Agentic Coding Against Live Object Models

Author: David Zimmer <dzzie@yahoo.com> (5.9.26)
Site: http://sandsprite.com

A few weeks ago I had a working VB6 project with a few classes, some test data, and a vague idea that I wanted to ask an AI questions about it. Not "generate code that looks like it could query my data" — actually ask questions, get answers from the live, in-memory state of the running program. By the end of the afternoon I had something that worked, ran on either ChatGPT or Claude with the flip of a radio button, and was small enough to fit in my head. This post is about how it works and why I think the pattern is worth knowing if you're a VB6 or COM developer who hasn't yet found a place for AI in your stack.

I built it with Claude as a pair-programmer, working through API trial and error, design choices, and a long string of small bugs and wrong turns. I'll be honest about that throughout. Some of the better design moves came from the AI; some of the worse ones did too. The pattern is mine; the rubber-duck partner was a language model.

What "agentic AI" actually means

The word "agent" has gotten loose. Marketing uses it to mean anything from "a chatbot with a memory" to "an autonomous robot that orders your groceries." I'll use a tighter definition: **an agent is an AI that can take actions, see the results, and decide what to do next** — not just answer a single question with a single response.

The mechanism is plainer than the hype suggests. An agent runs in a loop. Each cycle, the AI emits something — usually a piece of code or a tool call. Some host program executes that thing, captures the result, and feeds the result back to the AI. The AI decides whether it has enough information to answer the user, or whether to take another action. When done, it emits some agreed-upon signal — in my case, the literal string `DONE` on a line by itself — and the loop exits.

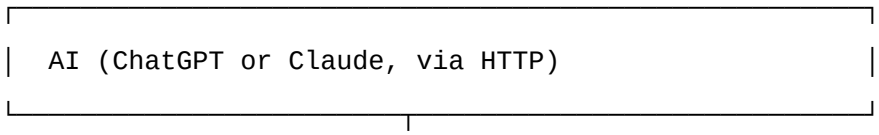
That's it. The intelligence is in the model. The agency comes from the loop. Everything else — tool definitions, sandboxing, retries, context management — is plumbing.

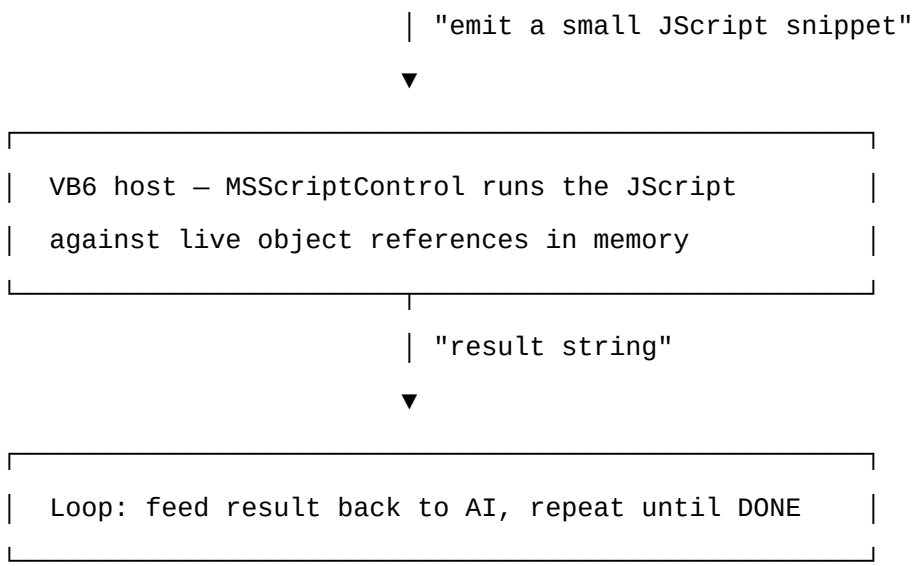
What makes this useful is that **the AI can do things it can't do from training data alone**. It can read a file that didn't exist when it was trained. It can compute things across data it's never seen. It can interrogate the actual state of a running program and reason about what's there.

That last one is what this post is about.

