NEWS · OCTOBER 23, 2026

Not Only Delphi: How Three Quarters of 2026 Rewrote My Job

A prospect left my site thinking "this guy only does Delphi" -- so here is what three quarters of 2026 actually looked like, with TypeScript, Swift, Kotlin, C# and Delphi all shipped to production, and why the job is now architecture and verification, not fluency in one syntax.

By Dr. Holger Flick



A few weeks ago, a company was looking for someone to build an application for them. Someone pointed them to my site. They read it, and they walked away with one impression: "Oh, this guy only does Delphi. We don't use Delphi." I heard that second-hand, and it was precisely the feedback I didn't want to hear -- not because it was unfair, but because my own front page had earned it. The word "Delphi" appeared in the hero, in the stats, in the footer, and in the first service on the list.

Here is the thing, though. In the nine months since January, I have shipped production code in TypeScript, Swift, Kotlin, C#, SQL, and, yes, Delphi. None of that is a stretch, and none of it is new for me in the sense that matters. What is new is how the work gets done. In 2025, I would have described myself as a Delphi expert who also builds for the web. In October 2026, that description is simply wrong. I am the architect. I know the frameworks and the architectures, I decide what the system should be, and I direct the tools that now write most of the code -- in whatever language is right for the job. Then I verify every line, because 35 years in this business have taught me what correct looks like.

This post is the long version of what the front page should have said. You'll get a public sample of what I built this year and in which language, the argument for why "which language" has become the least interesting question in my work, the evidence for and against that argument, and -- because I'm not burning any bridges -- what all of this means if you're a Delphi developer looking at the same calendar.

What the site said, and what it should have said

The old front page wasn't lying; it was just a year out of date. The phrase "30+ years of Delphi" was true, the Embarcadero MVP title was true, and the Delphi-to-web migrations I listed as case studies were real and are still running. But a visitor without any Delphi history reads a page like that and reasonably concludes that the Delphi part is the point and the web part is a side effect. For most of my career, that reading would have been roughly right. In 2026 it isn't.

So I rewrote the page. The stacks I shipped this year are now listed where the stats used to brag about one language. The services lead with custom systems and apps for every platform, and the Delphi work -- migrations, consulting, code rescue -- is still there, as one service among seven. I'll come back to why it stays. First, some receipts.

A sample of what shipped in 2026, by language

Rather than claim versatility, let me show a sample. I want to be clear that this is a sample and not the ledger: most of my work is for customers and never gets a blog post or a case study, and there are more iOS and Android apps behind it than the ones named below. What the table does is pick, for each language, one or two things I can point to in public. Everything in it is either published on this site or described in an anonymized case study on the work page, with one honest exception I'll flag.

Language	A public example	Where to look
TypeScript, Next.js, React, PostgreSQL	This website, a warehouse management system, a B2B shop, an offline-first delivery app, a Delphi 7 application rebuilt for the browser	Case studies, the Delphi-to-Next.js series
Swift, SwiftUI	Hearth: apps for iPhone, iPad, Apple TV, and Mac on one backend, 313 automated tests, 48 releases in twelve days -- the one iOS project I can show in full	Build, Don't Buy, Not Vibe Coding
Kotlin	Android apps built on top of existing web applications, for customers	No post yet -- this is the exception; I'll write one up
C# and .NET	A C# report service read, understood, and folded into the Next.js app it belonged in; two .NET 10 book reviews along the way	From a C# Report Service to One Next.js Codebase, C# 14 and .NET 10
Delphi	FireDAC and PostgreSQL from scratch, WebStencils with SQLite, Delphi 13.2 on Linux, a native tool spec-coded in an afternoon	FireDAC CLI, WebStencils and SQLite, Spec coding with Delphi

