NEWS · APRIL 2, 2026

# Spec Coding with Delphi: How I Built a Native Game Save Backup Tool in One Afternoon with Claude Code

By Dr. Holger Flick

> **Note**
> This is not an April Fools joke. This is a real project I built in an afternoon using Claude Code and Delphi. It is a command-line tool that scans my PC for installed games, identifies their save file locations, and creates backups with optional S3 upload. The entire implementation was generated by Claude based on my specifications, and it compiles to a single native executable with zero dependencies beyond `sqlite3.dll`. This is the power of spec coding.

There are a handful of tools out there for backing up PC game saves. They all share the same problems: bloated UIs, slow performance, confusing workflows, and the kind of feature creep that turns a simple task into a chore. I wanted something different. Something fast, something I control, something that just works.

So I built my own. In an afternoon. With Claude Code as my pair programmer.

## The Database That Knows Every Game

The foundation of this project is a SQLite database that has been curated for years by the GameSave Manager community. It contains entries for thousands of PC games -- every major title you can think of and many you have forgotten about. For each game, the database knows:

- Where save files live (AppData, Documents, Steam userdata, registry-derived paths)
- Which files to include and exclude
- How to detect if a game is installed via registry keys
- Platform-specific paths for Steam, Steam Cloud, Uplay, GOG, and more

This is not a toy dataset. It covers over 12,000 games with nearly 20,000 directory entries and hundreds of registry detection rules. The path resolution alone involves 13 different special path types that map to Windows shell folders, Steam directories, and dynamically resolved registry values.

## From Database Schema to Working CLI in Hours

I gave Claude Code two things: the database structure and the SQL dump. No UI mockups. No architecture diagrams. No hand-holding.

What I did provide were **specifications**. I know Delphi. I know what I want. I described the behavior:

- A `scan` command that iterates every game in the database, checks the Windows registry and filesystem, and reports which games are installed
- A `list` command to review scan results without re-scanning
- A `backup` command that collects save files into a zip with a manifest for future restore

- An `-s3` flag that uploads the backup to my self-hosted S3-compatible storage

Claude Code generated the entire implementation: a native SQLite3 wrapper unit, Windows registry access with both 32-bit and 64-bit hive support, recursive file collection with wildcard pattern matching, path resolution for all 13 special path types, zip creation with embedded metadata, and S3 upload using Wagner Landgraf's AWS SDK for Delphi.

When things did not compile, I pasted the errors. Claude fixed them. When the S3 signature calculation failed against my RustFS server, we debugged it together. When Steam Cloud paths turned out not to follow the documented `\remote` subfolder convention, we adapted. This was not a one-shot prompt. It was a working session between a developer who knows the domain and an AI that can write solid Delphi code.

## Why Delphi, Why Native

The existing tools in this space are built on Electron, .NET, or similar managed runtimes. They are slow to start, heavy on resources, and their UIs get in the way of a task that should take seconds.

My tool is a single 4 MB executable (with debug code). It starts instantly. Scanning 12,000+ games against the registry and filesystem completes in seconds because it is native compiled code calling Win32 APIs directly. There is no garbage collector pause, no JIT warmup, no framework overhead.

This matters when you want to automate backups. A tool that takes 30 seconds to load a UI before you can click three buttons is not a tool you will use regularly. A CLI that runs in a script or a scheduled task is.

## Spec Coding vs. Vibe Coding

There is a term floating around the AI-assisted development space: **vibe coding**. It describes the practice of

prompting an AI with vague intentions and hoping something useful comes out. Sometimes it works. Often it produces generic, framework-heavy code that sort of does what you wanted.

What I did here is the opposite: **Spec Coding**.

Spec coding means you know your tools, your language, your platform. You know what `KEY_WOW64_32KEY` means. You know why `TIniFile.Create` needs an absolute path. You know that Wagner's AWS SDK takes ownership of the config object. You bring the domain expertise and the architectural decisions. The AI brings the velocity.

The difference is not subtle. A vibe coder would have ended up with an Electron app, a React frontend, and a Node.js backend talking to SQLite through three layers of abstraction. I ended up with 950 lines of Delphi that compile to a single native executable with zero runtime dependencies beyond `sqlite3.dll`.

## The AI Amplifier Effect

Claude Code did not replace my judgment. It amplified it.

I decided to use Delphi. Claude wrote the SQLite wrapper. I specified the scan logic. Claude implemented the registry traversal. I caught that Steam Cloud paths do not always have a `\remote` subfolder. Claude fixed it to scan all user directories. I knew Wagner's SDK existed. Claude looked up the API, found the correct constructor overloads, and wired up the S3 upload with path-style access for my custom endpoint.

Every decision was mine. Every line of code was produced at a speed that would have been impossible alone. An afternoon instead of a week.

This is what AI does for developers who know their craft. It is not a replacement. It is a force multiplier. The developers who understand their languages, their platforms, and their problem domains will use AI to produce in hours what used to take days. They will build exactly what they need instead of settling for whatever framework is trending.

## The Developers Who Will Be Left Behind

There is an uncomfortable truth in all of this. AI-assisted development rewards expertise. If you know what to ask for, you get exactly what you need. If you do not, you get generic solutions to generic problems.

Developers who refuse to integrate AI into their workflow are not just missing a productivity tool. They are falling behind colleagues who ship in an afternoon what used to be a week-long project. The gap will only widen.

But the developers who will be hit hardest are not the skeptics. They are the ones who lean on AI as a crutch without building real expertise. Vibe coding works until it does not. When the generated code fails in production, when the S3 signature does not match, when the memory manager throws an `EInvalidPointer` during SDK finalization -- you need to actually understand what is happening.

The future belongs to developers who are **both** deeply skilled in their craft **and** fluent in working with AI. Spec coding is not about letting AI do the thinking. It is about thinking clearly and letting AI do the typing.

## The Result

The result is a command-line tool that does exactly what I need:

```
FlixSaves scan                          # Find installed games
FlixSaves list                          # Review without re-scanning
FlixSaves backup 12824                  # Backup Crimson Desert locally
FlixSaves backup -s3 11358 12824 12093  # Backup and upload to S3
```

Every backup includes a manifest with metadata: game ID, timestamps, computer name, source paths, and a complete file listing. The zip structure preserves the original path context for future restore.

It is fast. It is simple. It does one thing well. And it took an afternoon.

That is spec coding. That is the future.

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 —**