NEWS · FEBRUARY 23, 2026

Adapt or Disappear: How AI Turned a 2-Year Project Into a 1-Week Sprint

By Dr. Holger Flick

The Delphi community has strong opinions about AI — and I get it. Privacy concerns, code quality worries, the instinct that real developers should not need a machine to help them write code. I have heard every argument in the book, and honestly, some of them are fair. The skepticism is real, though the tide may be turning. Still, the doubters outnumber the converts, and I am not here to pretend otherwise. But after what I just witnessed on a real-world migration project, I think even the skeptics will have a hard time arguing with the results. This is not about replacing developers — it is about what happens when developers stop fighting the tool and start using it.

The Project

A customer approached us with a legacy desktop application built in **Delphi 7**. Let that sink in for a moment.

Delphi 7 — released in 2002. No Unicode support. No HiDPI awareness. Running on **Firebird 1.5** — a database engine so old its last update predates the iPhone. The application had been serving its purpose for over two decades, but the world had moved on. No responsive design. No web access. No mobile support. Customers had been begging for a web-based solution for years, but the cost and complexity of a ground-up rewrite in the legacy stack — let alone migrating to a modern platform — had always been prohibitive.

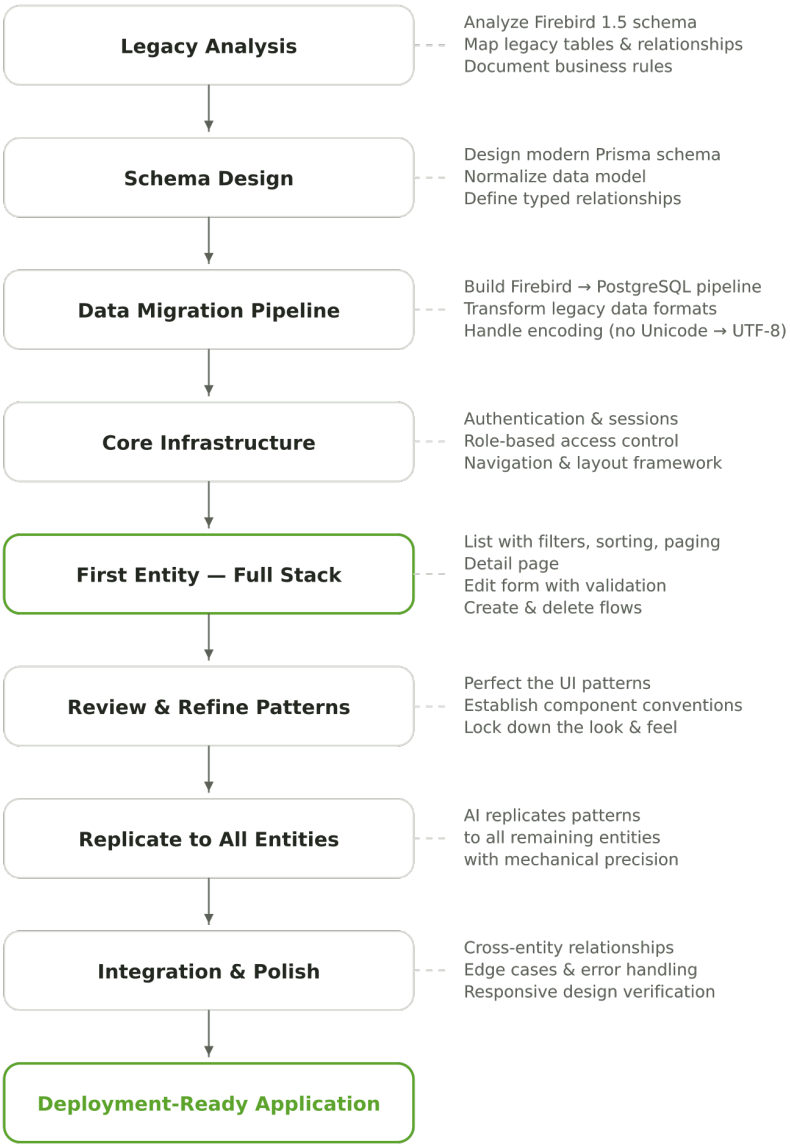
The target: a full-stack TypeScript application built on Next.js with server-side rendering, a component-based UI, Prisma ORM for database access, and a responsive design that works on both desktop and mobile devices. Not just a facelift — a complete architectural transformation from a Win32 desktop application to a modern, deployable web platform.

