A Case for Re-implementation

Putting AI to work inside business applications, so the people who run the business can shape them, and rebuilding the systems that can't take it

Lonnie Watson, Software Architect

November 2026

harlock123.com

With working examples from seven applications: **AVA Modeler**, **AVALightWave**, **CirqAvalonia**, **AVAPageStream**, **AVASvgMaker**, **AVAUIDesigner** and **X12 Explorer**. Every screenshot in this document is of software that exists and runs today, and every capability described is covered by its automated tests. Most chat screenshots were taken by those test suites: the model's side of the conversation is replayed from a script so the pictures can be retaken at will, and every tool call in them ran for real against the application.

Contents

1. The short version.....
2. The problem: business needs outrun the IT queue.....
3. The opportunity: an AI layer over the application.....
   3.1 What MCP is.....
   3.2 Three ways in.....
4. Empowering business users: the techniques.....
   4.1 Expose the command layer as tools, in business language.....
   4.2 Put the chat inside the application.....
   4.3 Saved recipes: plans users can start from.....
   4.4 From conversation to process flow.....
   4.5 Safe to experiment.....
   4.6 A person in the loop where it matters.....
   4.7 Governance built in, not bolted on.....
   4.8 Test the AI layer like the rest of the code.....
5. Tools for building it.....
6. Working examples.....
   6.1 AVA Modeler: a 3D modeler anyone can direct.....
   6.2 AVALightWave: a whole scene from a description.....
   6.3 CirqAvalonia: an AI that checks its own work.....
   6.4 AVAPageStream: documents produced on request.....
   6.5 AVASvgMaker: from source code to a flowchart.....
   6.6 AVAUIDesigner: data-entry screens from the database itself.....
   6.7 X12 Explorer: the shape of a PHI pilot.....
   6.8 What this looks like in a business application.....
7. A case for re-implementation.....
   7.1 When the layer can't be added.....
   7.2 Examining a system in flight with AI.....
   7.3 Rebuild with the AI layer designed in.....
   7.4 Evidence from these projects.....
   7.5 The scale of what was built.....
   7.6 Risks, and how to manage them.....
8. A proposed path.....
   8.1 Track A: empower an existing application.....
   8.2 Track B: re-implement a system that can't take the layer.....
   8.3 What success looks like.....

Appendix: the applications.....

1. The short version

- Business users and product people wait on IT for changes that are, at heart, **a few existing operations in a new order**: a report run a different way, a month-end routine, a bulk correction, a document assembled from data. Each one becomes a ticket, a queue and a release.
- The **Model Context Protocol (MCP)** lets an AI model operate an application through the same commands its screens use, with every step visible, checked and undoable. Add a **chat panel** inside the application, and a business user can describe an outcome and watch the application produce it.
- The step that removes the ticket is the next one: what the user and the AI worked out together is **kept as a process flow**, a saved recipe or a steps file with named parameters, that anyone can run again by name. Business users author flows; IT owns the tools and the guardrails they run inside.
- I have built this into seven applications of my own (screenshots in section 6), including a HIPAA X12 viewer. The pattern is **well understood engineering, not research**, and it works with Claude, with OpenAI and other cloud services, and with open-weight models running on hardware the client controls.
- Some systems can't take an AI layer: no API, a closed or aging platform, no tests, knowledge that has left with the people who built it. For those, AI changes the economics of **re-implementation**. It can examine the system in flight (its screens, files, data, manuals and code), help write the specification and the tests that prove parity, and rebuild it on a base that can then grow functions the original was never going to support. Four of the seven applications here are re-implementations of older software.

What I'm proposing

1. Pick one client application and one repetitive, ticket-driven task. Put an MCP layer and a chat panel over its existing commands, read-only first, on a model that suits its data.

2. Let its business users save what works as process flows, and measure the tickets that don't get raised.

3. In parallel, assess one system that can't take this layer as a candidate for AI-assisted re-implementation.

2. The problem: business needs outrun the IT queue

In most organizations we serve, the people closest to the work know exactly what they need from their systems, and can't get it without a developer. The pattern repeats across healthcare, government and commercial clients:

- **Process flows live in people's heads.** "Every month, pull the denied claims, group them by reason, correct the ones with a bad modifier, resubmit, and send the rest to the review team" is twenty clicks across four screens, done by hand, the same way, every month.
- **Configuration is code.** A new report, a new export format, a rule that applies to one region: each is a change request, an estimate, a sprint and a release, even when every piece already exists in the application.
- **Workarounds become shadow IT.** Spreadsheets, macros and copy-paste between systems fill the gap, outside any audit, validation or backup.
- **IT spends its time on small changes**, not on the platform work only IT can do.

None of this is a failure of the application. The operations are there; what's missing is a way for a business user to **combine them safely without a developer in the loop**. That is precisely what MCP and a built-in chat provide.

3. The opportunity: an AI layer over the application

3.1 What MCP is

MCP is an open standard, supported across the major AI vendors and developer tools, for letting an AI model use an application. The application publishes a catalogue of operations (add a part, run a query, open a claim, lay out a page), each with a plain-language description and a schema for its arguments. The model reads the catalogue, decides which operations to call, and sees their results. **The application stays in charge**: it validates every call, enforces permissions and records what was done. The MCP layer sits on top of commands the application already has, so its screens, business rules and database don't change.

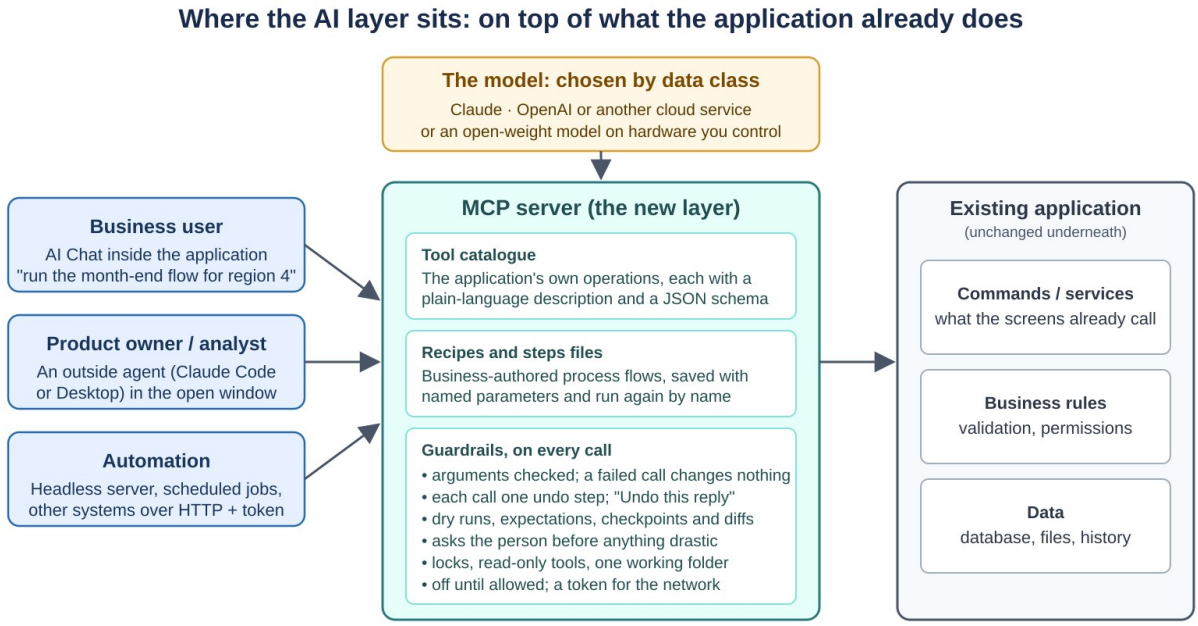


Figure 1. The AI layer: one tool catalogue and one set of guardrails for people, agents and automation, over the commands the application already has.