The setup

The AI's view of my program goes through three things:





The AI doesn't see my objects directly. It writes a tiny snippet of JScript like:

```
host.describe(manager)
```

…which the host evaluates against a live `CManager` instance. The result — a description of the class's public surface — comes back as a string the AI can read. The AI then writes the next snippet, and so on, until it has enough information to answer the question.

Three pieces deserve a closer look: the script-evaluation bridge, the introspection contract, and the way I handed the AI a description of my class hierarchy.

The bridge: MSScriptControl

If you've been around COM for any length of time you've probably touched `MSScript.ocx` — the ActiveX scripting control that lets a VB6 host run VBScript or JScript at runtime. You add it as a Component, drop the control on a form, set `Language = "JScript"`, call `AddObject` to expose VB6 objects to the script namespace, and call `Eval` or `ExecuteStatement` to run script code against those objects.

I exposed two things to the script side:

- `manager` — the root domain object the AI is allowed to introspect
- `host` — the form itself, which has a small number of helper methods

That's the entire surface area. The AI can navigate from `manager` to anything reachable through it, and it can call `host.describe(obj)` to learn what an object looks like or `host.answer(text)` to deliver a final answer to the user.

A handful of practical things bit me here, worth flagging:

JScript's idea of VB Collection access is its own. `manager.Users.Count` reads like a property — that's what the typelib says — but from JScript through MSScriptControl it has to be called as a

method: `manager.Users.Count()`. Skip the parens and you get a runtime error so vague it's borderline insulting. For some things you will need to add explicit instructions in the base AI prompt.

Reset the engine between scripts. When the agent loop runs through multiple stages, the JScript engine retains variables from one `Eval` to the next by default. That's convenient until it isn't — once, the AI got the right answer in stage 4 only because a variable from stage 3 happened to still hold the value it needed. That's "accidental correctness," and it would have bitten us the moment the model picked a different variable name. I now call `sc.Reset` between stages, re-set `sc.Language = "JScript"` (which Reset clears), and re-bind the host objects. Each script runs in a fresh engine. Hermetic stages mean each script either works on its own or fails on its own — no shared state to hide bugs behind.

The introspection contract: proto files

The harder design question was how the AI should learn what's in my object model in the first place.

Approach #1 — hand-write a `describeSelf()` method on every class that returns a description string. Easy to start, awful to maintain. Add a property to a class, forget to update describeSelf, the AI sees stale info, makes confident wrong decisions.

Approach #2 — use COM type information. `IDispatch::GetTypeInfo` will hand you an `ITypeInfo` for any IDispatch object, and you can walk it via `TlbInf32.dll`. This works beautifully for ActiveX EXE and ActiveX DLL projects, where every public class gets a registered typelib. It fails completely for Standard EXE projects which is what most of my real code is.

Approach #3 — parse the source code at build time and emit a proto file per class. This is what I went with. The same `.vbp` file the IDE uses to know what's in the project tells my generator what classes to scan. For each class, the generator reads the source, picks out `Public` field declarations, `Sub` and `Function` and `Property` signatures (skipping bodies), and writes a stripped-down text file:

```
Class CProject
  Public name As String
  Public leadName As String
  Public budget As String
End Class
```

The protos live in a `./protos/` subdirectory alongside the .exe. There's a "Regen" button on the form that runs the generator on demand. After every schema change I click it and the AI sees the new shape on the next run.

The reasons this approach wins for me:

- **Works on Standard EXE.** No registered typelibs, no ActiveX dance, no COM registration. Most real VB6 code lives in this shape.
- **The source code is the source of truth.** No hand-written description to drift. If I add a property, the proto regenerates from the class file.

- **The proto file is human-readable.** I can open it in Notepad and read it. So can the AI. So can whoever inherits this project from me.
- **Diffable.** When something breaks, I can `git diff` the protos to see what changed in the schema.
- **No runtime reflection dependencies.** No `TlbInf32.dll` to redistribute, no registration required for the user.

The trade-off: I have to parse VB6 source. That sounds harder than it is. A line-based scanner handles 95% of real code; the remaining 5% (multi-line continuations, oddly formatted properties, the visual designer block in `.frm` files) is annoying but tractable. The whole parser is about 250 lines.