This was not a toy project. It involved user authentication, role-based access control, property management workflows, incident tracking, image handling, complex relational data models, and a migration pipeline to move real production data from the legacy Firebird database. The kind of application that, in any traditional development shop, would require a team and a timeline measured in months — which is precisely why it had not been done in over twenty years of customers asking for it.

How We Did It

A migration like this cannot be done in a single big bang. You do not simply point an AI at a Delphi 7 project and say "make it modern." It requires a deliberate, phased approach — and critically, it requires **two kinds of expertise working in tandem**: deep knowledge of the legacy system (its data structures, its business rules, its quirks accumulated over two decades) and deep knowledge of the modern target platform (its architecture patterns, its security model, its deployment pipeline).

Here is how the process unfolded:



The phased migration process, from legacy analysis to a deployment-ready application

Phase 1 — Legacy Analysis & Schema Design. The developer with legacy expertise mapped out the existing Firebird 1.5 database: every table, every relationship, every implicit business rule buried in stored procedures or application code. This was not something AI could do alone — it required institutional knowledge of *why* certain fields existed, which tables were still active, and where the data had accumulated inconsistencies over twenty years.

Phase 2 — Data Migration Pipeline. We built a migration pipeline that could read directly from the legacy Firebird database and transform the data into the new schema. This included converting character encodings (the old system had no Unicode — every special character was a potential landmine), normalizing denormalized data, and handling two decades worth of data quality issues. The AI generated the migration scaffolding; the legacy expert validated that the transformations preserved business meaning.

Phase 3 — Core Infrastructure. Authentication, session management, role-based access control, the navigation framework, the layout system — all the plumbing that every page would depend on. This was built once, carefully, with both developers reviewing the security model.

Phase 4 — The First Entity. This was the most time-intensive phase. We built one complete entity — list view with filtering, sorting, and pagination; detail page; edit form with validation; create and delete flows with confirmation dialogs — and refined it until every interaction pattern, every visual detail, every error state was exactly right. This became the template.

Phase 5 — Pattern Replication. This is where AI transformed the economics of the project. With the first entity perfected, the AI replicated the exact same patterns across every remaining entity in the system. What would have been weeks of tedious, error-prone copy-paste work became hours of guided generation and review.

Phase 6 — Integration & Polish. Cross-entity relationships, edge cases, responsive design verification across device sizes, and final refinements. The application was deployment-ready.

Each phase built on the previous one. Each phase combined human expertise with AI execution. Neither could have done it alone — and together, they accomplished in one week what neither could have done in six months.

The Numbers

Within **one week**, the project accumulated approximately **50,000 lines of code**. But raw line counts do not tell the full story. Here is how that breaks down:

Category	Lines of Code	Description
Frontend — Pages & Layouts	~10,200	Server and client components, forms, data tables, dialogs, responsive layouts
Frontend — Custom Components	~1,950	Reusable components like searchable selectors, navigation, shared UI elements
Backend — Server Actions & Business Logic	~3,200	Data access, validation, CRUD operations, authentication, migration logic
Backend — Database Schema & Config	~525	Prisma schema with 20+ models, app configuration
Backend — Generated ORM Client	~29,100	Auto-generated type-safe database client (see below)
UI Component Library	~2,750	Imported and customized component primitives — not written from scratch
Styling & Other	~125	CSS, Tailwind configuration
Total	**~50,000**	

Strip away the generated ORM client and the imported component library, and you are still looking at approximately **16,000 lines of hand-crafted, production-grade TypeScript** — frontend and backend combined — written, tested, debugged, and refined in a single week.

A Note on the ORM: 29,000 Lines You Never Have to Write

Those ~29,000 lines of generated Prisma client code deserve special attention — because they illustrate a point that the Delphi community should find particularly uncomfortable.

In a modern TypeScript stack, you define your data model **once** in a schema file — 438 lines in our case, covering 20+ entities with their relationships, constraints, and types. From that single source of truth, Prisma generates a fully type-safe database client: every query is checked at compile time, every relation is validated, every field type is enforced. You cannot write a query that references a column that does not exist. You cannot pass a string where the schema expects an integer. The compiler catches it before the code ever runs.

In the Delphi world, developers that neglect ORM are still writing this kind of data access code **by hand**.