3.2 Three ways in

Who	How they reach the application	Typical use
Business user	An AI Chat panel inside the application. No other software; the conversation and the work happen in the same window.	"Make the quarterly summary for Ohio", "fix the eligibility dates on these members", "lay out this month's newsletter"
Product owner, analyst, power user	An outside agent (Claude Code, Claude Desktop or any MCP client) attached to the open application, on the same computer or over the network with a token.	Building and refining process flows, bulk work across files, combining this application with others the agent can also reach
Automation	The same server headless, with no window, on a schedule or called by another system over HTTP.	Running a saved process flow nightly; batch conversions; integration without a bespoke API

All three go through the same tools, so a flow a business user works out in the chat is the same flow automation can run unattended, with the same checks.

4. Empowering business users: the techniques

These are the techniques that turn "an AI that can click buttons" into something a business can trust business users with. Each is implemented in the applications in section 6; the names in brackets are the actual tools.

4.1 Expose the command layer as tools, in business language

Start from what the application already does. Each tool wraps an existing command or service and describes it the way a user would: what it does, what each argument means, its units and limits, and what to do when it fails. Errors say **what to do instead**, not just that something went wrong, because the model reads them and corrects itself. A tool catalogue is typically a few dozen operations, not the whole application.

4.2 Put the chat inside the application

A chat panel docked in the application's own window means no new software for the user, and no context switching: they ask, and watch the screen change. The panel shows every tool call as a line that opens to its exact input and result, and the pictures the model looked at, so nothing happens out of sight. The model is **a setting, not a dependency**: Claude, OpenAI or another cloud service, or an open-weight model on a workstation, server or private cloud, chosen by the data the application holds.

4.3 Saved recipes: plans users can start from

A recipe is a short, named, step-by-step plan written with the real tool names and arguments, served to the model as an MCP prompt ("two-column newsletter", "555 timer", "product shot"). Users pick one, adjust it in plain language, and the model follows it. In these applications **every recipe is run call by call by the automated tests**, so a recipe that stops working fails the build instead of failing a user.

4.4 From conversation to process flow

This is the technique that removes the ticket. The application records the steps a chat session took; the user (or the model, asked to) saves them as a **steps file with named parameters**; and anyone can run it again with other values, as one undoable batch or as a dry run first. In AVA Modeler these are `get_recording`, `save_steps` and `run_steps`, with parameters written as `${height}` or `${height * 1.2}`.

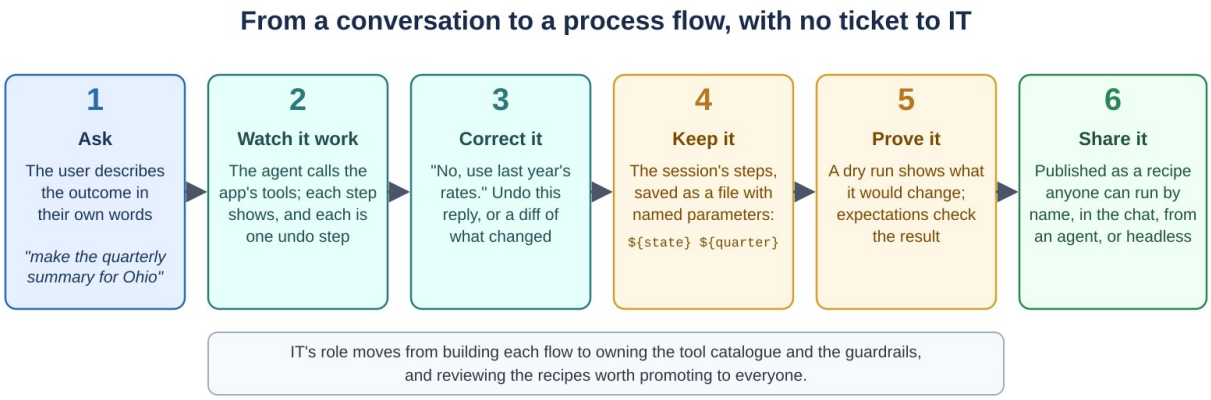


Figure 2. A business user's conversation becomes a reusable, parameterised process flow, proved by a dry run before it is shared.

4.5 Safe to experiment

Business users will only try things if trying is safe. The layer makes it so:

- **Every call is one undo step**, in the same history as the user's own edits, and a call that fails changes nothing.
- **Undo this reply** puts back everything one answer did; a **timeline** lists every answer that changed something, with a picture of how it was before.
- **Checkpoints and diffs** (`checkpoint`, `diff`, `restore_checkpoint`) compare with, or go back to, a named moment.
- **Batches** run several calls as one step, all or nothing; a **dry run** reports what they would change, then takes it back; **expectations** (`expect`) check the result and fail the batch if it's wrong.

4.6 A person in the loop where it matters

- The application **asks the person before anything drastic** (MCP elicitation): "This would remove 288 of the 288 polygons in layer 1. Go ahead?"
- **Locks** keep an agent off what a person is working on; agents **wait** while a dialog is open or the person is mid-drag; a **list of connected agents** shows who is in.

4.7 Governance built in, not bolted on

Control	How it works
Off until allowed	Outside agents can't connect until the user turns it on; the network door needs it turned on as well, and a token.
Narrow access	Files are read and written in one working folder only; nothing outside it, and nothing over an existing file unless asked.
Read-only first	A first deployment can expose only tools that read and explain (X12 Explorer's are all read-only).

Control	How it works
Protected data	De-identify at the source (X12 Explorer's `--deidentify`), or run the model where the data lives.
Keys and tokens	Kept in owner-only files, sent only to their own server, only ever shown masked, never in settings, logs or saved chats.
Cost and history	Every turn's tokens and estimated cost shown; a budget stops the chat; chat history kept only if the user opts in.
Audit	Every change is a named step in the undo history; a chat can be exported as a record of what was asked and done.

4.8 Test the AI layer like the rest of the code

Every tool is tested through a real MCP client: its arguments, its errors, its undo, and that a failure leaves nothing behind. Every recipe is run step by step. The chat is tested against **scripted fake model servers**, so no test ever calls a real API or needs a key. This is what makes it safe to let business users author flows: the platform under them doesn't move.

5. Tools for building it

Everything below is available today; most of it is open source. These are the pieces I used for the examples in section 6.

