Cutting the Cost of Agentic Refactoring

Authors: Andy Soffer *(BrontoSource CTO)*
Date: 2026-07-02

Abstract

Large language model (LLM) agents are increasingly used to perform multi-step software engineering tasks such as refactoring. While capable, a general-purpose agent is many orders of magnitude more expensive than purpose-built tools. Providing a coding agent with access to efficient tooling can significantly improve its capabilities while controlling costs.

This paper reports a controlled comparison of a refactoring task performed two ways: With LLMs operating on their own, using only generally available tools, and by those same LLMs operating with the `bronto-refactor` tool exposed through the `/bronto:refactor` Claude plugin. Independent of the model, the plugin saw a 14-26x cost savings (depending on model) without any additional human intervention. We describe the task, the measurement methodology, the results, and the mechanisms that account for the difference. We also discuss the limits of what a single-task study can establish.

Introduction

Coding agents built on frontier LLMs can plan and execute large-scale engineering work, but their cost can be prohibitive, even for relatively simple tasks. An unaided agent typically:

1. Reads portions of the codebase to understand structure and conventions.
2. Searches repeatedly to find every site affected by a change.
3. Applies edits, runs checks, observes failures, and iterates.

While this approach fares significantly better than placing the entire codebase in context for each request, it introduces other scaling problems. Notably, because the entirety of the earlier conversation is sent to the LLM on each request, costs grow *quadratically* in the number of requests. Iterative approaches increase the number of requests, bloating costs for tasks that need more than a few iterations.

There is another approach that can significantly reduce the cost. One could architect an agentic system that offloads most of the reasoning burden to static analysis tools. In this paradigm, rather than using an LLM to generate code or orchestrate complex refactoring steps, the agent acts primarily as a high-level router that interprets user intent and maps it to specific, static analysis primitives. This approach shifts the reliance from the model's probabilistic reasoning to

the determinism of the underlying tools, allowing for greater cost efficiency and correctness guarantees by minimizing the surface area for model hallucinations.

The question this paper answers is narrow and practical: *To what extent do efficient static analysis tools improve the capabilities and reduce the costs of agentic workflows?*

The `bronto-refactor` tool

`bronto-refactor` is a deterministic, static-analysis-based tool for C and C++ that generates patches based on refactoring specifications. It relies on semantic understanding available through compilation, including knowledge of types, macros, implicit conversions, compile-time evaluation, and many other subtleties that often may have no textual indication in any given source file.

The `/bronto:refactor` plugin

`/bronto:refactor` is a Claude Plugin wrapping the calls to `bronto-refactor`. When invoked, it asks the user what they want to refactor. From the user's response, it synthesizes a refactoring plan and suggests a sequence of steps. Once the user accepts the plan, Claude executes the plan according to the steps. Below is an example Claude session (edited slightly for legibility in this paper):

```
❯ /bronto:refactor

⏺ You already have compile_commands.json set up, so we can skip setup.

What would you like to refactor?

Describe the change you have in mind, and I'll break it down into a sequence
of steps (additions, reimplementations, propagations, cleanups, and
deletions) for you to review before we execute anything.

For example:
- Replacing a type/type-alias with another
- Replacing a function with a different implementation
- Adding, removing, or changing a namespace
- Replacing all calls to one function with another

The more specific you can be about the "replace all X with Y" you want, the
better I can plan it.

❯ remove all `using namespace std`. the code needs to compile afterwards.
```

⏺ Let me find all the `using namespace std` directives in your codebase to understand the scope.

Found them. Here's the breakdown. There are **22 real `using namespace std;` directives**.

Let me note the non-targets:
- XXXXX is a **comment**, not a real directive — skip.
- testdata/** are **test fixtures**. Modifying them would change test inputs/expectations — I'll exclude them unless you say otherwise.

Plan

This is a single **renamespacing** task, which maps cleanly onto one primitive.

Step 1 — Propagate (Renamespacing).
Annotate each of the 22 `using namespace std;` directives with the `BRONTO_INLINE()` prefix:

```
BRONTO_INLINE() using namespace std;
```

Then run:

```
run-bronto-refactor --db compile_commands.json
```

The tool walks every translation unit in the compile DB, requalifies all previously-unqualified `std` symbols (e.g. `vector` → `std::vector`, `move` → `std::move`) across all users of these headers, and removes the `using namespace std;` directives. Because all 22 are the same kind of change, they go in **one** refactor run.

No separate Addition/Reimplement/Cleanup/Deletion steps are needed — the renamespacing primitive handles requalification and removal of the directive in one pass.

Confirm and I'll execute. Should I exclude the `testdata/` files as proposed, or do you want those included too?