Every single time. TDataSet wrappers, field-by-field mappings, manual SQL strings, hand-rolled object-relational mapping layers — thousands of lines of tedious, error-prone boilerplate that adds zero business value and introduces bugs with every typo. Many Delphi shops have spent years building their own ORM frameworks, and most of them still cannot match the type safety that a modern generated client provides out of the box.

This is not about AI versus humans. This is about choosing to do grunt work manually when the tooling exists to eliminate it entirely. And if you are still hand-coding your data access layer in 2026, AI is not your biggest problem — your entire development philosophy is.

What This Would Normally Cost

Before anyone dismisses these numbers, let us talk about where they come from — because this is one of the most well-studied metrics in software engineering.

The figure of **50 to 80 lines of production code per day** is not an insult to developers. It is a reality confirmed across decades of industry research:

- **Fred Brooks**, in *The Mythical Man-Month* (1975), documented that experienced IBM developers produced roughly 10 lines of debugged, documented code per day on the OS/360 project — one of the largest software efforts of its era. That was for systems programming, but the principle has held: writing code is the easy part. Getting it *right* is what takes time.
- **Steve McConnell**, in *Code Complete* and *Software Estimation: Demystifying the Black Art*, reports industry averages of **40 to 80 lines per day** across projects of varying complexity. For complex enterprise systems, it skews lower. For simpler CRUD applications, it skews higher.
- **The COCOMO II model** (Constructive Cost Model), developed by Barry Boehm at USC and used by organizations worldwide for project estimation, produces similar ranges when you factor in all phases of development.
- **Microsoft's internal data**, shared publicly by various engineering leads over the years, suggests their developers average roughly **50 to 100 lines of shipping code per day** — and these are among the best-resourced engineering teams on the planet.
- **A 2017 study by DORA** (DevOps Research and Assessment, now part of Google) found that even elite-performing teams, with the best CI/CD pipelines and automation, still face fundamental throughput limits driven by code review, testing, and integration.

Why so "low"? Because production code is not just typing. A developer's day includes understanding requirements, reading existing code, designing solutions, writing tests, debugging failures, participating in code reviews, refactoring for maintainability, writing documentation, attending standups, and — let us be honest — recovering from the mental fatigue of holding complex systems in your head. The actual *typing* is perhaps 10-20% of the job.

This is not controversial. Any experienced developer knows this intuitively. The 50-80 lines figure is not a measure of typing speed — it is a measure of **the full cost of producing reliable, maintainable software**.

Using a conservative midpoint of 65 lines per day, here is what 16,000 lines of hand-written code would require under traditional development:

- **Single developer**: ~246 working days — roughly **one full year**
- **Small team (2-3 developers)**: 4 to 6 months, factoring in coordination overhead, merge conflicts, meetings, and the communication tax that Brooks himself warned about
- **Realistic cost** (at a modest rate of $80/hour): **$160,000 to $250,000+**

Even a senior full-stack developer with deep expertise in the exact tech stack, working at an aggressive 120 lines per day — which would place them well above industry average — would need **133 working days — over six months**.

We did it in **one week**. With two developers — one with deep knowledge of the legacy system and its business logic, the other with extensive experience in the modern target stack — and an AI pair programmer.

That is not a marginal improvement. That is a **15 to 25x productivity multiplier**.

The Pattern Effect: Build Once, Replicate Everywhere

One of the developers on this project made an observation that deserves its own section, because it reveals something fundamental about how AI changes the development workflow.

A business application like this one is, at its core, a series of variations on a theme. You have entities — users, invoices, line items, products, ... — and for each entity you need roughly the same set of views: a filterable, sortable, paginated list. A detail page. An edit form with validation. Create and delete flows with confirmation dialogs. Breadcrumbs, navigation, access control.

In traditional development, you build the first one and then — if you are disciplined — you extract patterns, create templates, maybe build a code generator. More often, you copy-paste from the first implementation, adapt it, and hope you catch all the places where "User" needs to become "Property." This is where bugs breed. Every copy-paste is a lottery ticket for a missed rename, a forgotten field, a broken reference.

With AI, the workflow is fundamentally different. You build the first entity's full stack — list, filters, pagination, detail page, edit form, server actions — and you get it exactly right. You refine the look and feel, the interaction patterns, the error handling, until it matches the vision. Then you tell the AI: *"Now do the same for Properties.*

Follow the exact same patterns, the same UI conventions, the same component structure."