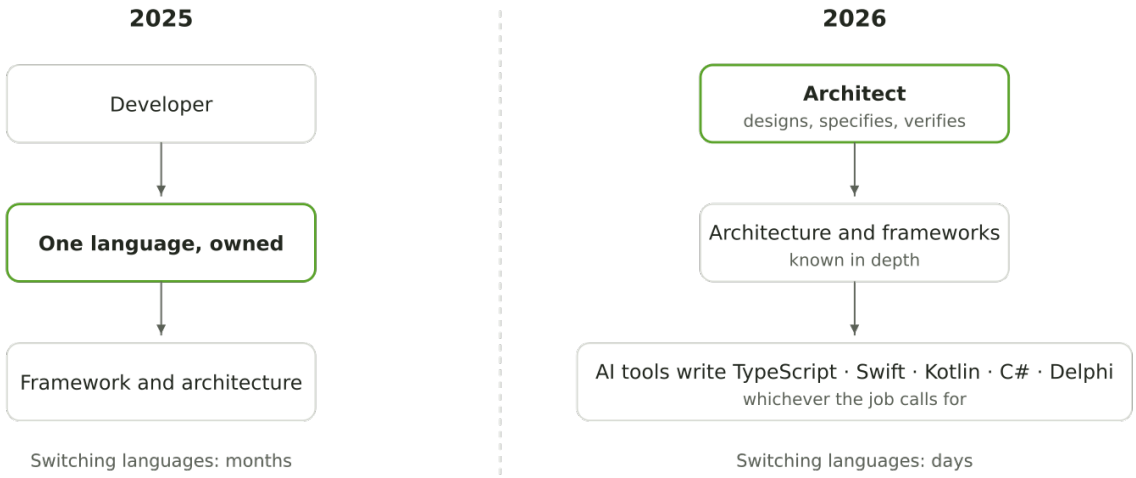
Read that table once more and notice what's missing: a line that says "spent six months learning Swift." There isn't one. The Swift apps exist because I knew what an iPad client of a Next.js backend with passkey sign-in should look like, I wrote that down, and I directed the tools until the result matched -- and then I tested it on the real device, as the Not Vibe Coding post documents in some detail. The Kotlin apps came about the same way, on top of a web app that already defined every rule. In my experience, having a working web application as the

base is what makes a native app a matter of days rather than months; the platform code is thin by design, and all the logic that can go wrong lives on the server where it was already tested.

The job changed; the expertise did not

Here is the part I think is genuinely new, and I want to be careful to describe it rather than hype it. For most of my career, the unit of expertise was the language. You were a Delphi developer or a C# developer or a Java developer, and switching meant months of becoming a worse version of yourself before you got good again. The frameworks and the architecture mattered, of course, but you accessed them through a syntax you had to own completely.

That is no longer how my days work. The diagram below shows what I mean, and it's the most honest picture I can draw of 2025 against 2026.



2025: a language specialist reaches the architecture through one syntax. 2026: the architect designs the system and directs tools into whichever language fits.

The left column is how I worked for roughly thirty years, and I'm not knocking it; it produced a lot of good software. The right column is how I worked this year. The thing to notice is which box has the thick border. In 2025, the language was the irreplaceable layer. In 2026, the irreplaceable layer is the person at the top who knows what the system should be, knows the frameworks and the architecture well enough to specify it precisely, and knows enough about all of those languages to read what comes back and reject what is wrong.

That last clause is the whole job now. The tools can produce idiomatic Kotlin faster than I can type it. What they cannot do is decide that family isolation belongs in database triggers rather than in application code, or notice that a sign-in screen on the iPad looks like a television's pairing screen, or recognize that a report that "looks right" is computing a tax the wrong way. Those decisions came from me, and the Not Vibe Coding post lists them one by one. Thus, I don't need to learn Kotlin the way I learned Object Pascal in 1995, with a thick book and a year of mistakes. I need to know Android architecture, I need to know what a correct Kotlin program looks like when I read one, and I need the discipline to read every one of them. I have the first two from decades of doing this across platforms, and the third is just character.

This is not vibe coding, and here is where that claim stops

I want to be explicit that the paragraph above is my opinion, and that I am aware it sounds like every "10x with AI" post you have scrolled past this year. So let me support it and then challenge it properly.