Layer	Tools	Notes
Protocol	Model Context Protocol; official SDKs for C#/.NET (`ModelContextProtocol`), Python, TypeScript, Java and others	Transports: stdio for a local agent, a local pipe into the open application, Streamable HTTP for the network
Cloud models	Anthropic Claude (Messages API, official SDKs); OpenAI; services with OpenAI-compatible APIs (OpenRouter, Groq, Mistral, Google Gemini)	Frontier models for open-ended work where data rules allow; prompt caching keeps long tool sessions cheap
Local and private models	Ollama, LM Studio, llama.cpp, vLLM: all speak the OpenAI Chat Completions API; open-weight models such as Qwen, Llama and Mistral	Run on a workstation, an on-premises server or a private cloud tenancy; nothing leaves the client's boundary
Agent clients	Claude Code, Claude Desktop, any MCP client; or the application's own built-in chat	One `claude mcp add …` line attaches an agent to the running application
Application side	A command or service layer; an undo history; a scripting sandbox for macros (Lua through MoonSharp in these examples); ASP.NET Core (Kestrel) for the network door	Most business applications already have the first two
Testing	xUnit; headless UI testing (Avalonia.Headless); fake model servers replaying scripted streams; the MCP SDK's own client in tests	Screenshots in this document were taken by the test suites themselves
Development	Claude Code as the engineering agent: reading code, writing tools and tests, running the suites	Every application here was built this way

6. Working examples

Seven applications of mine carry this layer. They span 3D modeling, animation and rendering, electronics simulation, desktop publishing, diagramming, screen design and healthcare EDI: very different domains, the same pattern.

Application	MCP tools	Recipes	Built-in chat	Ways in
AVA Modeler	133	19	Claude or any OpenAI-compatible model; undo reply, timeline, history	Chat; agent in the window; headless; network
AVALightWave	117	19	The same (AVA Modeler's panel)	Chat; agent in the window; headless; network
CirqAvalonia	45	12	The same	Chat; agent in the window; headless; network
AVAPageStream	40	3	The same	Chat; agent in the window; headless; network
AVASvgMaker	32	3	The same	Chat; agent in the window; headless; network
AVAUIDesigner	39	11	The same	Chat; agent in the window; headless; network
X12 Explorer	7, read-only	none	No: deliberately read-only for PHI	Headless agent, with de-identification

6.1 AVA Modeler: a 3D modeler anyone can direct

A polygon modeler with an animation and rendering workspace. Ask its chat to "make a chess pawn" and the model draws half a profile, lathes it, sets a surface, renders a view to check it, and says what it made, each step a line in the panel and an undo step in the application. A second request puts it in a lit scene and renders it.

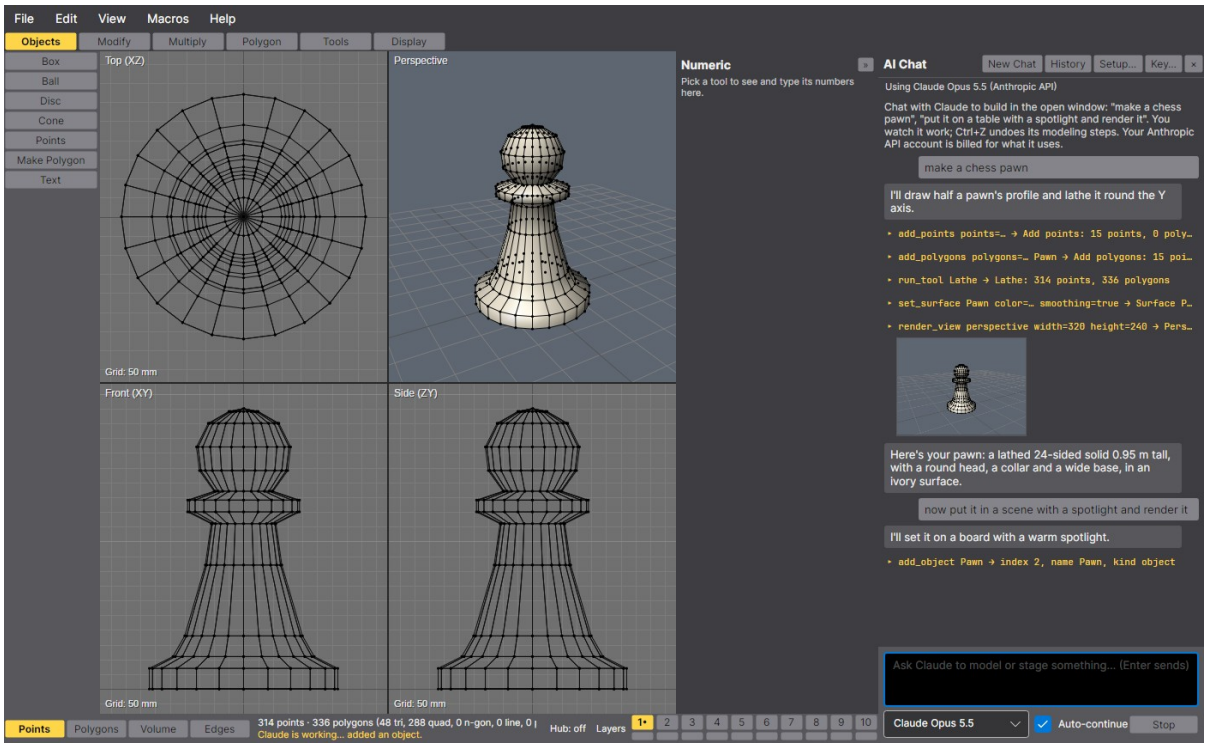


Figure 3. AVA Modeler: "make a chess pawn". Each tool call shows in the chat; the model looks at its own render before answering.

AVA Modeler is where the more advanced techniques were proved first: batches with dry runs, expectations, checkpoints and diffs, stable ids for geometry, locks, background jobs, and **steps files** that turn a session into a parameterised, repeatable procedure. Two of the safety features show here:

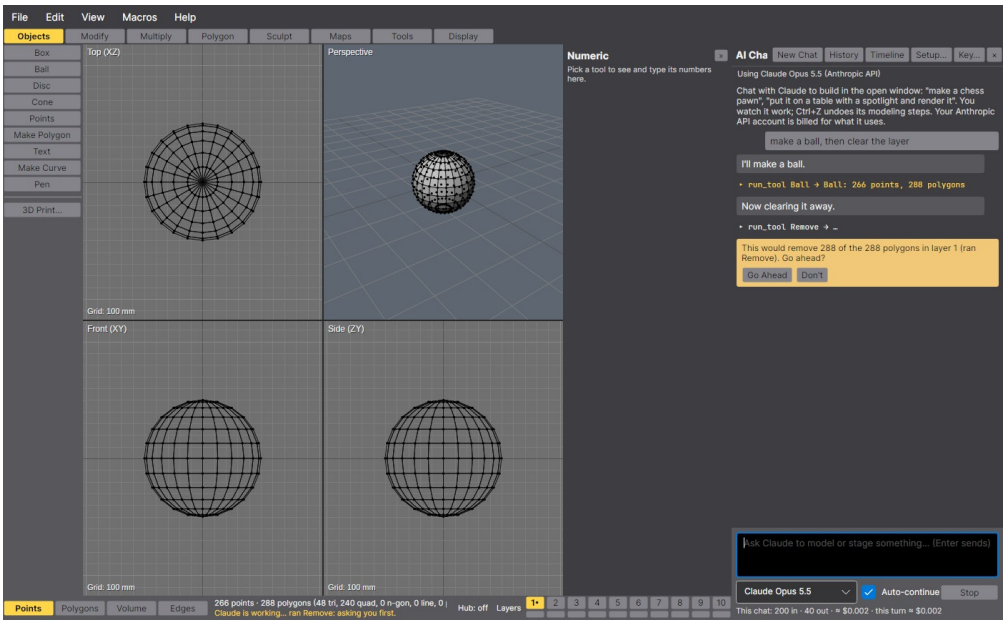


Figure 4. A person in the loop: before removing every polygon in a layer, the application asks.

