NEWS · JULY 14, 2026

TypeScript 7 Is Here: What's 10x Faster Today, and What Still Needs Manual Wiring

By Dr. Holger Flick



A few weeks ago I wrote about Anders Hejlsberg — the man who gave us Turbo Pascal, Delphi, C#, and then bolted a real type system onto JavaScript with TypeScript — and I noted, almost in passing, that his team had "been rewriting their compiler for raw speed." Well, that rewrite just shipped. TypeScript 7.0 **— the compiler reimplemented from scratch in Go — reached general availability on July 8, 2026**, and today Visual Studio Code announced it ships integrated.

I'll be honest about my bias up front: I've been waiting for this one. The headline number is roughly a **10x faster type-check**, and for anyone who has watched a large project's `tsc` crawl, that's not a rounding error — it's the difference between a build you tolerate and a build you don't think about.

But "GA" and "VS Code ships it" do not automatically mean *your* pipeline got 10x faster this morning. There's a gap between the compiler being ready and your framework — Next.js included — actually routing its work through it. This post is the honest state of play: what's genuinely faster today, and what still needs a little manual wiring for a few more days or weeks.

First, the name: it's TypeScript 7, not "the tsgo thing"

Let me clear up the versioning, because it tripped me up too. The native compiler had a working name — `tsgo` — during its 2025 preview, distributed as the `@typescript/native-preview` npm package. That name is going away. Internally the effort was called Project Corsa, replacing the old JavaScript-based codebase nicknamed *Strada*.

The version numbering matters because it tells you the migration story at a glance.



The JavaScript line closes out at 6.x; 7.0 is the native Go compiler

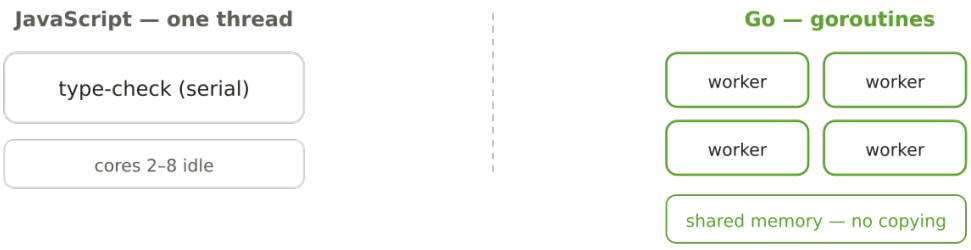
The takeaway from that line: **6.x is the last JavaScript-based TypeScript**, kept around to smooth over deprecations, and **7.0 is the clean break to native code**. If you've been typing `tsgo` in preview builds, you can forget it — in the release, the native compiler simply *is* `tsc`.

Why the old compiler was slow — and why Go, not JavaScript

Here's the part that fascinated me, and it's worth getting right because it's easy to tell this story wrong. The old `tsc` wasn't slow because the team wrote bad code. It was slow because of a **fundamental mismatch between the workload and the runtime**.

JavaScript's execution model is a single-threaded event loop. That design is *brilliant* for I/O-bound work — web servers juggling thousands of connections, waiting on the network — which is exactly what Node.js was built for. But type-checking a large codebase is the opposite kind of work: it's **CPU-bound**, a long, dense computation over deeply interconnected data structures. On a single-threaded event loop, that computation monopolizes the one thread it has, and there's no clean way to spread it across the eight or sixteen cores sitting idle in your machine.

Go is a much better fit for *this specific job*, for reasons that are concrete, not hand-wavy.



Single event loop vs. goroutines spreading type-checking across cores

Reading that diagram left to right: the same type-checking work that JavaScript had to run serially, Go can fan out across cores. Three properties of Go make that possible:

- **Native compilation.** Go compiles straight to machine code. There's no JavaScript engine warming up a JIT, no Node.js runtime layer to boot — the compiler *is* a native binary. That alone removes a startup and interpretation tax on every run.
- **Goroutines — cheap, real concurrency.** Goroutines are lightweight (a couple of kilobytes each) and the Go runtime schedules them across all your CPU cores. Parsing, binding, and type-checking can genuinely run in parallel instead of taking turns.
- **Shared memory.** A compiler is a graph of symbols, types, and nodes all pointing at each other. Go's goroutines share the same memory directly, so those workers read the same structures without serializing and copying data between threads — the exact overhead that made parallelizing the JS version impractical.

