



*This article is part of “Exploring Gen AI”.
A series capturing Thoughtworks
technologists' explorations of using
gen ai technology for software
development.*

30 July 2026

The Economic Benefit of Refactoring

Giles Edwards-Alexander



Giles is CTO for Europe, Middle East and India at Thoughtworks. He has over 25 years experience in engineering and technology leadership across technologies from mobile to AI and industries including retail, fintech and healthcare.

As part of getting to grips with the new world of agentic engineering, I built an application to support my work. It's a sophisticated app: high-quality web UI with dynamic refresh and look-up, modals and auto-save, integrations to external systems, machine learning and text analysis, background jobs, and a proper environment setup with fully automated deployment. It's approximately 150,000 lines of code, primarily in Rust (~120 kLoC) with the remainder in TypeScript and Terraform.

This was entirely written by agents. Mostly Claude Code, and some use of Cursor. I didn't read or review any of the code, except occasionally, out of interest.

While building the application, I could see some things going awry. After watching an edit to line 4,000 of a file scroll by in the terminal, I had a closer look. The data access layer had grown to over 6,000 lines. As more features landed, this continued to grow. Every query, read or write, repeated the same HTTP request setup, the same JSON encoding and decoding. Eventually, it reached 17,155 lines. In a single Rust file.

An experiment in refactoring

The 17,155 line file was the entire data access layer. A single, self-contained module. Reviewing the code, there was no de-duplication, no internal language, limited extraction of functions, and very little extraction of classes. It did have a clear boundary with an interface to preserve. It was a great target for refactoring.

The goal of refactoring an agentic code base is to spend tokens now in refactoring to make token consumption for future work lower. An experiment should be able to show that as this file was refactored the token cost of making separate feature implementations in this code base would decrease.

Precisely because agents never learn this was now possible to run as an experiment. I could prompt a fresh agent to make exactly the same change after every refactoring stage. Unlike a human engineer, the experiment would not be tainted by learning from previous steps.

1. Create an overall refactoring plan, following strict refactoring discipline.
2. Craft a representative change, described in a single prompt.
3. Establish a baseline cost of change: in a sub-agent, execute that prompt, including asking the sub-agent to report token consumption.
4. Throw away the change.
5. In a loop:
 1. Apply a single step of the overall refactoring.
 2. In a sub-agent, execute *exactly* the same change receiving the token cost of the change.
 3. Throw away the change.
6. Record all token costs, time to execute the change, and lines of code after each step of the refactoring, including the baseline.

The prompt used for the representative change and the refactoring steps applied are shown in the appendices, below.

