2026-01-19

It's early 2026. Industry practice is divided on how to structure tool descriptions within the context window of an LLM. One strategy is to provide top-level tools that perform fine grained actions (e.g. list pull requests in a GitHub repo). Another increasingly popular strategy is to eschew new tools per se and to simply inform the model of useful shell commands it may invoke. In both cases reusable skills can be defined that give the model tips on how to perform useful work with the tools; the main difference is whether the model emits a direct tool call or instead an `exec_bash` call containing a reference to CLIs.

To me it is clear that the latter represents an innovation on the former. The best feature of the unix shell is command composition. Enabling the model to form pipelines of tool calls without re-prompting the model after each stage should present huge savings in token cost. The resulting pipelines can also be saved to scripts or be customized and interactively executed by human operators.

The command line is an *interface* compatible with humans and machines. If the model is adept at using it (it's already text), why fall back to a machine-native *protocol*?

One good response is that MCP is an easy way to expose SaaS functionality to agents. In lieu of MCP, how can we achieve that? I'll answer this question by providing two quite different examples from my recent work: giving an agent access to Google Docs and to Google Groups.

HTTP APIs

I wanted my agent to be able to list my cloud-based Google Docs, to read them as markdown, and to read and understand any attached comment threads.

Google provides a very nice API to fulfill all of this functionality (well, comments are harder). I did the obvious thing and spun up a Google Cloud project, pasted the API documentation into an LLM, and the result was a `gdrive` CLI with subcommands to list files and to export a particular one.

That worked. But as in the title of this post, *the best code is no code*. This script seemed entirely like boilerplate which *shouldn't have to exist*. This would be true even if the script were to use an SDK rather than make HTTP calls directly. In reality, Google—and many SaaS vendors—*already define* a program which can be used to call all of their APIs. *It's their OpenAPI spec!* The program just needs a sufficient interpreter.

I Googled around and was thrilled to discover Restish, a tool which nearly perfectly matches my philosophy. If OpenAPI specs are programs, Restish is their interpreter. Sample usage (cribbed from their docs):

```
# Register a new API called `cool-api`
$ restish api configure cool-api https://api.rest.sh/openapi.yaml

# This will show you all available commands from the API description.
$ restish cool-api --help
```

```
# Call an API operation (`list-images`)
$ restish -r cool-api -H 'Accept: application/json' list-images | jq '.[0].name'
"Dragonfly macro"
```

Restish even generates shell completions for the API endpoints (subcomands) and parameters (options/args)!

I have only two complaints:

- Restish wants to handle API authorization for me (persisting e.g. OAuth tokens). I want it to just be an "interpreter for OpenAPI programs". I'll manage my own auth flows and inject my own tokens.
- Executing commands against an api spec requires registering the spec with Restish ahead of time. See above—I want just an interpreter.

Both points imply that I'll want a wrapper script around restish. The wrapper script will manage the second issue (it will create a temporary spec directory to satisfy Restish). The script will also perform my desired authorization flow and inject tokens into Restish ephemerally.

API Authorization

Looking back at the omnibus script that I generated initially, it contained an OAuth 2.0 client to hit Google's authorization flow, get tokens, and refresh them upon expiry. OAuth 2.0 is a standard. A particular set of parameters (Google's OAuth URL, client id, client secret, grant type, scopes) could be thought of as *a valid program in the OAuth 2.0 client language*.

I, again, just needed an interpreter. I, again, found one.

oauth2c is a command-line client for OAuth 2.0-compliant authorization servers. You input the aforementioned program (i.e. URL, grant type, ...) and it begins the ensuing flow (usually by opening your browser) then prints the resulting tokens to stdout.

With this missing piece, what was previously a couple-hundred lines of dense Python is now an order-of-magnitude smaller shell script which performs the logical equivalent of `oauth2c "https://accounts.google.com/..." | restish google drive-files-list`.