The support is simple: the process is documented and the numbers are public. Hearth went through 52 pull requests, each one described, reviewed, and tested, with 313 automated tests, 34 database isolation checks, and a deploy gate that we deliberately broke once to prove it refuses to ship. Simon Willison drew the line that I

find most useful: vibe coding is "building software with an LLM without reviewing the code it writes," and if you review it, test it thoroughly, and can explain how it works, it is simply software development. His golden rule is that he will not commit code he couldn't explain exactly what it does to somebody else. I hold myself to the same rule, and it is the rule that 35 years make possible: you cannot review Kotlin you cannot read, and you cannot read Kotlin well if you have never thought hard about how an Android app is supposed to be structured.

Now the challenge, and I take these seriously.

> **Where this doesn't hold**
>
> METR ran a randomized trial in early 2025 with 16 experienced open-source developers on 246 real issues in their own repositories. With AI tools allowed, they took 19 percent *longer* -- and still believed afterward that they had been 20 percent faster. The authors list the caveats themselves (small sample, developers new to the specific tool, results already superseded by their own 2026 follow-up), but the perception gap is the finding that should worry everybody, me included. The feeling of speed is not evidence of speed. The 2025 Stack Overflow Developer Survey found 84 percent of developers using or planning to use AI tools while 46 percent say they don't trust the output, up from 31 percent a year earlier. More use, less trust. And the 2025 DORA report finds that AI adoption now correlates with higher throughput *and* with higher instability -- more change failures, more rework. Their phrase is that AI "amplifies what's already there."

So here is the scoped version of my claim, which is the only version I'll defend. Directing AI tools across languages works when the person directing them already has the architectural judgment and the reading ability to catch what is wrong, and when the process forces that reading to happen: pull requests, tests, review, a deploy gate. Under those conditions, I have shipped more and better software this year than in any year before it. Without those conditions, the research says you will likely ship faster-feeling, less stable software and not notice. DORA's "amplifies what's already there" is exactly right: the tools amplified 35 years of experience in my case. They would amplify the absence of it just as faithfully.

> **The other side of the architecture argument**
>
> A fair objection: if the native apps are "thin by design" and the logic lives on the server, haven't I just chosen the easy case? Partly, yes. A Kotlin app whose job is to render data and talk to a well-defined API is a far smaller problem than a Kotlin app with complex local state, background processing, and offline conflict resolution. I'd still argue that pushing logic to one tested place is the right architecture for most business applications, and that this is precisely the kind of decision an architect makes before any language is chosen. But I wouldn't claim that every native app can be produced this cheaply, and a project that genuinely needs deep platform expertise still needs someone who has it.

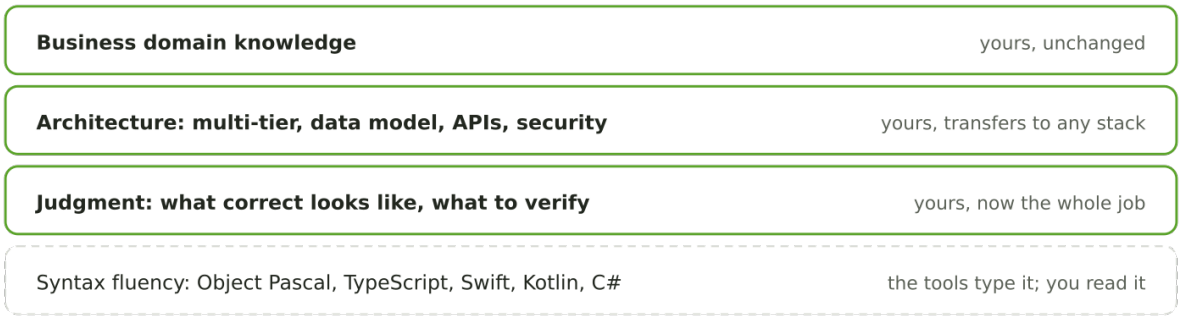
What this means for the Delphi community

Now the part I promised. I'm not leaving Delphi behind, and I want to say plainly why, because the honest numbers are not kind this year and I'm not going to pretend otherwise.

