NEWS · JULY 14, 2026

TypeScript 7 ist da: Was heute schon 10× schneller ist – und was noch Handarbeit braucht

By Dr. Holger Flick



Vor ein paar Wochen habe ich über Anders Hejlsberg geschrieben – den Mann, der uns Turbo Pascal, Delphi, C# schenkte und dann JavaScript mit TypeScript ein echtes Typsystem verpasste – und ich erwähnte fast beiläufig, dass sein Team „den Compiler auf reine Geschwindigkeit umschreibt". Nun, dieses Umschreiben ist gerade fertig geworden. TypeScript 7.0 **– der von Grund auf in Go neu implementierte Compiler – hat am 8. Juli 2026 die allgemeine Verfügbarkeit erreicht**, und heute hat Visual Studio Code angekündigt, dass es ihn integriert ausliefert.

Ich bin ehrlich, was meine Voreingenommenheit angeht: Auf diese Neuerung habe ich gewartet. Die Schlagzeile lautet etwa **10× schnellere Typprüfung**, und für jeden, der zugesehen hat, wie sich das `tsc` eines großen Projekts dahinschleppt, ist das kein Rundungsfehler – es ist der Unterschied zwischen einem Build, den man erträgt, und einem, an den man keinen Gedanken mehr verschwendet.

Aber „GA" und „VS Code liefert es aus" bedeuten nicht automatisch, dass *deine* Pipeline heute Morgen 10× schneller geworden ist. Zwischen „der Compiler ist fertig" und „dein Framework – Next.js inbegriffen – leitet seine Arbeit tatsächlich durch ihn" klafft eine Lücke. Dieser Beitrag ist die ehrliche Bestandsaufnahme: Was ist heute wirklich schneller, und was braucht noch für ein paar Tage oder Wochen etwas Handarbeit.

Zuerst der Name: es ist TypeScript 7, nicht „das tsgo-Ding"

Räumen wir die Versionsnummern auf, denn sie haben auch mich verwirrt. Der native Compiler trug während seiner Vorschauphase 2025 einen Arbeitsnamen – `tsgo` –, ausgeliefert als npm-Paket `@typescript/native-preview`. Dieser Name verschwindet. Intern hieß das Vorhaben Project Corsa und ersetzte die alte, JavaScript-basierte Codebasis mit dem Spitznamen *Strada*.

Die Versionsnummerierung ist wichtig, weil sie die Migrationsgeschichte auf einen Blick erzählt.



Die JavaScript-Linie endet mit 6.x; 7.0 ist der native Go-Compiler

Die Kernaussage dieser Linie: **6.x ist das letzte JavaScript-basierte TypeScript**, aufbewahrt, um Deprecations zu glätten, und **7.0 ist der saubere Schnitt zu nativem Code**. Wenn du in Vorschau-Builds `tsgo` getippt hast, kannst du es vergessen – in der finalen Fassung *ist* der native Compiler schlicht `tsc`.

Warum der alte Compiler langsam war – und warum Go, nicht JavaScript

Dieser Teil hat mich fasziniert, und es lohnt sich, ihn richtig zu erzählen, denn man erzählt ihn leicht falsch. Das alte `tsc` war nicht langsam, weil das Team schlechten Code geschrieben hätte. Es war langsam wegen einer **grundlegenden Nichtübereinstimmung zwischen Arbeitslast und Laufzeitumgebung**.

JavaScripts Ausführungsmodell ist eine einsträngige Event-Loop. Dieses Design ist *brillant* für I/O-lastige Arbeit – Webserver, die Tausende Verbindungen jonglieren und aufs Netzwerk warten – genau wofür Node.js gebaut wurde. Aber die Typprüfung einer großen Codebasis ist die entgegengesetzte Art von Arbeit: Sie ist **CPU-lastig**, eine lange, dichte Berechnung über tief vernetzte Datenstrukturen. Auf einer einsträngigen Event-Loop belegt diese Berechnung den einen Thread, den sie hat, vollständig, und es gibt keinen sauberen Weg, sie über die acht oder sechzehn Kerne zu verteilen, die untätig in deinem Rechner sitzen.