Model Cost

The cost of achieving a refactoring can be broken down into the costs of executing a model, typically metered in “tokens,” and the costs of the tools invoked by the model. Even the most uncharitable approximations of tool execution cost model them at 4 orders of magnitude

cheaper than the cost of an LLM, so for the purposes of this experiment, we simply ignore their costs.[1]

As of June 2026, Anthropic billed at the following rates (USD) per million tokens.

Model	Input Tokens	5min Cache Writes	1hr Cache Writes	Cache Reads	Output Tokens
Fable 5	$10	$12.50	$20	$1.00	$50
Opus 4.8	$5	$6.25	$10	$0.50	$25
Sonnet 4.6	$3	$3.75	$6	$0.30	$15
Haiku 4.5	$1	$1.25	$2	$0.10	$5

For example, a simple request that sent 1000 input tokens and received 5000 output tokens (assuming no cache reads or writes) would cost $0.026 with Haiku, $0.078 with Sonnet, $0.13 with Opus, and $0.26 with Fable. These are exceedingly low estimates for any real-life task.

Experiment

In order to test the hypothesis that refactoring costs could be dramatically reduced by static-analysis, we ran an experiment, asking several models to perform a specific refactoring task, both with and without the use of the `bronto-refactor` tool.

The task

We evaluated a single refactoring task on a codebase with ~50,000 lines of C++23 code. The codebase was prepared by stripping all `std::` prefixes and adding `using namespace std;` to sufficiently many headers to ensure that the code compiled. The task was to undo this change: That is, to remove every instance of `using namespace std;` and add `std::` qualifications to every symbol from the `std` namespace.

This particular task was chosen because it represents a balance between what can be done with only textual information and what requires deeper programming language understanding. A task such as “Rename all uses of **MyUniquelyNamedFunction**” would be uninteresting as `sed` could achieve this task perfectly (and indeed, even the smallest frontier models in 2026 will decide to use `sed` for this task). On the other extreme, a refactoring task that has no textual indication (say, replacing all implicit casts with calls to `static_cast`) would result in LLMs

[1] For example, `bronto-refactor` draws ~25W on an M3 Macbook, and processes a 50,000 line C++ codebase in ~7 minutes. Assuming the most expensive electricity costs in the United States in June 2026, the cost is $0.0013 per execution. Comparing this to our results, the cheapest execution with an LLM on the same codebase was $14.16.

unable to make such changes effectively, and unable to verify whether or not they had succeeded. While `bronto-refactor` is perfectly capable of this task, it would not be meaningful to make this comparison. With qualifications to names in the standard library, a language model has the possibility of succeeding at the task, purely by looking for names like `vector` or `string_view` and adding qualifications.

This task also serves as a boundary test. Because `bronto-refactor` intentionally avoids modifications within macro expansions, it leaves residual work that requires the LLM's contextual reasoning capabilities. By selecting a task where neither approach offers a 'silver bullet' solution, we create a more rigorous baseline for evaluating how LLMs and static analysis tools perform in tandem.

Methodology

For each of the models listed below, we took the following approach three times

- Claude Haiku 4.5
- Claude Sonnet 4.6
- Claude Opus 4.8
- Claude Fable 5

When not given access to `bronto-refactor`,

1. Prompt the model with *"remove all `using namespace std`. the code needs to compile afterwards."*
2. While the end result did not compile, repeatedly prompt the model until it did with prompts such as *"the code doesn't compile. run `bazel build ...` to see the problems. fix them".*
3. Record the total token cost (including the cost for any subagents the model chose to launch).

When given access to `bronto-refactor`,

1. Run the `/bronto:refactor` Claude plugin.
2. When the model asks for the desired refactoring task, respond with the same prompt used above: *"remove all `using namespace std`. the code needs to compile afterwards."*
3. While the end result did not compile, repeatedly prompt the model until it did with prompts such as *"the code doesn't compile. run `bazel build ...` to see the problems. fix them".*
4. Record the total token cost (including the cost for any subagents the model chose to launch)