And there's a pragmatic reason Go won over the alternatives, including Rust: Anders has said plainly that **Go was the easiest language to port the existing codebase into**. Because Go's structure lined up well with the old code, the team could do a near *line-by-line* port — same algorithms, same data structures, and therefore the **same type-checking behavior**. That last point is the quiet hero of this whole project: your types don't start behaving differently just because the compiler underneath got rewritten.

> **Say it precisely: the compiler got faster, not the language**
> It's worth being exact, because the marketing shorthand blurs it. TypeScript-the-language didn't change — the same `tsc` you know, checking the same types, just runs on a native, parallel engine now. As one analyst put it, whenever you see a "10x faster" headline you should read it with healthy skepticism and ask *what*, precisely, got faster. Here the honest answer is: type-checking and editor responsiveness. Your app's *runtime* speed is unaffected — TypeScript still compiles down to the same JavaScript.

The numbers, and where they came from

I don't want to throw "10x" around without showing the receipts, so here are the figures Microsoft published, measured on real codebases rather than a synthetic benchmark.

Codebase	Old `tsc`	Native (Go)	Speedup
VS Code (~1.5M LOC)	~77.8s	~7.5s	**10.4x**
TypeORM (~270K LOC)	~17.5s	~1.3s	**13.5x**
Editor project-load	~9.6s	~1.2s	**~8x**

On top of the wall-clock wins, memory usage drops to **roughly half** of the previous implementation — which matters as much as raw speed on a laptop or in a memory-capped CI runner. The editor numbers are the ones I care about day to day: project load and "go to definition" and rename feeling instant is a different *quality* of working, not just a faster one.

> **Your mileage will vary — and that's expected**

> These are Microsoft's numbers on their chosen projects. A ~10x speedup on a 1.5M-line codebase says little about a 5,000-line one, where `tsc` was already fast enough that you'd never notice. The gains are most dramatic exactly where the pain was worst: large projects, cold builds, and busy language-service operations. Measure your own before and after — don't take my excitement (or Microsoft's table) as a promise about your repo.

What's actually faster *today*

So GA landed on July 8, and today VS Code announced it ships the native TypeScript integrated. Concretely, what changed for you right now:

1. The editor experience, once you're on the native language service

VS Code's TypeScript support — completions, hovers, rename, go-to-definition — runs through the language service. With the native engine driving it, those operations get noticeably snappier, especially on big projects. Visual Studio 2026 Insiders builds have been previewing the same native language service as well.

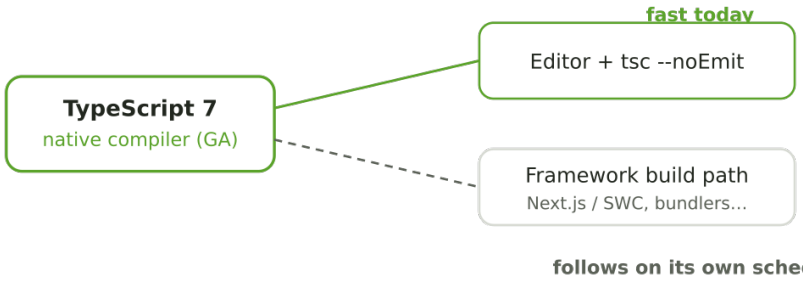
2. Command-line type-checking and builds

Running `tsc --noEmit` in CI, or a plain build, goes through the native compiler now. This is where the headline speedups live, and it needs no framework cooperation — it's just the compiler doing its job faster.

What still needs manual wiring (for now)

Here's the honest caveat, and the reason I titled this post the way I did. **Frameworks that run their own TypeScript pipeline don't automatically inherit the native compiler the moment it's GA.** Bundlers, dev servers, and build tools integrate `typescript` as a library, drive their own transforms, and sometimes strip types with a *separate* tool entirely — so "TS7 is out" doesn't flip a switch inside them.