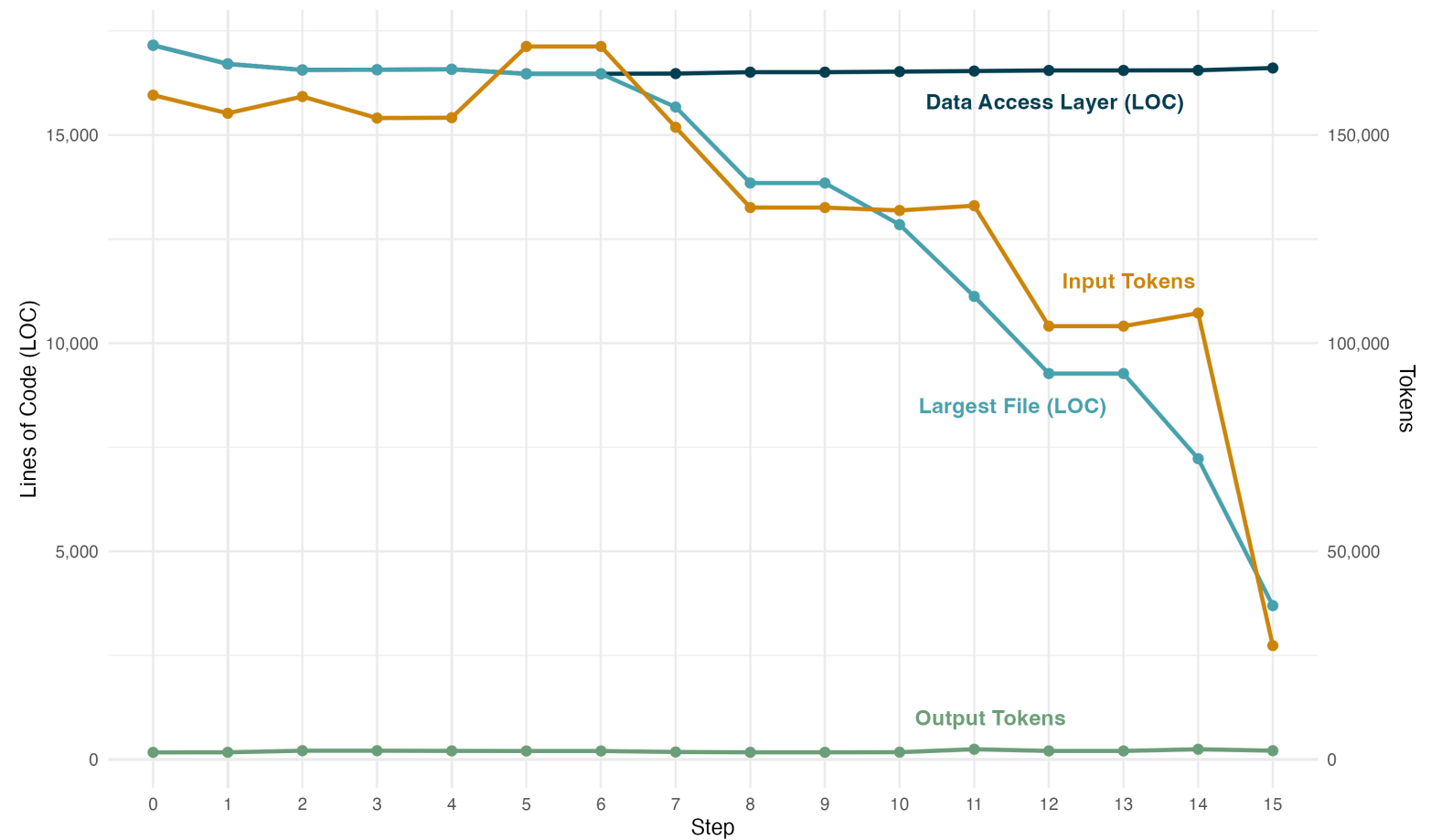
One caveat: Claude doesn't provide reliable methods for counting tokens live despite showing token counts, reporting tokens consumed per session, and billing for tokens. I'm assuming this is a temporary issue that will improve over time. Instead, the sub-agent reported the number of characters received and sent and used tiktoken to approximate tokens, by dividing character count by four.

Results

Step	Data Access Layer LoC	Largest file LoC	Total Rust LoC	Input tokens per change	Output tokens per change	Time per change (s)
Baseline	17,155	17,155	50,359	159,564	1,705	342
Step 1 (FirestoreClient)	16,706	16,706	49,910	155,205	1,723	530

Step	Data Access Layer LoC	Largest file LoC	Total Rust LoC	Input tokens per change	Output tokens per change	Time per change (s)
Step 2 (extract_doc_id, new_link)	16,562	16,562	49,766	159,227	2,105	574
Step 3 (link-query helpers)	16,567	16,567	49,771	154,054	2,105	524
Step 4 (FakeStore predicates)	16,577	16,577	49,781	154,146	2,060	654
Step 5 (value ctors)	16,469	16,469	49,673	171,251	2,036	1,353
Step 6 (FieldsBuilder)	16,469	16,469	49,673	171,251	2,036	1,353
Step 7 (queries.rs)	16,474	15,670	49,678	151,850	1,800	587
Step 8 (traits.rs)	16,508	13,845	49,712	132,558	1,723	446
Step 9 (traits/ split)	16,508	13,845	49,712	132,558	1,723	446
Step 10 (codec.rs)	16,521	12,846	49,725	131,871	1,750	540
Step 11 (fake_store.rs)	16,535	11,122	49,739	133,016	2,460	600
Step 12 (store/ split)	16,550	9,269	49,754	104,080	2,050	490
Step 13 (co-locate tests)	16,550	9,269	49,754	104,080	2,050	490
Step 14 (complete fake_store.rs)	16,553	7,225	49,757	107,205	2,453	523
Step 15 (store/ split)	16,608	3,695	49,812	27,360	2,113	454

The interesting metrics here are the total lines of code in the data access layer, the total lines of code in the *largest single file* in the data access layer and the input tokens consumed while producing the change.



This chart shows four things. The first point is the baseline, step 0, and then the same metrics are repeated *after* each refactoring step has been applied.

1. The total lines of code in the data access layer as a whole. Initially, this is just the single file I started with. This becomes many files as refactorings are applied. By the end there are 19 Rust files.
2. The lines of code in the single largest file in the data access layer. This started as the entirety of the data layer in the single initial file. By the end, the single largest file is a test library. Further refactoring passes could apply the same approach to this.
3. The total input tokens consumed by the sub-agent while applying the representative change.
4. The total output tokens produced by the sub-agent while applying the representative change.