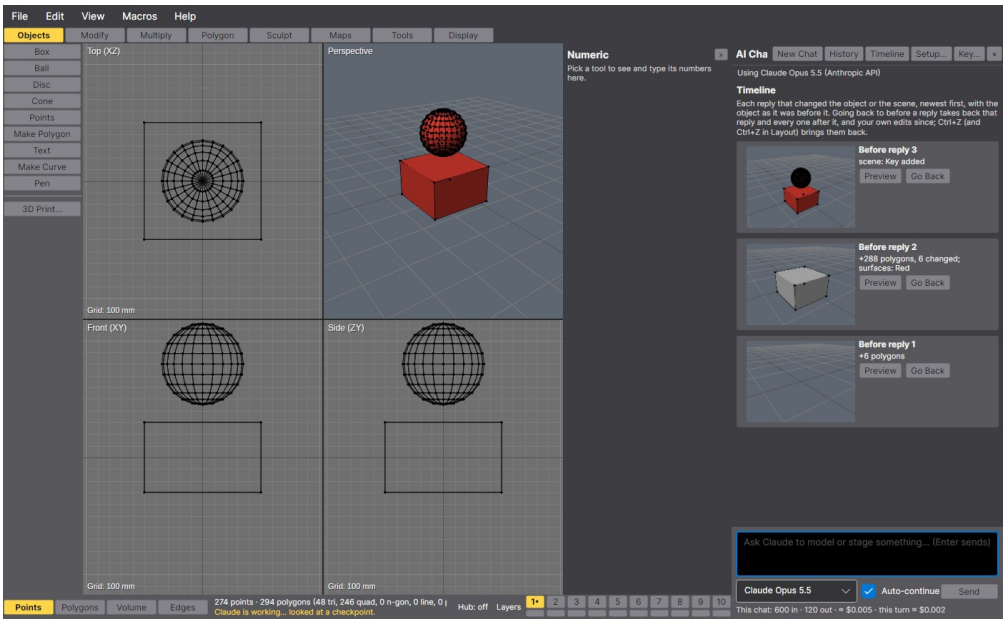


Figure 5. The timeline: every reply that changed something, with a picture of how it was before, and Go Back.

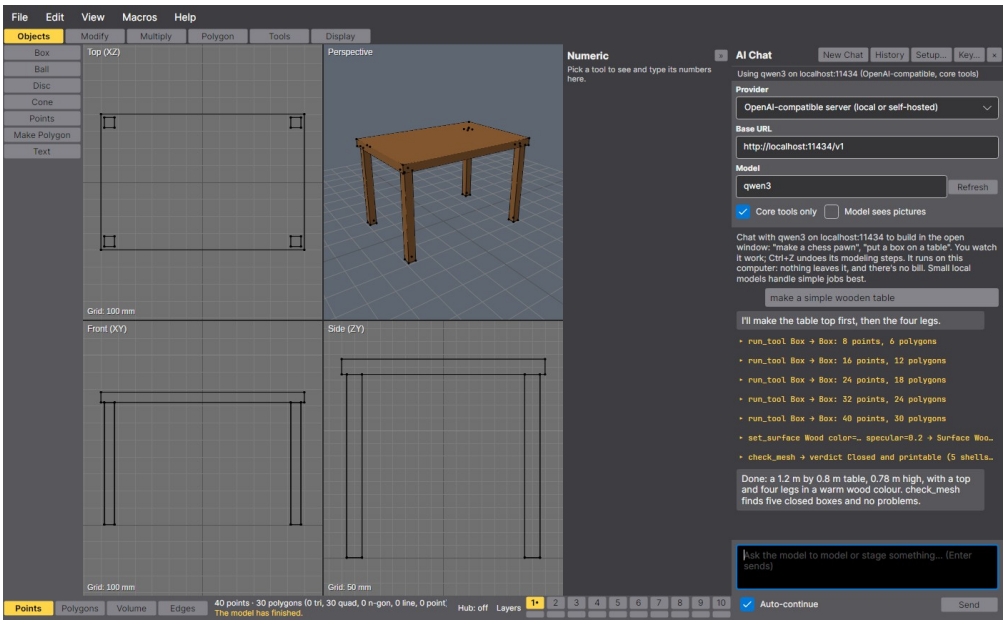


Figure 6. The same chat on a local model (qwen3 through Ollama): nothing leaves the computer, and there is no bill.

6.2 AVALightWave: a whole scene from a description

A scene, animation and render program built on AVA Modeler's engine. Its 117 tools work at a higher level: build a scene from a description, frame the shot, key a camera move in a phrase, apply a whole look in one style, simulate physics, lay out a cut of several shots, and check each shot with a contact sheet before it renders. This is the business-user pattern in another form: **domain-level tools** ("apply a style", "build the shots") let a non-specialist direct work that once needed an expert at every step.

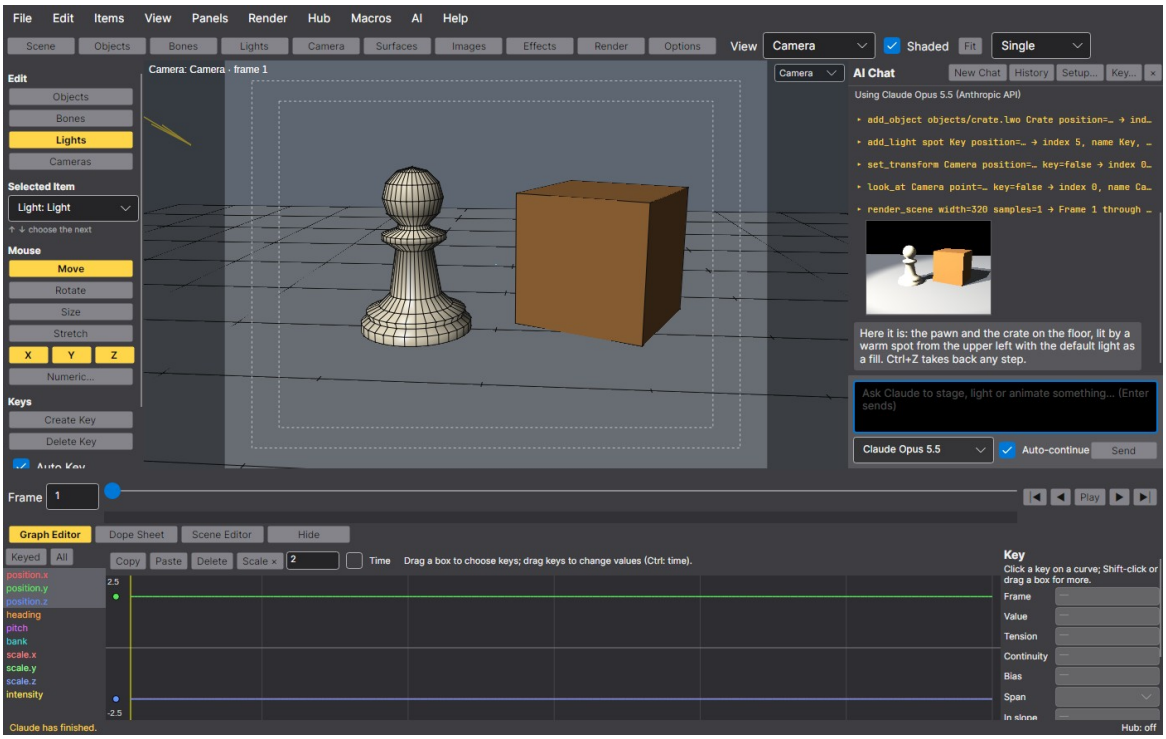


Figure 7. AVALightWave: the chat loads an object, adds and aims a spotlight, positions the camera and renders, then describes the result.