Go passt für *genau diese Aufgabe* deutlich besser, und zwar aus konkreten, nicht schwammigen Gründen.



Eine Event-Loop vs. Goroutinen, die die Typprüfung über Kerne verteilen

Liest man das Diagramm von links nach rechts: Dieselbe Typprüfung, die JavaScript seriell abarbeiten musste, kann Go über die Kerne auffächern. Drei Eigenschaften von Go machen das möglich:

- **Native Kompilierung.** Go kompiliert direkt zu Maschinencode. Keine JavaScript-Engine, die einen JIT aufwärmt, keine Node.js-Laufzeitschicht, die booten muss – der Compiler *ist* eine native Binärdatei. Schon das nimmt jedem Lauf einen Start- und Interpretationsaufschlag.
- **Goroutinen – günstige, echte Nebenläufigkeit.** Goroutinen sind leichtgewichtig (je ein paar Kilobyte), und die Go-Laufzeit verteilt sie über alle CPU-Kerne. Parsen, Binden und Typprüfen können echt parallel laufen, statt sich abzuwechseln.
- **Gemeinsamer Speicher.** Ein Compiler ist ein Graph aus Symbolen, Typen und Knoten, die alle aufeinander zeigen. Gos Goroutinen teilen sich denselben Speicher direkt, sodass diese Worker dieselben Strukturen lesen, ohne Daten zwischen Threads zu serialisieren und zu kopieren – genau der Aufwand, der die Parallelisierung der JS-Version unpraktikabel machte.

Und es gibt einen pragmatischen Grund, warum Go sich gegen die Alternativen durchsetzte, Rust eingeschlossen: Anders hat klar gesagt, dass **Go die Sprache war, in die sich die bestehende Codebasis am leichtesten portieren ließ**. Weil Gos Struktur gut zum alten Code passte, konnte das Team eine nahezu *zeilenweise* Portierung vornehmen – dieselben Algorithmen, dieselben Datenstrukturen und damit dasselbe **Typprüfungsverhalten**. Dieser letzte Punkt ist der stille Held des ganzen Projekts: Deine Typen fangen nicht plötzlich an, sich anders zu verhalten, nur weil der Compiler darunter neu geschrieben wurde.

> **Präzise gesagt: der Compiler wurde schneller, nicht die Sprache**
> Es lohnt sich, hier genau zu sein, denn die Marketing-Kurzform verwischt es. TypeScript-die-Sprache hat sich nicht geändert – dasselbe `tsc`, das du kennst, das dieselben Typen prüft, läuft jetzt nur auf einer nativen, parallelen Engine. Wie ein Analyst es formulierte: Wann immer du eine „10× schneller"-Schlagzeile siehst, solltest du sie mit gesunder Skepsis lesen und fragen, *was* genau schneller wurde. Hier lautet die ehrliche Antwort: Typprüfung und Editor-Reaktionsfähigkeit. Die *Laufzeit* deiner App ist unberührt – TypeScript kompiliert weiterhin zu demselben JavaScript.

Die Zahlen, und woher sie stammen

Ich möchte „10×" nicht in den Raum werfen, ohne die Belege zu zeigen, also hier die Werte, die Microsoft veröffentlicht hat – gemessen an echten Codebasen, nicht an einem synthetischen Benchmark.

Codebasis	Altes `tsc`	Nativ (Go)	Beschleunigung
VS Code (~1,5 Mio. LOC)	~77,8 s	~7,5 s	**10,4×**
TypeORM (~270 Tsd. LOC)	~17,5 s	~1,3 s	**13,5×**
Editor-Projektladezeit	~9,6 s	~1,2 s	**~8×**

Zusätzlich zu den Wanduhr-Gewinnen fällt der Speicherverbrauch auf **etwa die Hälfte** der bisherigen Implementierung – was auf einem Laptop oder in einem speicherbegrenzten CI-Runner ebenso wichtig ist wie reine Geschwindigkeit. Die Editor-Zahlen sind die, die mir im Alltag am meisten bedeuten: Wenn Projektladen, „Gehe zu Definition" und Umbenennen sich sofort anfühlen, ist das eine andere *Qualität* des Arbeitens, nicht bloß eine schnellere.