What the AI actually does

A run looks like this. The user types a question into a textbox and clicks Agentic Test. Say the question is:

> Who leads the project with the highest budget, and what is their job?

What happens next:

Stage 1. The AI gets the question plus a system prompt that tells it the rules of the environment — how to call describe, how to deliver an answer, the JScript gotchas, when to emit DONE. The AI doesn't know my object model, so its first move is:

```
host.describe(manager)
```

…which returns:

```
Class CManager
  Public Users As New Collection
  Public Projects As New Collection
  Public Sub addUser(name, age, job)
  Public Sub addProject(name, leadName, budget)
End Class
```

Stage 2. The AI now knows there are two collections — Users and Projects. It introspects a sample project:

```
host.describe(manager.Projects.Item(1))
```

…which returns the `CProject` schema with `name`, `leadName`, `budget`.

Stage 3 (sometimes). Some models also introspect a User at this point to confirm field casing before writing the lookup. Others skip ahead. Both behaviors are reasonable.

Stage 3 or 4. The AI writes the actual solver — a loop over Projects finding the highest budget, then a loop over Users finding the one whose name matches the project's leadName, then a call to `host.answer` with the result.

Final stage. The AI emits `DONE` and the loop exits.

Total wall time: 6–15 seconds depending on backend and how much the model wants to introspect. The output appears in a textbox on the form: *"The project with the highest budget is Sentinel, led by Carol whose job is Threat Intelligence Lead."*

Worth noting what *didn't* happen here: the AI was never told that Users and Projects exist. It was never told there's a connection between `Project.leadName` and `User.name`. It discovered all of that by calling describe and reading the output. If I add a new class tomorrow — say, `CCustomer` with its own collection on the manager — and regenerate the protos, the AI can find it and reason about it without any prompt changes.

This is the part that feels powerful in a way that gets buried by the hype. The AI isn't running a hardcoded query. It's looking at my object model and reasoning about what it finds. The same code path will answer questions about any VB6 domain I drop into the framework.

When the AI gets it wrong

The happy-path walkthrough above is misleadingly tidy. In practice, the AI's first script is often broken in some small way, and the system has to handle that gracefully or the whole run dies on stage three.

The mechanism is simple and worth describing exactly, because it's the part of the agent loop that does the actual work of agency. After the host evaluates a script, it ends up in one of three states:

- **The script succeeded.** Capture the eval result as a string. Build the next user message: *"Result of your last script: [the result]. If the task is fully answered, return DONE. Otherwise emit the next JScript."*
- **The script failed.** Capture the error — both VB6's `Err.Description` and MSScriptControl's `sc.Error.Line` and `sc.Error.Text`, since one or the other is usually populated and never both. Build a different user message: *"Your previous JScript produced an error. ERROR: [error text]. SCRIPT THAT FAILED: [the script verbatim]. Return corrected JScript. Do not repeat the same script."*
- **The AI signalled completion.** The script was literally the string `DONE` — the loop exits.

That second case is the interesting one. The error path is not just "report the error" — it's "show the model the exact text of the broken script alongside the exact text of the error." The *"do not repeat the same script"* clause matters more than it sounds; without it, models will sometimes resubmit the identical broken script as if hoping for a different result. With it, they understand they have to change something.

This handoff is what turns a fragile one-shot call into a self-correcting loop. The AI can probe. It can write a tentative script, see if it works, and refine. It can split a complex task across several stages, using each result to inform the next. When introspection reveals a field with an unexpected type, the next script accounts for that. When a Collection access throws an empty error, the AI can guess at the cause and try a variation in the next stage.

A few details that matter in practice:

- **Both the script and the error get included verbatim.** Truncation or summarization would defeat the purpose. The model is good at reading its own emitted code and spotting the bug — but only if it sees the exact text it sent, not your paraphrase of it.
- **The error from MSScriptControl is sometimes very vague.** The infamous `Line 1:` with no detail. When the AI sees this, it doesn't have much to go on, but it can at least narrow the problem to the first line of its script and try alternative syntax. The prompt has explicit guidance about this exact pattern: *"if you see an empty Line 1 error, the most likely cause is a missing parenthesis on a Collection method."* That came from observing real failure modes — the AI thrashing through three different variations of the same broken Count access before something stuck. The prompt now short-circuits that thrashing.
- **Failed stages count against the cap.** If MAX_STAGES is twenty and the AI burns the first seven on bind errors, it only has thirteen left to do useful work. In practice, a healthy run produces zero or one errors. More than three suggests something structurally wrong — either the prompt is missing a rule the model needs, or the task is malformed, or the AI is genuinely stuck.
- **The fresh-engine reset between stages applies here too.** When a script fails, the engine resets before the AI's next script runs, so any partial state from the failed script — half-assigned variables, broken closures — can't leak through and confuse the retry. Hermetic stages mean a failure is a clean failure.