6.3 CirqAvalonia: an AI that checks its own work

An electronic circuit simulator with 428 parts, including working 6502, Z80, 6809, 8085 and 68000 computers. The model places and wires parts, probes them, runs the simulation and **reads back measured results**: asked for a 555 timer at about 1 kHz, it reports 989.5 Hz and a 52.8% duty cycle, measured, not guessed. That is the property business users need from an assistant: it verifies its result with the application's own checks before saying it's done.

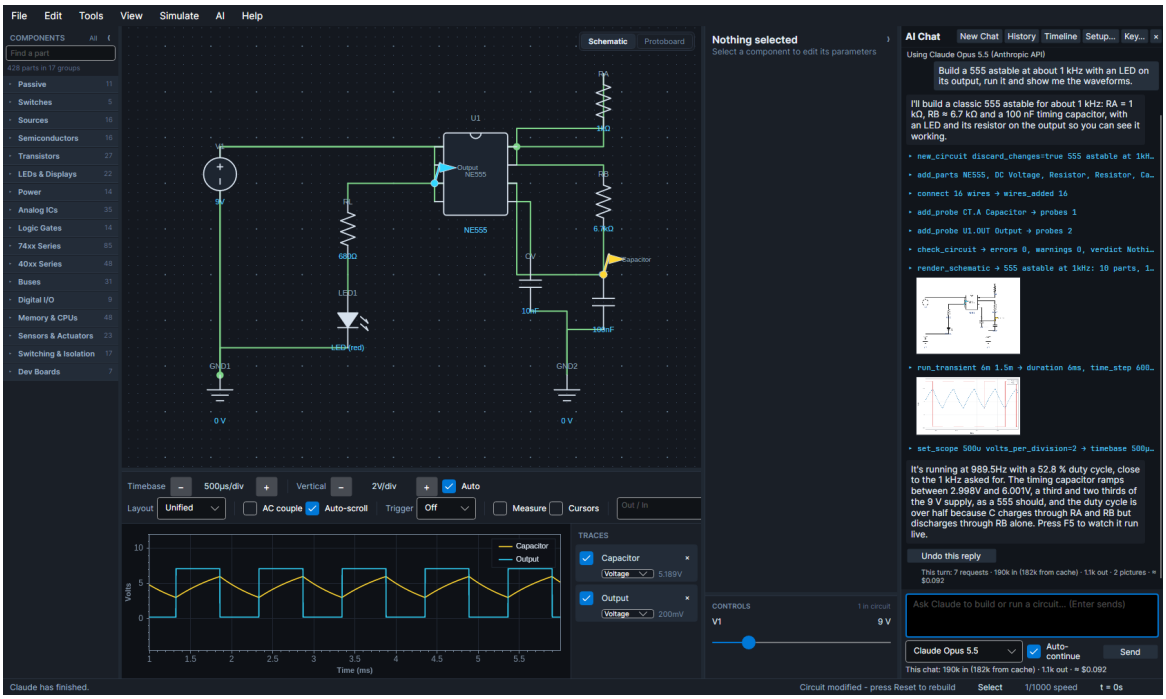


Figure 8. CirqAvalonia builds a 555 astable, runs it, and quotes the measured frequency; the scope shows the run.

CirqAvalonia also shows the chat used for **training rather than doing**: in its fault-finding lessons, *Coach Me* asks the model to guide a learner to a hidden fault by asking what they would measure, never by telling them the answer.

6.4 AVAPageStream: documents produced on request

A desktop publishing program. Its 40 tools lay out pages, flow text through linked frames, apply styles and swatches, place pictures and tables, run a print preflight and export PDF (including press-ready PDF/X-4). For client work this is the closest analogue to a business document pipeline: letters, notices, reports and newsletters assembled from content and data, laid out to a standard, and checked before they go out.

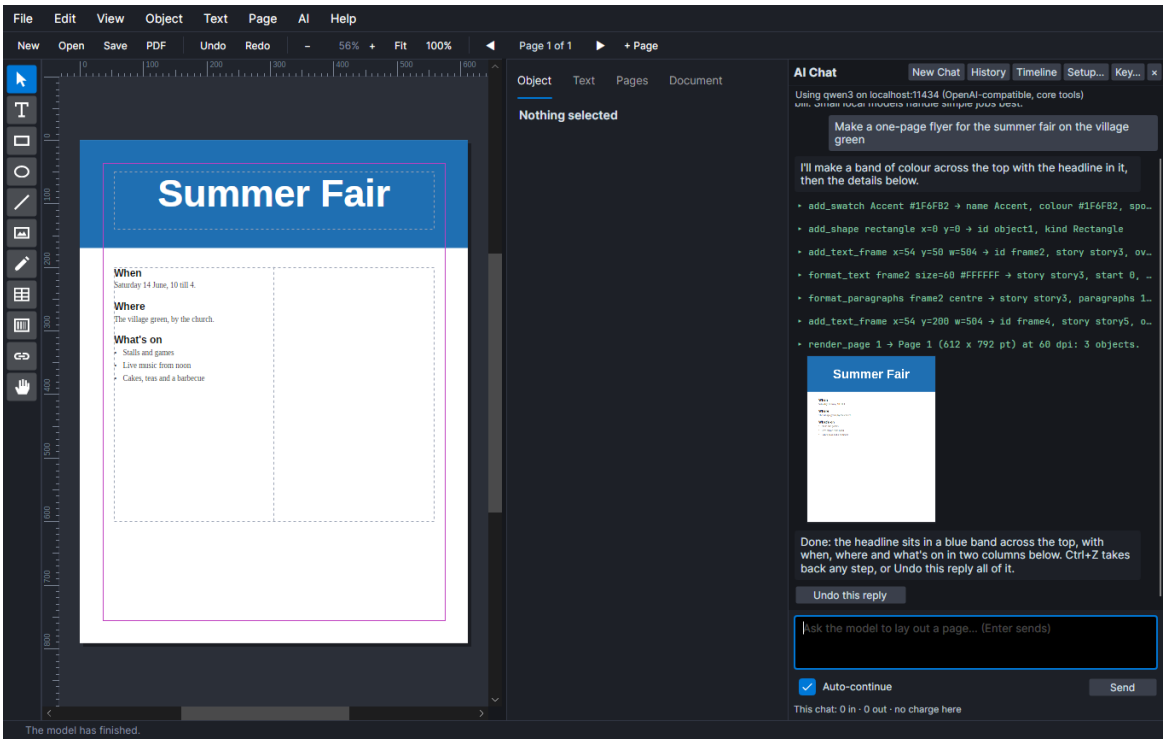


Figure 9. AVAPageStream: asked for a flyer, a local model lays it out, looks at the page, and offers Undo this reply.