And it does. Perfectly. Every time. No copy-paste errors. No forgotten renames. No drift in UI conventions between the first entity you built on Monday and the eighth one you built on Friday when you were tired. The AI applies the established pattern with mechanical precision, and you review the output to ensure it matches your intent.

This is where the productivity multiplier truly compounds. The first entity might take a few hours of collaborative work. The second takes minutes. By the fifth, you are producing complete, production-ready feature sets faster than you could even *read* the equivalent hand-written code.

"But AI Code Is Insecure and Low Quality"

Let us address the elephant in the room. The argument from parts of the Delphi community — and from skeptics in general — is that AI-generated code is somehow inferior: riddled with security vulnerabilities, poorly structured, unmaintainable.

Here is what actually happened on this project:

- **Server-side rendering by default.** The architecture uses React Server Components, minimizing client-side attack surface. Sensitive logic never ships to the browser.
- **Type safety everywhere.** Full TypeScript with strict typing, Prisma's type-safe query builder, and typed routes. Entire categories of bugs are eliminated at compile time.
- **Proper authentication and authorization.** Session management, password hashing with bcrypt, role-based access control — all following established security patterns.
- **No API routes exposed.** Server Actions handle mutations directly, reducing the surface area for injection attacks.

Was the code perfect from the first keystroke? Of course not. But the AI did not just generate code — it engaged in an iterative dialogue. It flagged potential issues. It suggested improvements. It caught mistakes that a tired developer at hour ten of a long day would have missed.

The code quality was not worse than what a human team would produce. In many cases, it was more consistent — because an AI does not have bad days, does not cut corners before a deadline, and does not forget to add input validation because it is Friday afternoon.

"I Would Have to Read All 50,000 Lines to Trust It"

This was the immediate response from one skeptic when shown the numbers. And on the surface, it sounds reasonable — how can you trust code you have not personally inspected line by line?

But let us turn that question around and apply it honestly.

How much of your own codebase have you actually tested? Not read — *tested*. With automated tests.

With mathematical rigor. Be honest. If you are a Delphi developer maintaining an application that has been in production for 10, 20, or even 30 years — how much of that code has proper unit test coverage? How many of those business logic paths have automated regression tests that run on every build?

The answer, for most legacy projects, is: very little. Maybe some critical paths. Maybe a handful of tests written years ago that no one is sure still pass. The rest is held together by institutional knowledge, manual testing, and the hope that "it has been working fine for years." That is not security. That is survivorship bias.

Now consider this: AI does not just write application code. It writes **tests**. You can instruct it to generate unit tests, integration tests, and end-to-end tests as part of the development workflow. You can require that every server action has corresponding test coverage. You can wire automated testing into the CI/CD pipeline so that nothing gets deployed without passing. The AI does not forget to write tests because the deadline is tomorrow. It does not skip test cases because they are tedious. It does not argue that "we will add tests later" — a promise every developer has made and most have broken.

And here is the question that should make every skeptic uncomfortable: **Do you apply the same scrutiny to your third-party libraries?**

Every modern application — Delphi included — relies on third-party code. Component libraries, database drivers, encryption packages, HTTP clients, JSON parsers. Thousands, sometimes tens of thousands of lines of code written by strangers on the internet. Have you read all of it? Have you audited every dependency for security vulnerabilities? Have you verified that the logging library you pulled from GetIt or npm does not phone home?

Of course you have not. Nobody does. You trust it because it is widely used, because it has a community behind it, because other people have presumably reviewed it. You apply a *reasonable standard of trust* based on provenance and track record.

AI-generated code deserves the same rational evaluation — not a higher bar born from fear. It should be reviewed, tested, and validated. Just like every other piece of code in your project. The difference is that with AI, you can also have the AI itself write comprehensive tests, perform security audits, and flag potential vulnerabilities — at a scale and consistency that no manual process can match.

Why AI-Generated Code Is Often *Safer* Than Senior Developer Code

The argument is usually framed as "AI code versus human code," with the implicit assumption that human code is the gold standard. But is it?

