written in the same language as the application, but offer higher-level concepts and ready-made decisions: instead of manually handling HTTP requests, I use a web framework; instead of manipulating the DOM directly, I use a component library. So this is more than translation between two syntaxes. Abstraction can exist within one language as a set of conventions, contracts, and reusable mechanisms.

My hypothesis is that, with natural-language programming, the boundaries between a language, a framework, and a particular application will become less distinct. An agent may adopt some rules as a "framework", interpret some directly, materialise some as code, and anchor others. But this is not a proven direction of history or a claim that frameworks will disappear. It is a working prediction based on how instructions, tools, and generated artifacts overlap in practice.

What actually happened to Promptbook?

An autonomous agent kept adding features over a long period. Every task started from the existing project. Because we preserved practically everything, we were unknowingly anchoring almost the entire codebase. Not because every component was valuable. We simply left it in place.

The generated result had become a bundle of historical decisions. When I had a specification created and Promptbook rewritten, I did two things: I brought the important intent up into a higher abstraction and removed most of the unwanted anchors.

I did not change the principle of autonomous development. I did not abandon the Agent–Project–Task Framework. I simply stopped pretending every line of the existing implementation was worth preserving.

Agent, Project, Task: where these layers live

In Promptbook, we use the Agent–Project–Task Framework (APT), described in more detail in the whitepaper at https://www.ptbk.io/cs/whitepaper.

An Agent carries a role, goal, rules, knowledge, and tools. In our analogy, it partly resembles a compiler configuration, but it also makes its own decisions about work. A Project is a container for context: it holds specifications, files, history, and any anchored artifacts. A Task is a specific step of change that a person can request, or that can arise from an agent's goal or an external event.

These three parts form a living context. APT was the same before and after Promptbook's rewrite. The project contents and degree of anchoring changed, not the principle of the framework.

The dark factory and the limits of autonomy

Our long-term vision is something like a dark factory: a system in which agents develop and operate the work entrusted to them without constant human instructions. A project has a goal, and agents identify work, create tasks, verify results, and continue.

At the same time, I think absolute autonomy is an unattainable ideal in practice. Not because models will never be smart enough, but because value is more than a technical property of a result. Someone has to decide what to preserve, which compromise to accept, which property nobody needs anymore, and when it is better to discard the implementation.

AI can help with those decisions. But responsibility for a project's intent and values does not magically disappear.

The programmer of the future will do more than write more lines

I think many people whose work today mainly consists of producing more features will have to change how they work. But that does not mean programming is ending. To me, programming in specifications is still programming. Just at a different level of abstraction.

The key skill will be more than knowing how to create another feature. It will be recognising what has real value, what should be the source of truth, what to anchor, and what to discard without regret. Which goal an agent should pursue. Which boundaries it should respect. When further generation helps, and when it merely piles up clutter.

Perhaps we will keep calling these people programmers. Perhaps someone will find another name for them. To me, it still looks like programming: we have simply moved up another level of abstraction.

And that is why I am not afraid that human work will simply evaporate. Some current roles will undoubtedly disappear or change. But the ability to decide about value, priorities, and the right boundaries of abstraction will become even more important.

When I rewrote Promptbook, the biggest achievement was not that AI could write new code. It had been able to do that for a long time.

What mattered most was that I could finally say which old code was no longer worth preserving.

Perhaps the future of programming will not be about how much code we can create. It will be about how much we can let go of with a clear conscience.

About the author.

Pavol Hejný is a software developer and founder of projects in AI. He has been developing software for roughly 15 years and has spent the last four years working intensively with AI in programming. He works on Promptbook and other projects at the intersection of web technologies, agents, and automation. This article is a personal technological reflection: the historical analogies are explanatory, while anchoring and the outlook for the future are hypotheses open to discussion.