6.5 AVASvgMaker: from source code to a flowchart

A Visio-style diagram editor with 167 stencils (flowchart, BPMN, UML, network, electrical), which reads and writes Visio drawings and Mermaid. Its 32 tools add shapes and connectors, lay a page out, and export SVG, PDF, images or .vsdx; two of them, `read_file` and `add_flowchart`, let a person say **"examine this source file and flowchart it for me"**. The model reads the file (only inside the folder the person chose), works out its decisions and steps, and draws the flowchart on a new page in one undoable step, then renders it to check the layout.

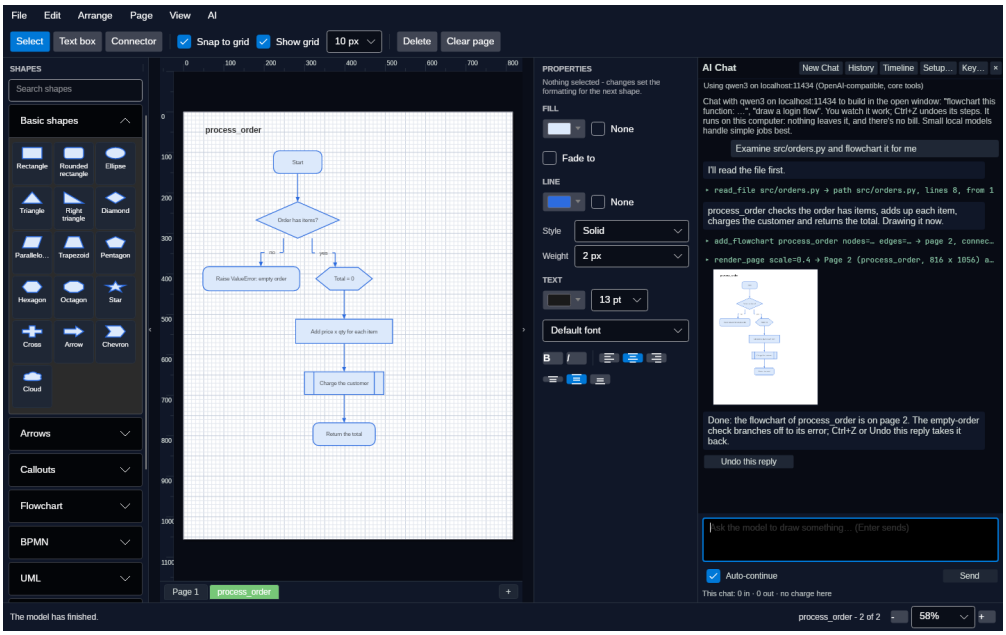


Figure 10. AVASvgMaker: asked to flowchart a source file, a local model reads it, draws its logic on a new page and checks the picture.

This is the examination of section 7 in miniature: **code nobody has documented becomes a diagram a business owner can read and correct.** The same tools draw approval processes and network diagrams from a description.

6.6 AVAUIDesigner: data-entry screens from the database itself

A drag-and-drop designer for Avalonia screens, in the tradition of the Windows Forms designer. Its 39 tools build and check screens, and several of them read **data structures**: C# classes, SQL scripts, or the catalogue of a live SQL Server, PostgreSQL, MySQL, Oracle, SQLite or Db2 database. It reads table definitions, keys and relationships, **never rows**, and agents never see a connection's password. From a table, one call makes the standard **CRUD screen**: a searchable list beside an edit form with the right editor on each field, a view model that checks required fields and lengths before saving, and a repository in that database's own SQL with every value passed as a parameter. Generated repositories were run end to end against each engine.

Its eleven saved recipes are the screens healthcare software is made of: a **patient chart**, intake, vital signs, medications, a SOAP progress note, scheduling, and scored assessments (**PHQ-9** with its item 9 safety alert, GAD-7, a **CANS** core set, the Morse Fall Scale), with any other instrument described by its scale, sections and items.

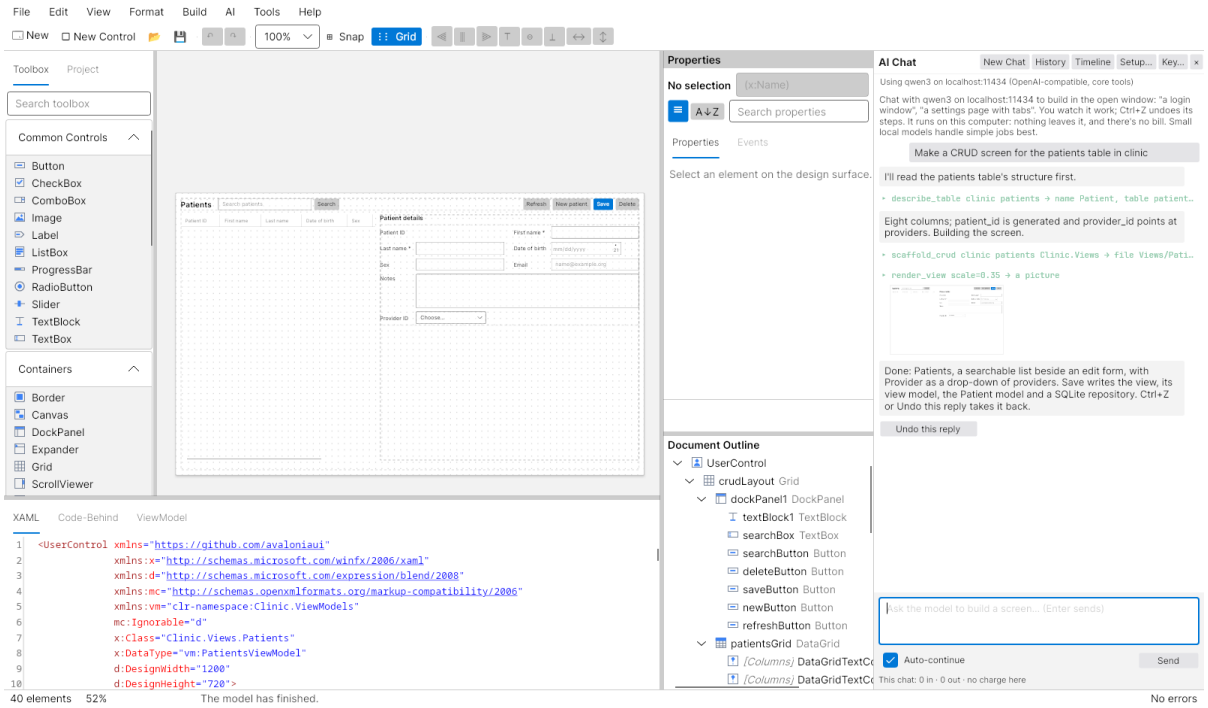


Figure 11. AVAUIDesigner: asked for a CRUD screen for the patients table, a local model reads the table's structure, builds the list and form, and checks the picture.

For re-implementation this closes a loop: **point it at the old system's database and the new screens start from the real tables**, not from someone's memory of them.