> **Deine Werte werden abweichen – und das ist zu erwarten**
> Das sind Microsofts Zahlen für ihre gewählten Projekte. Eine ~10×-Beschleunigung bei 1,5 Mio. Zeilen sagt wenig über eine mit 5.000 Zeilen aus, bei der `tsc` bereits so schnell war, dass es dir nie auffiel. Die Gewinne sind genau dort am dramatischsten, wo der Schmerz am größten war: große Projekte, kalte Builds und stark beanspruchte Sprachdienst-Operationen. Miss dein eigenes Vorher und Nachher – nimm weder meine Begeisterung noch Microsofts Tabelle als Versprechen für dein Repository.

Was *heute* schon schneller ist

GA landete also am 8. Juli, und heute hat VS Code angekündigt, das native TypeScript integriert auszuliefern. Konkret, was sich für dich jetzt geändert hat:

1. Das Editor-Erlebnis, sobald du auf dem nativen Sprachdienst bist

VS Codes TypeScript-Unterstützung – Vervollständigungen, Hovers, Umbenennen, Gehe-zu-Definition – läuft über den Sprachdienst. Mit der nativen Engine als Antrieb werden diese Operationen spürbar flotter, besonders in großen Projekten. Auch die Visual-Studio-2026-Insider-Builds haben denselben nativen Sprachdienst in der Vorschau.

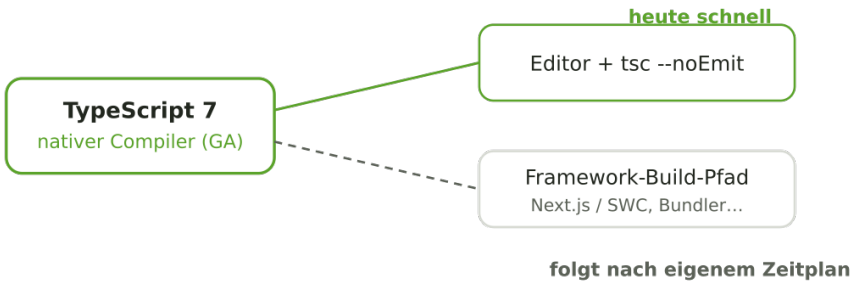
2. Typprüfung und Builds auf der Kommandozeile

Ein `tsc --noEmit` in der CI oder ein einfacher Build läuft jetzt durch den nativen Compiler. Hier leben die Schlagzeilen-Beschleunigungen, und es braucht keine Framework-Kooperation – es ist einfach der Compiler, der seine Arbeit schneller erledigt.

Was noch Handarbeit braucht (vorerst)

Hier kommt der ehrliche Vorbehalt und der Grund, warum ich diesen Beitrag so betitelt habe. **Frameworks, die ihre eigene TypeScript-Pipeline betreiben, erben den nativen Compiler nicht automatisch in dem Moment, in dem er GA ist.** Bundler, Dev-Server und Build-Werkzeuge binden `typescript` als Bibliothek ein, treiben ihre eigenen Transformationen an und entfernen Typen mitunter mit einem *separaten* Werkzeug – „TS7 ist draußen" legt also keinen Schalter in ihrem Inneren um.

Next.js ist der Fall, auf den ich zuerst gestoßen bin. Ein produktiver Next.js-Build stützt sich für die Ausgabe deines JavaScripts gar nicht auf `tsc` – er prüft die Typen mit TypeScript, transpiliert aber mit SWC, seinem Rust-basierten Compiler. Die native TypeScript-Engine hilft deinem *Editor* und deinem `tsc`-*Typprüfschritt* also sofort, aber sie in den eigenen Build-Pfad des Frameworks einzuweben ist eine separate Integrationsarbeit, die nach dem Zeitplan des Frameworks landet, nicht nach dem des Compilers.



GA macht heute Editor und tsc schnell; die Framework-Build-Pfade folgen nach eigenem Zeitplan

Das Diagramm sagt es klar: Der eine Pfeil ist durchgezogen und heute schon aktiv, der andere gestrichelt und im Kommen. Meine Einschätzung ist, dass die Framework-Seite Tage bis wenige Wochen zum Aufholen braucht – aber ich möchte klarstellen, dass **das meine Erwartung ist, kein ausgeliefertes Datum auf einer Roadmap, auf die ich verweisen könnte.** Nimm es als informierte Ungeduld, nicht als Versprechen.