The end result is that the agent loop has a built-in feedback channel that mirrors how a human would actually debug. Try something. See what happened. Adjust. Try again. The AI doesn't get any special diagnostic powers from this — it gets the same information a developer at a keyboard would get. What makes it work is that it gets that information in a format it can read and reason about, every stage, automatically.

The two backends

I built this against the OpenAI API first, then added support for the Anthropic API as a second backend. They behave the same from the caller's perspective — same method names, same return values, swap them via a radio button — but they're shaped quite differently underneath.

OpenAI's `/v1/responses` endpoint has conversation chaining built in. Each response comes back with an ID. To continue the conversation, you send the new user message along with the previous response ID, and the OpenAI side reconstructs the context for you. Stateful from the caller's perspective.

Anthropic's `/v1/messages` endpoint is stateless. Every request must include the entire conversation history as an array of `{role, content}` pairs. The server doesn't remember anything between calls. So for Claude, my class maintains an in-memory history collection, appends each user message and assistant response as the conversation progresses, and resends the whole array on every turn.

Same external behavior, very different internals. The class abstracts this away — `agentAI.CreateResponse(message, systemPrompt, maintainContext)` works

identically against either backend. The polymorphism comes from VB6's late binding: I declare `agentAI As Object` and the radio button picks which concrete class fills it.

There are limits to this kind of "two AIs, one interface" pattern. The models behave differently in practice — ChatGPT tends to write more thorough introspection code; Claude tends to commit to a solution faster. For the simple test problem they both arrive at the right answer in roughly the same number of stages, but the path each takes is recognizably different. That's interesting on its own.

What goes in the prompt

The system prompt — the instructions the AI sees at the start of every run — is where the framework meets the domain. The framework gives the AI tools and information; the prompt tells it how to use them.

My prompt has grown organically over the project. Every section earned its place because a real bug exposed a real need:

- A section on **JScript binding quirks** (the `.Count()` parens issue, default-member-binding flakiness on collections).
- A section on **hermetic stages** — telling the AI not to rely on variables from previous scripts.
- A section on **string-vs-number comparison**, because the AI cheerfully wrote `if (project.budget > maxBudget)` against budgets stored as Strings, and JScript's `>` on strings is lexicographic. `"85000"` beats `"420000"` in lexicographic comparison, so Hydra "won" the highest-budget contest. The prompt now reminds the AI to `parseFloat` numeric-looking string fields before comparing.
- A section on **sanity-checking the answer** before declaring done. If the answer would surprise you, recheck the script. This one was added after Claude confidently produced the wrong answer in three quick stages. The model wasn't dumb; it was uncautious.

What's *not* in the prompt: my class names, my collection names, what fields they have, what relationships exist between them. That's all discoverable through the introspection contract. The prompt is generic to the framework; the data is specific to the run.

I think of this as a clean split:

- **Framework layer** — the agent loop, the script evaluator, the proto generator, the API clients. Domain-agnostic. Reusable.
- **Prompt layer** — the rules of the environment and any style guidance. Tuned over time as failure modes surface, but still domain-agnostic.
- **Schema layer** — the proto files. Auto-generated from your code. Always current.
- **Data layer** — whatever's in your live objects at the moment the AI asks.

When I drop this framework into a different VB6 project, only the schema layer changes automatically (because it's generated). The framework and the prompt stay the same. The data is whatever the user's program happens to hold.

Async and the UI

A practical note for VB6 developers used to synchronous COM calls: when you put a multi-second HTTP request inside an agent loop, the UI freezes hard. The first version I wrote did exactly that — five stages, three seconds each, the form unresponsive for fifteen seconds at a stretch. Cancellable? Forget it.