6.7 X12 Explorer: the shape of a PHI pilot

A HIPAA X12 EDI viewer for nine transaction types (820, 834, 835, 270, 271, 837, 276, 277, 999). Its MCP server has **seven read-only tools** that balance an 835, find every denied claim and trace a 999 rejection to the element that caused it, reading one folder and writing nothing. With `--deidentify`, every file is de-identified as it is read: names, identifiers, addresses and dates are replaced with consistent stand-ins so the totals still balance, and no tool ever returns PHI. **This is the shape a first pilot on protected data should take.**

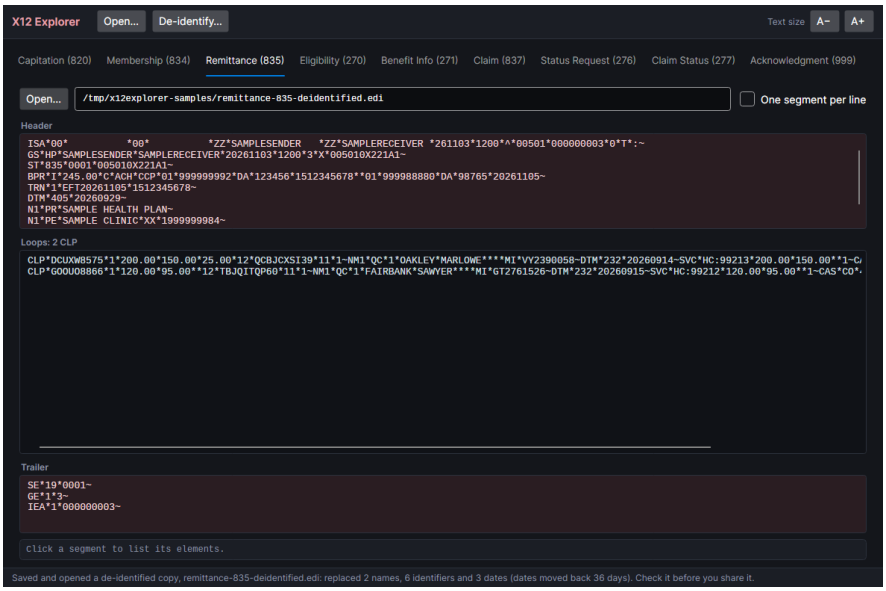


Figure 12. X12 Explorer: a remittance (835) de-identified. Names, identifiers and dates are replaced; the amounts still add up.

6.8 What this looks like in a business application

Business scenario	Technique	Where it's shown here
"Explain why these claims were denied, across the whole batch"	Read-only tools over existing parsers; de-identification; local model	X12 Explorer
"Run the month-end correction for region 4" (a recurring flow)	Recording → steps file with parameters → dry run → shared recipe	AVA Modeler (`save_steps`, `run_steps`)
"Produce this quarter's member letters from the template"	Domain tools for layout; preflight before output	AVAPageStream
"Give us the data-entry screens for these tables"	Read the database catalogue, never rows; generate screen, validation and parameterised SQL	AVAUIDesigner (`describe_table`, `scaffold_crud`)
"Show me how this process actually works today"	Read-only access to a chosen folder; the model draws what it reads	AVASvgMaker (`read_file`, `add_flowchart`)
"Configure the new program's rules the way last year's were"	Checkpoints, diff, Undo this reply; confirmation before bulk change	AVA Modeler, CirqAvalonia
"Set up the review for every site in the district"	Higher-level domain tools (one call does what took many)	AVALightWave (styles, shot lists)
"Did it actually work?"	The model checks results with the application's own measures	CirqAvalonia (measured results)

Business scenario	Technique	Where it's shown here
"Train new staff on the process"	The chat as a coach that asks rather than tells	CirqAvalonia (Coach Me)

7. A case for re-implementation

7.1 When the layer can't be added

The approach above assumes an application with a command layer to put tools over. Many systems in flight don't have one, or can't be changed safely:

- **Closed or aging platforms**: VB6 and early .NET WinForms, PowerBuilder, Access, 4GLs, vendor packages with no extension points.
- **No seams**: business rules inside screen event handlers, so nothing can call them except a person clicking.
- **No tests**: every change risks breaking something nobody remembers depends on it.
- **Lost knowledge**: the people who built it have gone; the documentation, if any, describes an earlier version.

Historically the answer was to leave such systems alone, because re-implementing them was too slow, too expensive and too risky. **AI changes that arithmetic.** The expensive parts of a re-implementation, understanding what the old system really does, writing it down, writing the new code and proving it behaves the same, are exactly the parts AI now does well, under a person's direction.

7.2 Examining a system in flight with AI

A running system can be studied from the outside and the inside without changing it:

Source	What AI does with it	What it produces
Screens and recorded sessions	Reads screenshots and walkthroughs; lists every field, action and validation message	The user-facing behaviour and the workflows, in order
Files, reports, messages	Reads real samples; infers formats, record layouts and rules from many examples	Format specifications, and parsers that read the old data as it is
Database	Reads the schema, constraints, stored procedures and representative data	The data model and the rules enforced in the database
Manuals, help, tickets	Reads them all, and cross-checks them against the observed behaviour	Intent, edge cases and the known bugs to fix or keep
Source code, where available	Reads the legacy code module by module, explains what it does and draws its flowcharts	The exact behaviour, including the parts nobody wrote down
Inputs with outputs	Runs real inputs through the old system and keeps both	Golden masters: the tests that prove the new system matches

The output of this examination is a **written specification** a business owner can read and correct, and a **set of golden-master tests** a machine can run. Those two artefacts are what make the rebuild safe.

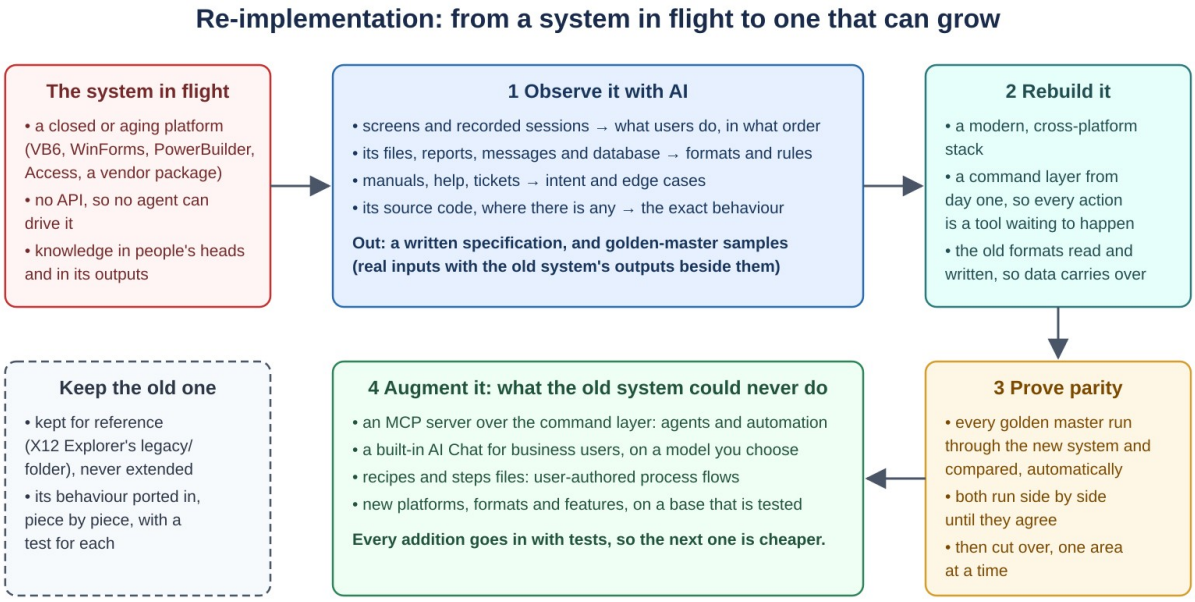


Figure 13. Re-implementation: observe the system in flight, rebuild it with a command layer from day one, prove parity, then augment it.