> **Ausprobieren, ohne deinen Build zu verwetten**
> Wenn du den Unterschied heute spüren willst, lass die native Typprüfung neben deinem normalen Build laufen, statt etwas herauszureißen: Belass den Build deines Frameworks, wie er ist, und ergänze ein natives `tsc --noEmit` als separaten CI-Job oder lokales Skript. Du bekommst die schnelle Rückkopplungsschleife sofort und wartest nicht blockiert auf die Framework-Integration. Wenn dein Framework den nativen Compiler in die eigene Pipeline einwebt, entfernst du den zusätzlichen Schritt.

Ein faires Wort zu den Alternativen

Weil dieser Blog dir die gesamte Landschaft zeigen will: Das JavaScript-Ökosystem hat bereits *andere* sehr schnelle Typ-Werkzeuge, und TS7 löscht die Gründe nicht aus, aus denen man zu ihnen greift. Werkzeuge wie esbuild und SWC transpilieren TypeScript in atemberaubender Geschwindigkeit, indem sie **Typen entfernen, ohne sie zu prüfen** – eine andere Aufgabe als die von `tsc`. Node.js selbst kann inzwischen `.ts`-Dateien direkt ausführen, indem es Typen im Flug entfernt. Diese bleiben hervorragend, wenn du *Transpilierungs-geschwindigkeit* willst und die *Korrektheit* gern deinem Editor oder einem separaten CI-Schritt überlässt.

Was TS7 ändert, ist das Einzige, das nur der echte Compiler leisten kann – **vollständige, semantische Typprüfung** –, die von langsam zu schnell wird. Es ist also nicht „natives TypeScript gewinnt"; es ist „der eine Teil der Werkzeugkette, der noch langsam war, hat gerade mit den Teilen gleichgezogen, die bereits schnell waren". Wähle nach Bedarf: Transpilieren-und-los-Werkzeuge für rohen Build-Durchsatz, das native `tsc`, wenn du das volle Urteil des Typsystems in hohem Tempo willst.

Fazit

TypeScript 7 ist der Lohn eines wirklich ehrgeizigen Umschreibens, und die ehrliche Fassung der Geschichte ist interessanter als die Schlagzeile.

- **Es ist Version 7, und es ist nativ.** Der Go-basierte Compiler (Project Corsa) ist seit dem 8. Juli 2026 GA; 6.x war die letzte JavaScript-Linie. `tsgo` war nur der Vorschau-Name.
- **Go passt zur Arbeitslast.** Native Kompilierung, Goroutinen und gemeinsamer Speicher lassen die Typprüfung parallel laufen – etwas, das JavaScripts einsträngige Event-Loop nicht konnte. Eine nahezu zeilenweise Portierung hielt das Typprüfungsverhalten identisch.
- **~10×, bei großen Codebasen, für den Compiler – nicht die Sprache.** Deine Laufzeit ist unverändert; deine Builds und dein Editor werden schneller, am dramatischsten dort, wo es am meisten wehtat. Miss dein eigenes Repository.
- **Editor + `tsc` sind heute schnell; Frameworks folgen.** VS Code liefert es jetzt aus, aber Next.js und Co. müssen ihren eigenen Build-Pfad hindurchleiten – erwarte das in den kommenden Tagen und Wochen, und ergänze in der Zwischenzeit einen nativen `tsc --noEmit`-Schritt.

> Die Sprache hat sich nicht geändert. Der eine langsame Teil der Werkzeugkette hat endlich mit den schnellen Teilen gleichgezogen – und das ist Grund zur Freude.

Ich habe darauf gewartet, seit ich über Anders geschrieben habe und über die Überzeugung, die durch alles zieht, was er baut: schnelle Werkzeuge, ehrliche Typen, Compiler und Editor als ein Produkt behandelt. TypeScript 7 ist diese Überzeugung, ausgeliefert. Ich kann es kaum erwarten, zuzusehen, wie der Rest des Ökosystems dahinter aufleuchtet.

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 —