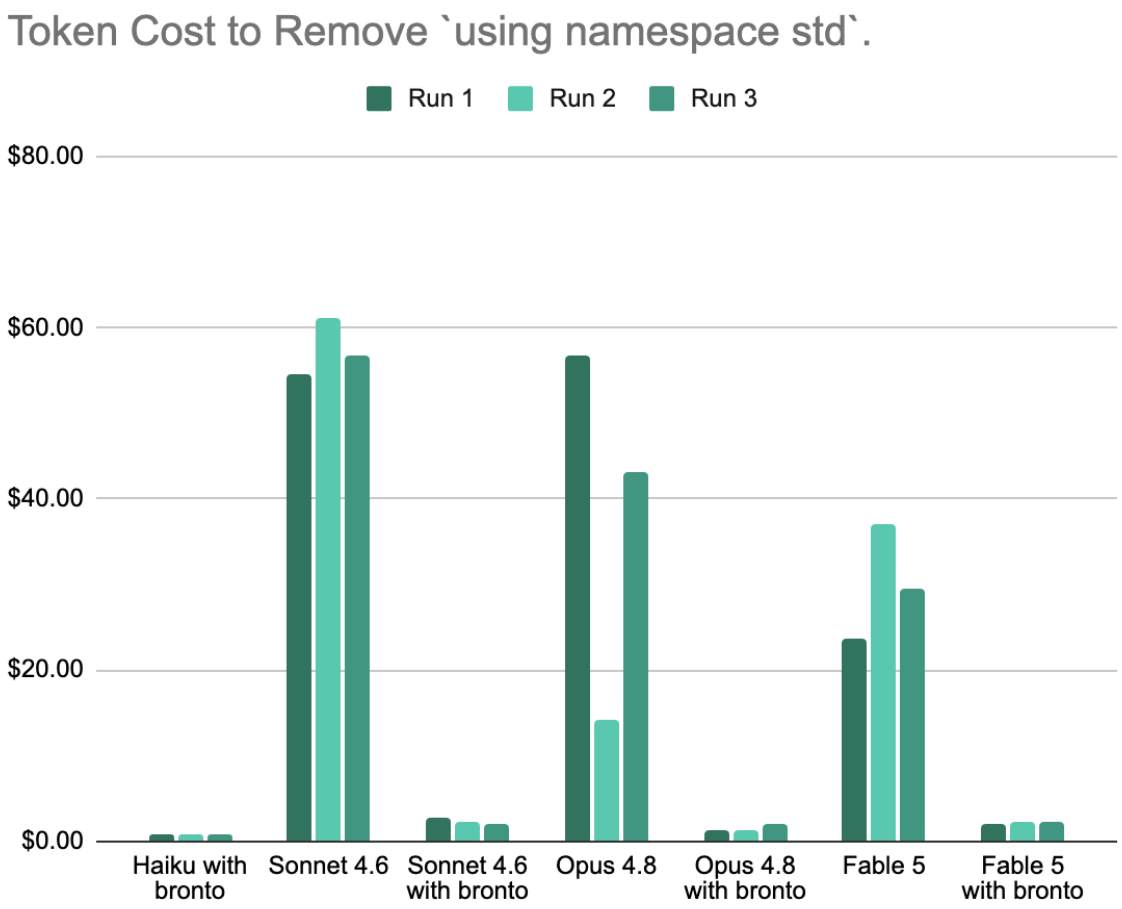
Notably, the only difference between these workflows was whether or not the `/bronto:refactor` plugin was invoked. Each run occurred on a codebase without any CLAUDE.md file or special instructions; this was a deliberate control choice to isolate the raw

tool-vs-model performance and remove potential confounding variables from pre-optimized system prompts.[2]

Results

Overview

The results are summarized in the figure below. Costs are measured in dollars according to the June 2026 billing rates for each of these Claude models. This includes the cost attributions of input and output tokens as well as cache reads and writes. A full breakdown can be found in the table below



On average, `/bronto:refactor` with Sonnet produced a 25x cost reduction over using Sonnet without `/bronto:refactor`, a 26x cost reduction when using Opus, and a 14x cost

[2] In an earlier version of this experiment, models looked at the version control history, saw the changes that prepared the codebase and decided the solution would be to revert the commits that prepared the codebase.

reduction when using Fable. Haiku on its own was unable to perform the refactoring task at all, even with repeated human intervention[3]. Using `/bronto:refactor`, Haiku did complete the task with anywhere from 0 to 3 additional prompts.

Data

Below is the breakdown of each experimental run measured in terms of tokens spent and USD. Recall that, while we ought to also consider the costs of tool execution, the most conservative overestimates place these costs 4 orders of magnitude less than the overall costs of any experimental run, so they can be safely neglected.