The TIOBE index for October 2026 puts Delphi/Object Pascal at 19th with 0.83 percent, down from 9th a year earlier. That is a steep drop in a ranking that is admittedly a measure of search interest rather than of software in production, and I'd caution anyone against reading a single index as a verdict. At the same time, Embarcadero shipped RAD Studio 13 Florence in September 2025 and 13.1 in March 2026, I wrote about running Delphi 13.2 on Linux in September, and I still get mail every week from people who maintain Delphi systems that run real businesses. Those systems are not going anywhere soon, and neither is the need for people who can read them.

My opinion -- and it is an opinion, so weigh it as one -- is that Delphi as a *primary career skill* is on its way out, in the sense that fewer new systems will be started in it, while Delphi as a *body of running software* will need expert care for a long time. If you have spent twenty years in Delphi, those are both good news, and here is why.

Everything that made you valuable is still valuable: you understand your business domain, you know how a multi-tier application is structured, you have debugged FireDAC transactions at three in the morning, and you know what a correct program looks like when you read one. What has changed is that the language is no longer the moat around that knowledge; it's now the cheapest part. That means the thing that kept you in Delphi -- the cost of becoming a beginner again in TypeScript or C# -- has mostly evaporated. You don't have to become a beginner. You have to become the architect you already are, out loud, and let the tools type.



A Delphi developer's assets, and which of them the tools change. The valuable layers stay yours; the syntax layer is what got cheap.

The three solid boxes are what a Delphi veteran brings to any project, and they are exactly what I brought to the Swift and Kotlin work this year. The dashed box is what used to be the barrier. My Delphi-to-Next.js series was written for this transition, and if you're on the Delphi side of it, that is still the first thing I'd hand you. The destination stack matters less than you think; the habit of reading everything before it ships matters more than ever.

> **If you'd rather stay in Delphi**
> That is a legitimate choice, and nothing here argues against it for the systems you already run. The same AI tooling works inside a Delphi project; I spec-coded a native Delphi tool in an afternoon and embedded a local language model in a Delphi app this year. The point is not the destination. The point is that the language stopped being the thing that limits you, in either direction.

What this means if you're hiring

If you're the company that walked away a few weeks ago, this section is for you, and I'll keep it short. Bring me the problem, not the language. Tell me the workflow that needs software, the systems it has to talk to, the people who will use it, and the devices they hold. I'll propose an architecture and the stack that fits it -- Next.js and PostgreSQL where a browser application is right, C# and ASP.NET where your team already lives in .NET, Swift or Kotlin where it must be native, Delphi where a working Delphi backend should stay exactly where it is. Then I'll build it with the tools, read every line, test it, and show you the pull requests.

Which language I "do" has become the least interesting question about my work. What I do is decide what the system should be and make sure it is that. In my experience, that is also the part customers actually pay for.

Takeaways

Three quarters of 2026 changed what my job is, and the front page finally says so. The short version is this.

- I shipped production code in TypeScript, Swift, Kotlin, C#, SQL, and Delphi this year. The table above is a public sample of it, not the whole list, and one of those languages I still owe you a post about.
- The job is now architecture, specification, and verification. The tools write the syntax; I decide what the system is and read everything that comes back. That works because of 35 years of knowing what correct looks like -- and the research says it would not work without that.
- The evidence against overclaiming is real: experienced developers measurably slower in METR's trial, trust in AI output falling, and instability rising where process is weak. My claim is scoped to a process with tests, reviews, and a deploy gate, and I'd scope yours the same way.
- Delphi stays. Its ranking fell hard this year and Embarcadero keeps shipping; both are true. For Delphi developers, the valuable layers -- domain, architecture, judgment -- are intact, and the barrier that kept you in one language has mostly dissolved.

> The language was never the expertise. It was the way in. In 2026, the way in is a specification and a careful reader, and I have become the second.

If you have a system to build and no particular attachment to how it's written, let's talk. And if you're a Delphi developer looking at the same calendar I am, start with the series, and write to me when you get stuck. That bridge is staying up.

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 —