Next.js is the case I hit first. A production Next.js build doesn't lean on `tsc` to emit your JavaScript at all — it type-checks with TypeScript but transpiles with SWC, its Rust-based compiler. So the native TypeScript engine helps your *editor* and your `tsc` *type-check step* immediately, but wiring it into the framework's own build path is a separate piece of integration work that lands on the framework's schedule, not the compiler's.



GA flips the editor and tsc today; framework build paths follow on their own schedule

The diagram says it cleanly: one arrow is solid and live today, the other is dashed and coming. My read is that it'll take days to a few weeks for the framework side to catch up — but I want to be clear that **this is my expectation, not a shipped date on a roadmap I can point you to.** Treat it as informed impatience, not a promise.

> **Try it without betting your build on it**
> If you want to feel the difference today, run the native type-check alongside your normal build rather than ripping anything out: keep your framework's build as-is and add a native `tsc --noEmit` as a separate CI job or a local script. You get the fast feedback loop immediately, and you're not blocked waiting on framework integration. When your framework wires the native compiler into its own pipeline, you remove the extra step.

A fair word on the alternatives

Because this blog tries to show you the whole landscape: the JavaScript ecosystem already has *other* very fast type-tooling, and TS7 doesn't erase the reasons people reach for it. Tools like esbuild and SWC transpile TypeScript at breathtaking speed by **stripping types without checking them** — a different job than what `tsc` does. Node.js itself can now run `.ts` files directly by erasing types on the fly. Those remain excellent when you want *speed of transpilation* and are happy to let your editor or a separate CI step handle *correctness*.

What TS7 changes is the thing only the real compiler can do — **full, semantic type-checking** — going from slow to fast. So it's not "native TypeScript wins"; it's "the one part of the toolchain that was still slow just caught up with the parts that were already quick." Pick by need: transpile-and-go tools for raw build throughput, the native `tsc` when you want the type system's full judgment at speed.

Takeaways

TypeScript 7 is the payoff of a genuinely ambitious rewrite, and the honest version of the story is more interesting than the headline.

- **It's version 7, and it's native.** The Go-based compiler (Project Corsa) is GA as of July 8, 2026; 6.x was the last JavaScript line. `tsgo` was just the preview name.
- **Go fits the workload.** Native compilation, goroutines, and shared memory let type-checking run in parallel — something JavaScript's single-threaded event loop couldn't do. A near line-by-line port kept type-checking behavior identical.
- **~10x, on large codebases, for the compiler — not the language.** Your runtime is unchanged; your builds and editor get faster, most dramatically where they hurt most. Measure your own repo.
- **Editor + `tsc` are fast today; frameworks follow.** VS Code ships it now, but Next.js and friends need to route their own build path through it — expect that over the coming days and weeks, and add a native `tsc --noEmit` step in the meantime.

> The language didn't change. The one slow part of the toolchain finally caught up with the fast parts — and that's worth being excited about.

I've been waiting for this since I wrote about Anders and the conviction that runs through everything he builds: fast tools, honest types, the compiler and the editor treated as one product. TypeScript 7 is that conviction, shipped. I can hardly wait to watch the rest of the ecosystem light up behind it.

INDEPENDENT · AD-FREE · READER-SUPPORTED

Free to read. Not free to make.

Every tutorial, article, and line of example code on Holger's Code is published without a paywall and without ads. This page is about what keeps it that way.

Professional work, published free

The content here is held to the same standard as professional client work: every tutorial is built as a real project, tested against current versions, documented step by step, and revisited when the frameworks move on. A single in-depth tutorial routinely takes days — and that's before the bills behind it: servers, storage, build infrastructure, licenses, and the hardware everything is verified on.

If an article or tutorial here has saved you an afternoon of debugging, taught you a technique, or helped you ship, it created real value for you. Supporting it isn't charity — it's paying for professional work you found useful, at whatever price you choose. And if you can't, keep reading anyway. That's what the free part means.

Ways to support

KO-FI

Buy me a coffee

A one-time tip — no account, no subscription, any amount.

BOOKS & COURSES

Buy the paid work

The paid tier of everything published free here — and you get something lasting in return.

SPREAD THE WORD

Share an article

Send a tutorial to a colleague, cite an article, subscribe to the RSS feed.

Nothing here goes behind a paywall, whether you chip in or not. But if something on this site earned its keep today, this is where to say so.

Support on Ko-fi —