Refactoring reduces token consumption

The results are clear. Input tokens stay fairly flat until the largest file starts to fall, and then they drop before, in the words of Claude, falling off a cliff.

Between the base line and the final refactoring, input tokens for the same task reduced from 159,564 to 27,360. A saving of 132,204 tokens, or 83%. And that saving is not a one-off.

Every single change that touches the data access layer from this point forward now costs significantly less.

How much of a saving? Assuming Sonnet 5 pricing at the time of writing of \$3/MTok, 39.7 cents. Not a lot. Does it multiply? How will this play out across debugging? More complicated features? This is refactoring only one portion of the code base, can the whole code base be aggressively refactored to find savings everywhere? How much would those refactorings *cost*?

This saving is because the agent has to read less code. But it is not because there is less code to read. The overall code in the data access layer as a whole has stayed fairly constant. Therefore to be able to bank this saving, the agent must be able to successfully identify the smallest subset of files necessary to read. The results make it appear this was happening. Reading the Claude Code thinking output and file read summaries as the change was being applied also indicates the sub-agent was successfully reading smaller and smaller sections of code each time.

In other words, randomly cutting the file into smaller files is unlikely to help as much: even if each file were smaller, the agent would be forced to read through many files looking for the relevant code. While the step with the biggest effect happens at the end, the previous steps were refactorings to set up this saving. This was not planned. It was simply a result of how refactoring typically proceeds: local file changes to extract duplication, before breaking down into smaller files once a repeating core emerges.

The refactoring did not make the representative change smaller. The number of tokens produced when writing code was largely unaffected: the output tokens do not move very much. Those tokens are five times the price of the input tokens. But, there are a lot less of them. Are there refactorings that could be applied to reduce output token production? I need a more complex sample change to explore these questions. The noise of the non-deterministic code generation process is hiding any variance caused by changes in the factoring of the code.

Notes on the process

Claude was not good at refactoring. If you read the prompt and the refactoring steps below, it's clear that the refactorings produced were directly in response to the prompt. Claude is unable to look at code, look at refactorings in general and work out which are suitable to apply: a human needs to actively guide it. This marries with wider experience in this app. The development harness includes an explicit refactoring step. That refactoring step did not prompt Claude into improving this file. More anecdotally, Claude.ai was better than Claude Code. I used both interfaces to create the refactoring plan. Claude Code spotted extract

function as the first step. Claude.ai went further and saw an entire client class to be extracted.

It was also bad at applying them. The mechanical act of refactoring was performed by writing Python scripts using grep and sed. These scripts frequently got confused by indentation. Oh, the irony. In addition, the single most valuable refactoring was missed in the first pass, and had to be re-applied as a follow-up step. This is why the number of steps in the figure don't match the refactoring steps in the appendix.

It took about eight hours to complete the entire experiment. This was mostly unattended. The only intervention was after six hours 40 minutes when it appeared to have finished, but had skipped that step and needed to be redirected. This experiment was running on slow hotel WiFi. I wondered if that contributed to time taken. But on deeper analysis of the code base, the cargo temporary build cache had become very large. Test execution was suffering, significantly.

Further work and broader implications

Unfortunately, it didn't occur to me to perform a count of the tokens required to create and execute the refactoring plan until it was already complete. I've looked at my aggregate consumption across the time window where I was doing this work, including designing and running the experiment. I can't say how many tokens were required to perform the refactoring. The upper bound is five million, however. This includes creating the refactoring plan twice, the work to design the experiment including the representative change, and various other tasks. Future work should include a more accurate count of tokens consumed to refactor.

This is just one experiment, on a significant application that is still greenfield and built and maintained by a single developer. But, I believe this is a potentially interesting first step. This effort shows the value, in time and money of refactoring. As well as measuring how expensive refactoring is. It would be interesting to look at more complex changes, at wider refactoring, refactoring continuously, and even the relative value of different refactoring approaches.

This is just the beginning.

Appendices

Note: These appendices include the prompts that I used, and the output that was returned. The only editing applied has been to remove the specific code changes to be made. These are included without editing to show how the agents were directed. There are no hidden

tricks. As such, there is some language in here that might be confusing. The error is in the original.

- The representative change
- Refactoring steps

latest article (Jul 30):
[The Economic Benefit of Refactoring](#)

previous article:
[Experiences with local models for coding](#)