- **AI does not have ego.** Senior developers cling to patterns they learned a decade ago — SQL concatenation, MD5 hashing, rolling their own auth. AI defaults to `bcrypt`, parameterized queries, and `httpOnly` cookies — not because it is smarter, but because it has no attachment to the old way.
- **AI does not get tired.** The most dangerous code is written at 4pm on a Friday or during a late-night hotfix. Fatigue is one of the leading contributors to security vulnerabilities. AI applies the same rigor to the last file of the day as it does to the first.
- **AI has no blind spots.** A developer who has stared at the same codebase for months *assumes* the input is sanitized because they remember writing the sanitizer — three years ago, before someone refactored it. AI checks. Every time.
- **AI produces consistent code.** Five developers writing the same pattern across fifty files produce five different interpretations. AI applies the *exact same pattern* everywhere. Inconsistency is where attackers find gaps.
- **AI knows more vulnerability patterns than any individual.** Trained on the OWASP Top 10, CVE databases, thousands of security audit reports. No single human holds all of that in working memory simultaneously.
- **AI leaves a complete audit trail.** Every interaction is logged. Every instruction, every generated file, every revision. Try that with a developer who wrote the code from memory and left the company two years ago.

None of this means AI code should be trusted blindly. It means the standard should be the same as for any other code: review it, test it, deploy it through a proper CI/CD pipeline. The difference is that AI makes it trivially easy to *also* generate the tests and the security validation — something human teams perpetually defer because they are "too busy shipping features."

The Real Threat Is Not AI — It Is Irrelevance

Here is what concerns me about the anti-AI position in communities like Delphi: it is not actually a technical argument. It is an emotional one. And I understand that. When you have spent decades mastering a craft, it is deeply unsettling to watch a machine replicate significant parts of your output in minutes.

But the market does not care about our feelings. Customers care about results, timelines, and costs. When two developers with AI can deliver in a week what traditionally required a larger team and months of budget, the economics are devastating for anyone who refuses to adapt.

This is not hypothetical. This is what happened on this project. **A real customer. A real application.** Real production code. One week.

The developers who dismiss AI today are not protecting their craft. **They are accelerating their own obsolescence.** The future does not belong to developers who write every line by hand — it belongs to developers who know how to **direct, review, and refine** AI-generated output. Who understand architecture well enough to guide the machine. Who can evaluate code quality, security implications, and design tradeoffs.

In other words: **the future belongs to developers who are willing to evolve.**

The Bottom Line

Metric	Traditional	With AI
Timeline	6–12 months	1 week
Team size	2–3 developers	2 developers + AI
Estimated cost	$160,000–$250,000+	A fraction
Code quality	Variable	Consistent, reviewable
Data access code	Hand-written, error-prone	Generated, type-safe

And for those who want to argue economics — let us talk about licensing costs:

Technology	Role	License Cost
Next.js	Web framework	Free (MIT)
TypeScript	Language	Free (Apache 2.0)
React	UI library	Free (MIT)
UI Component Library	Buttons, dialogs, forms — with full source code	Free (MIT)
Prisma ORM	Type-safe database access	Free (Apache 2.0)
PostgreSQL	Production database	Free (PostgreSQL License)
Linux	Server operating system	Free (GPL)
Visual Studio Code	IDE	Free (MIT)
AI Coding Assistant	The pair programmer that built all of this	**$20/month**

Compare that to a traditional desktop development stack where the IDE, the database server, the operating system for deployment, and third-party component suites all carry commercial license fees — per developer, per year. Those costs add up quickly, especially for small shops.

The entire technology stack behind this 50,000-line application costs **$20 per month**. That is not a typo.

AI is not evil. It is not a threat to good developers. It is the most powerful force multiplier our industry has ever seen.

The only question is whether you will use it — or be replaced by someone who does.

Still Stuck in the Delphi 7 World? There Is a Way Out.

If this story sounds familiar — if you are maintaining a legacy Delphi application that your customers have outgrown, if you have been told a migration is "too expensive" or "too risky," if you have been putting off the inevitable because the scope felt insurmountable — we have been there. We just proved it does not have to be that way.

FlixEngineering LLC specializes in exactly this kind of transformation. We combine deep legacy system

expertise with modern web development and AI-assisted workflows to deliver migrations that would have been economically impossible just two years ago.

Whether your application runs on Delphi 7, Delphi 10, or anything in between — whether your data lives in Firebird, Interbase, MS SQL, or Oracle — we can help you chart a path from desktop to web, from legacy to modern, from "we have always done it this way" to "we cannot believe we waited this long."

The technology exists. The methodology is proven. The only question is how much longer you can afford to wait while your competitors move forward.

Get in touch!

This post is based on a real migration project completed in February 2026. Technology stack: Next.js, TypeScript,

React Server Components, Prisma ORM, Tailwind CSS. Client details have been anonymized.

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 —