`MSXML2.ServerXMLHTTP.6.0` supports asynchronous requests. Open the request with `True` as the third argument, call `send`, then poll `readyState` in a loop with `DoEvents` and a short `Sleep`. When state hits 4, the response is ready. While polling, the UI processes messages — meaning a Cancel button can actually do something.

I made async an opt-in. The default behavior is synchronous, because most callers (non-UI scripts, batch jobs) don't want a `DoEvents` pump running inside their HTTP call — that's a subtle source of re-entrancy bugs. The UI path passes a flag to enable async; everyone else gets the simple blocking call.

A Cancel button on the form sets a flag; the polling loop checks it; if set, it calls `http.abort()` and returns immediately. A re-entry guard on the agent-button handler prevents a second click from kicking off a parallel run during the `DoEvents` pump. Belt and braces, and worth it.

Logging is the framework

If there's one part of this project I'd warn you not to skip, it's the logging. Not because the logs are pretty — they're not — but because logging is what made every other part of this project tractable.

Here's the thing about agentic AI that most posts gloss over: **you cannot debug an agent loop without a complete trace.** When the AI does something wrong, you have to know exactly what it saw, exactly what it emitted, and exactly what the host did with that emission. Without that, you're guessing. With it, you have a transcript you can read top to bottom and find the moment the wheels came off.

My logger is small. It opens a file at the start of every run, prints a timestamped section header, then captures, in order:

- The system prompt sent to the AI (the whole thing, every run — yes it's big, who cares, disk is cheap).
- The user message at the start of each stage.
- The HTTP status code and any error from the AI call.
- The exact JScript the AI emitted.
- The result of evaluating that JScript, or the error if it failed.
- Every call to `host.answer()` — what the AI tried to surface to the user.
- The final outcome (DONE, max stages reached, cancelled, HTTP error) with elapsed seconds.

That's it. Nothing fancy. Append-only text file, flush after each stage so I can tail it while a run is in progress.

The payoff is enormous. Two examples from this project:

In the early days, the AI kept producing the same broken script over and over. Without the log, I'd have been speculating. With the log, I could see the exact failure pattern: every script's first line was the same Collection access, and every error message was the same vague `Line 1:` with no detail. That narrowed the search instantly. Was it the line itself, or something that happened before the script even ran? Once I confirmed the script's other lines never executed, I knew it was a parse-or-bind error on line one. From there, manually pasting that exact line into the host's manual eval box told me `.Count` needed parens. The log made the wrong-step visible; the fix was minutes after that.

Another time, the AI got the right answer in a four-stage run — but the log showed it had referenced a variable from a previous stage that, by my framework's design, shouldn't have persisted. I'd thought I was resetting the engine between stages, but I wasn't. The "correct" answer was correct by accident — the script worked because a leftover variable happened to hold the right value. Without the log, I'd have called it a success and shipped. With the log, I caught the bug, added `sc.Reset` between stages, and discovered that some of my "passing" tests had been passing for the wrong reason.

Logging for AI-assisted development

Here is the technique that mattered most over the course of this build, and that I want to spell out for anyone working with an AI pair programmer:

The log is the artifact you share back with the AI.

When something broke during this project, I didn't try to describe the problem in English. I pasted the entire log file into the chat and asked the AI to read it. Every single time, the AI's analysis was sharper from reading the log than it would have been from any description I could have written. Why? Because the log contains things I wouldn't have noticed to mention. Timing, the exact text of an error, the precise sequence of attempts, the model's own reasoning visible in the scripts it emitted.

When you're working with an AI assistant on a real system, your job is partly to be the eyes and hands the model doesn't have. The log is the cleanest way to do that. It's the same trick that makes a good bug report effective: **don't describe the bug, show the bug.**

A few practical notes that came out of using this pattern:

- **Log before you think you need to.** The first hour I tried to debug from the IDE output window and my own memory. It went badly. Adding the logger was a fifteen-minute fix that paid for itself within an hour.
- **Flush after each stage.** Buffered writes mean a crashed run loses the last few minutes — exactly when you need them most. Close-and-reopen-in-append between stages is ugly but reliable.
- **Timestamps on every entry.** Stage timing tells you whether the AI was thinking or whether the API was slow — often a useful distinction.
- **Include the full system prompt at the top of every run.** Yes, it's repetitive across runs in the same file. But when you're debugging "why did the model do X?", the system prompt in effect at that moment is part of the evidence. Logs that omit it are missing half the picture.