I've published the resulting script to bmwalters/gdrive-client. The repo also contains a cool method for propagating shell completions.

Detour: secure token storage for macOS CLI scripts

While I'm dispensing pro-tips, I should highlight this pretty cool and under-documented way to securely store data (like a long-lived refresh token) from a macOS shell script.

Let me introduce the problem. The results of Google's OAuth flow are a short-lived access token (to hit APIs; valid for about an hour) and a long-lived refresh token (to mint new access tokens; valid for **6 months**). I wasn't comfortable with leaving that refresh token exposed on my machine. Services like the AWS CLI do indeed store plaintext credentials in ~, but those tend to expire much more frequently than 6 months.

I knew I wanted to reach for the macOS Keychain, and in particular some security level that would require biometrics / passcode when reading the refresh token.

macOS ships a handy keychain CLI named `security`. You can store secrets in Keychain with invocations like `security add-generic-password -s google-api -a my-account -w $REFRESH_TOKEN`. But biometrics are not trivially supported, and web search advised me to create a small Swift wrapper. After doing so, I learned that any `kSecAttrAccessControl` attribute that would lead to biometrics or device passcode would result in the binary requiring real signed entitlements through the Apple Developer Program. I was a bit stuck looking for a solution to what seemed to be a simple requirement.

I ran the `security` man page through Claude Opus 4.5 and the model made a very interesting discovery.

```
-T appPath      Specify an application which may access this item (multiple -T options are
```

It turns out that the keychain remembers which application stored the password—by default this is probably `security` itself or perhaps my shell; I haven't checked—and that application is permitted to read back the password without user-interactive authorization. Providing the `-T` flag to `security` when creating the password allows overriding said program entry, and crucially the *empty string may be used* to remove the default application entry.

In other words this code:

```
security add-generic-password -T"" ...
```

will prevent `security find-generic-password` from simply returning the secret, even when invoked immediately after secret creation. In practice, attempts to read the secret will prompt me for my device passcode, which is definitely good enough for my use case.

Putting it all together, I had a CLI that, when invoked, would try to use the stored access token with Restish (no passcode prompt needed). If the access token was invalid, it would invoke `oauth2c` to refresh the token and retry. This would prompt me for my devcie passcode. If that also failed, it would invoke the Authorization Code flow using `oauth2c` which would seamlessly open my browser and retry the command on success.

All with only shell pipelines, no bespoke code. Vastly reduced surface area for future maintenance and for bugs to hide in.

Adversarial interoperability

That's all-well-and-good for services which provide machine-readable API specs, but what about those which are less charitable?

Google Groups is one such case. I wanted to export the discussion history from pollenpub to serve as a Q&A knowledge base while developing this blog site. However my research turned up no such API from Google.

I love using LLMs to solve this class of problem. My workflow is as follows:

1. Open a fresh private browser (to capture any authorization flow, if needed).

2. Open Devtools > Network and filter to HTML, XHR, WS, Other.
3. Perform the actions that I would like to automate, i.e. load the Google Group site, navigate to the next page, and read a particular conversation.
4. Firefox Devtools > Network > right click > “Save All As HAR”.
5. Run the file through cloudflare/har-sanitizer
6. Prompt an LLM with: “in this directory there is a large HAR file captured while I did actions xyz; please create a Python client for this API”.
7. Edit the generated file to add a meaningful `User-Agent` string with a backlink.

I’ve repeated this workflow about three times and I have near-term plans for a couple more.

Note that I haven’t tried *combining* the above two workflows yet: I haven’t asked the model to produce an OpenAPI spec + reverse-engineered OAuth parameters, but that’s a logical next step.

Conclusion

There’s a lot of power in composing CLIs. You get human interaction and current-generation-LLM interaction for the price of one. And with some creativity, it’s often possible for one individual to maintain CLIs in place of an MCP server that has been developed for a given service, or even to do so before the comparable MCP server has been written.