7.3 Rebuild with the AI layer designed in

A re-implementation should not be a like-for-like copy on a new platform. Built today, it gets the things that make section 4 possible from the first line: a command layer every screen goes through, an undo history, automated tests for every behaviour, and an MCP server over the commands. **The old system's formats are read and written**, so existing data and files carry over unchanged and the two can run side by side.

7.4 Evidence from these projects

Four of the seven applications in section 6 are themselves re-implementations of older software, and each now does things its original never could.

X12 Explorer: an in-house tool rebuilt

X12 Explorer began in 2023 as a Windows Forms program in VB.NET, Windows-only, kept today in its `legacy/` folder for reference and never changed. It was re-implemented in C# on a cross-platform UI: it now reads nine X12 transaction types, runs on Windows, macOS and Linux, de-identifies PHI while keeping the totals balanced, and serves MCP tools to AI agents, **none of which the original did**. The project's own rules say it plainly: the legacy code is for reference; port its behaviour into the new code, with tests.

AVA Modeler and AVALightWave: a 1990s product re-imagined

AVA Modeler and AVALightWave re-imagine LightWave 3D's Modeler and Layout as they were in versions 4 and 5. They were built from the original's documented behaviour and file formats. They **read and write its object (.lwo) and scene (.lws) files**, so work made in the original opens and saves, and they keep its conventions. On that base they have grown what the original never had: an MCP server with 133 and 117 tools, a built-in chat on any model, glTF, STL and 3MF, a slicer for 3D printing, and agents that build whole animated scenes from a description.

AVAPageStream: a re-imagining in ten days

AVAPageStream re-imagines PageStream, the Amiga-era desktop publishing program. Its first commit was on 11 November 2026; its v0.2.0 release on 20 November 2026 has linked text frames, master pages, styles, tables, footnotes, an index, its own PostScript and PDF readers, CMYK and PDF/X-4 export, and the full AI layer: about 35,000 lines of C# with 584 automated tests. The speed is the point: **with AI doing much of the reading and writing, a re-implementation of a substantial application is weeks of directed work, not years.**

Proving parity: golden masters in practice

- **CirqAvalonia's** processor cores are checked locally against independent reference suites (Tom Harte's SingleStepTests, Klaus Dormann's functional tests) that record what the real chips do, cycle by cycle.
- **AVAPageStream's** EPS reader is tested against sample files from real producers, each with the picture its producer drew beside it; its reading of PDF pages is compared with an independent renderer (Poppler), picture by picture.
- **X12 Explorer's** de-identifier is tested so that every replaced file still balances: the same totals, statuses and errors as the original.

This is exactly the method a client re-implementation needs: real inputs, the old system's real outputs, and an automated comparison that must pass before anything is cut over.

7.5 The scale of what was built

Application	Begun	Lines of C# (source)	MCP tools	Re-implements
AVA Modeler	Oct 2026	about 69,000	133	LightWave 3D Modeler 4/5
AVALightWave	Nov 2026	about 14,000 (on AVA Modeler's engine)	117	LightWave 3D Layout 4/5
CirqAvalonia	Sep 2026	about 148,000	45	(new; SPICE-style simulation)
AVAPageStream	Nov 2026	about 35,000	40	PageStream (Amiga DTP)

Application	Begun	Lines of C# (source)	MCP tools	Re-implements
AVASvgMaker	Sep 2026	about 31,000	32	(new; Visio-style, reads Visio files)
AVAUIDesigner	Oct 2026	about 25,000	39	(new; in the style of the WinForms designer)
X12 Explorer	2023, rebuilt Oct-Nov 2026	about 2,900	7	Its own VB.NET WinForms original

7.6 Risks, and how to manage them

Risk	How to manage it
Intellectual property	Re-implement behaviour and documented formats, never copied code or artwork; name the original as an inspiration, not a source. A client re-implementing its own system owns the rights already.
Missed behaviour	Golden masters from real production inputs; run old and new side by side; cut over one area at a time.
Data migration	Read the old formats directly; where the database changes, migrate with the same compare-and-prove approach.
Change for users	Keep the screens and terms users know where they work; add, don't take away, in the first release.
AI and protected data	Choose the model by data class; de-identify where possible; keep the first AI features read-only.
Over-trust in the AI	Every change visible, undoable and checked; a person approves anything drastic; flows proved by dry run before they're shared.

8. A proposed path

8.1 Track A: empower an existing application

1. **Choose** one client application with a command layer, and one repetitive, ticket-driven task its business users own.
2. **Read-only first**: tools that search, summarize and explain, with a chat panel on a model approved for the application's data. Measure time per task and accuracy against a person doing the same work.
3. **Process flows**: add recording and steps files, so business users save and share what works. Measure the change requests that don't get raised.
4. **Changes with approval**: add tools that change data, each one confirmed by a person for anything drastic, undoable, and logged.
5. **Govern**: IT owns the tool catalogue, the guardrails and the review of shared flows; business users own the flows.

8.2 Track B: re-implement a system that can't take the layer

1. **Pick a candidate** using the criteria below.
2. **Examine it in flight**: collect screens, sample files, schema, manuals and code; capture golden masters from real inputs.
3. **Specify**: an AI-drafted, business-reviewed specification, area by area.
4. **Rebuild** one area at a time on a command layer, with tests from the golden masters and the MCP layer from the start.
5. **Run side by side**, cut over when parity holds, then **augment**.

A good re-implementation candidate	Why it matters
No API, and a platform near end of life	The layer can't be added; the system has to change anyway
Frequent change requests from the business	The value of business-authored flows is highest
Observable inputs and outputs	Golden masters can be captured, so parity can be proved
Bounded scope	Can be rebuilt and cut over area by area
Owned by the client	No question about the right to re-implement

8.3 What success looks like

- Change requests for small flows fall; business users run and share flows they authored.
- Time per task falls, with accuracy at least that of a person doing the same work.

- No PHI or PII leaves the client's boundary that their policies don't allow.
- Every flow in use has passed a dry run and is undoable; nothing changed out of sight.
- For Track B: the rebuilt system passes every golden master, and gains its first function the old one could never have had.

Next step

I'd welcome 30 minutes to demonstrate these applications live: the chat building and checking its own work, a recorded session replayed as a flow with new parameters, X12 Explorer de-identifying and explaining a remittance, and to talk through which client application, and which system in flight, would make the best first pilots.

Appendix: the applications

Application	Source
AVA Modeler	github.com/Harlock123/AVAModeler
AVALightWave	github.com/Harlock123/AVALightWave
CirqAvalonia	github.com/Harlock123/CIRQAvalonia
AVAPageStream	github.com/Harlock123/AVAPageStream
AVASvgMaker	github.com/Harlock123/AVASvgMaker
AVAUIDesigner	github.com/Harlock123/AVAUIDesigner
X12 Explorer	github.com/Harlock123/X12_820Explorer
Portfolio	harlock123.com
Companion brief	*Putting AI Agents to Work in Our Clients' Software* (harlock123.com/docs/AI-Agents-in-Client-Software.pdf)