- **Include the full inputs and outputs verbatim.** No truncation, no summarization. The whole point is to capture what actually happened, including the parts you didn't think were interesting. The parts you didn't think were interesting are exactly the parts that turn out to matter.

The header at the start of this section was a slight exaggeration — logging isn't *literally* the framework. But it's close. The framework gives the AI access to the world; the log gives me access to what the AI did with that access. Without that second feedback loop, building anything non-trivial with a language model is shouting into a void and hoping it answers.

What this isn't

It's not a chatbot. The agent runs to completion or to a stage cap; it's not for free-form conversation.

It's not a code generator. The AI doesn't write functions for me to paste in. It introspects and reasons about live data and reports back.

It's not an AI replacement for SQL or for a real database query layer. If you have a database, query it directly. This is for the case where the interesting state lives in your program's running object graph and putting a query layer in front of it isn't worth the effort.

It's not magic. The whole thing is a few hundred lines of VB6 plus a system prompt. There's no fine-tuning, no embeddings, no vector database, no agents-frameworks-as-a-service. It's a loop and a few tool functions.

Where this could go

A non-exhaustive list of things this pattern naturally extends to:

- **Triage tools.** Point the framework at an internal CRM, ticketing, or workflow object model. Ask "what's stuck?" and let the AI traverse to find out.
- **Data integrity audits.** "Are there any orphan records?" "Which entities are missing required fields?" The AI can navigate relationships and report.
- **Ad-hoc reporting.** "Give me a breakdown of X by Y for the last 30 days." The AI writes the query in JScript against live objects; you get an answer in seconds without standing up a reporting tool.
- **Legacy code archaeology.** Old systems whose only API is their in-memory state. The AI can read the proto, navigate the graph, and surface what's actually in there.
- **Anomaly surfacing.** Open-ended analytical questions — "what looks odd in this allocation?" — where the AI's value isn't computing a specific number but noticing patterns a human hasn't thought to look for.

The pattern generalizes beyond VB6 too. Any host language that can:

- Run a small embedded script engine,
- Expose live object references to that script,
- Make HTTP calls to an AI API,

…can do this. C++ with ChakraCore, C# with ScriptControl COM interop, Delphi with the same MSScript control, even something like Python with a similar setup. The mechanics carry over.

Closing notes

I'll publish the code under the project name **ai4vb**. It's not big. The proto generator is ~250 lines; the AI client classes are ~200 lines each; the agent loop in Form1 is another ~150 lines. The whole thing fits in a single afternoon's reading. If you're a VB6 developer who's been watching the AI conversation from the outside and wondering whether there's a useful entry point for the kind of code you write — there is. It's smaller and more practical than the marketing makes it sound.

A few things I'd do differently if I were starting from scratch:

- **Treat the prompt as a versioned file from the start.** The prompt evolves significantly as bugs surface, and being able to diff versions of it (or A/B test prompt changes) would have saved time.
- **Log everything from the first stage.** I added structured logging a few hours in, and I wish I'd had it from minute one — the failure traces are gold for understanding what the model is actually doing.
- **Don't be afraid to ask the AI to introspect more.** Some of my prompt evolution was about pushing the model toward "look at the data first, opinion second" — and once it has the data, it's reasonably good at reasoning about it.

One final thought: the most important property of this design isn't the technology. It's that the AI's reasoning is fully visible. Every prompt, every script it emits, every result it sees, every error it recovers from — all logged. When the AI does something I don't expect, I can read the log and understand exactly why. There's no opaque black box. There's a transcript I can grep.

That feels like the right way to ship AI inside a serious system. Make every choice the model makes auditable. Make every tool it calls explicit. Make every piece of context it sees a readable file.

That's the bar. The rest is plumbing.

Built with Claude as a pair programmer; tested against both Claude and ChatGPT as the agent backend. The code is at [link forthcoming].