Model and execution	Input Tokens	Output Tokens	5m Cache Tokens	1hr Cache Tokens	Cache Read Tokens	Cost (USD)
Haiku 4.5 with bronto-refactor (run 1)	505	19,227	0	83,568	5,315,100	$0.80
Haiku 4.5 with bronto-refactor (run 2)	506	19,621	0	81,448	5,268,453	$0.79
Haiku 4.5 with bronto-refactor (run 3)	443	22,894	0	195,662	3,427,750	$0.85
Sonnet 4.6 (run 1)	7,573	557,897	1,585,092	1,778,105	98,267,734	$54.48
Sonnet 4.6 (run 2)	5,709	411,793	1,796,315	59,622	159,607,911	$61.17
Sonnet 4.6 (run 3)	3,092	403,093	182,745	1,149,794	143,500,693	$56.69
Sonnet 4.6 with bronto-refactor (run 1)	41	23,337	0	281,007	2,145,179	$2.68
Sonnet 4.6 with bronto-refactor (run 2)	39	21,271	0	220,106	2,184,095	$2.30
Sonnet 4.6 with bronto-refactor (run 3)	35	16,657	0	194,727	1,825,140	$1.97
Opus 4.8 (run 1)	134,131	492,059	1,793,098	332,893	58,579,899	$56.80
Opus 4.8 (run 2)	5,298	155,022	0	232,559	15,857,090	$14.16
Opus 4.8 (run 3)	115,568	521,475	2,158,475	153,137	29,135,559	$43.20
Opus 4.8 with bronto-refactor (run 1)	3,352	15,788	0	33,766	994,094	$1.25
Opus 4.8 with bronto-refactor (run 2)	4,329	13,846	0	31,295	1,125,051	$1.24
Opus 4.8 with bronto-refactor (run 3)	6,196	22,703	0	49,287	2,007,923	$2.10
Fable 5 (run 1)	10,378	61,239	0	231,586	15,761,197	$23.56
Fable 5 (run 2)	21,979	128,234	491,935	294,936	18,365,904	$37.05
Fable 5 (run 3)	15,618	98,663	0	367,980	17,102,311	$29.55
Fable 5 with bronto-refactor (run 1)	3,251	9,905	0	28,600	941,064	$2.04
Fable 5 with bronto-refactor (run 2)	3,254	11,955	0	28,627	930,448	$2.13
Fable 5 with bronto-refactor (run 3)	3,262	10,450	0	31,692	1,116,654	$2.31

[3] Presumably with enough intervention this could have succeeded. While we did not set a particular threshold for when we would stop attempting, we believe that our attempts exceeded what any reasonable human would have done for this refactoring task.

Interventions

The following details summarize the human interventions required to produce the desired output.

- For Haiku with `bronto-refactor`, run 2 required no additional intervention beyond the initial execution. Runs 1 and 3 both required several prompts to remind Claude that the code must compile afterwards, and it should continue looking at compilation failures and fixing them.
- For Sonnet with `bronto-refactor`, all three runs required a single additional prompt to reach the desired completion state.
- For Sonnet without `bronto-refactor`, no additional prompting was required.
- For Opus and Fable, both with and without `bronto-refactor`, no additional prompting was required.

We expect that the issue of requiring additional prompting could be mostly mitigated by tweaks to the plugin itself, which does not mention anything about requiring the code compile afterwards.

Analysis of LLM Inefficiencies

With the exception of Haiku, unaided models were capable of reaching a solution. However, the high token cost reveals the inefficiency of autonomous error correction without specialized tooling. The primary drivers of this expense were two-fold:

1. **Iterative Self-Correction:** Models spent the majority of their token budget on repetitive cycles; generating speculative patches, recompiling, observing compilation failures, and rolling back changes. Each iteration required re-reading and re-processing context, compounding the cost.
2. **Subagent Overhead:** To accelerate the work, models often launched multiple subagents to handle different parts of the codebase simultaneously. These agents partitioned the codebase by directory, and then proceeded editing them in isolation. The editing mechanisms could have been shared across all partitions, and would have resulted in significantly fewer model invocations.

Opus' Anomalously low cost

In the second iteration of running Opus without `bronto-refactor`, costs were anomalously low. On inspecting the transcript, this was indeed a real execution. What differed between this run and the other two was that Claude decided **not** to launch subagents for this task, whereas for runs 1 and 3, Claude handed portions of the codebase to each of several agents. It's clear how the costs were reduced: For each of these runs, Claude wrote small programs in sed, perl, or python to achieve the goal, and ran the programs. These programs are just as efficient running on a subset of the codebase as they are on the entire codebase, so the primary difference in cost was the overhead of launching agents, each with their own context, and

having them duplicate work. Multiple separate agents also inherently produce less consistent outcomes.

Costs Relative to Model Size

A general pattern we see here is, perhaps surprisingly, that larger models that are more expensive per-token end up being **cheaper to run** without effective tools. Looking at the actual transcripts for these runs explains this pattern. These models are not somehow guessing the correct replacements more accurately. They are however more aggressive about using tools to perform tasks on their behalf. This is exactly the benefit `bronto-refactor` provides. What's more, when given a sufficiently capable tool, larger models stop providing extra value for their cost.

Scaling Behavior

Agents fundamentally have a cost **quadratic** in the number of conversation turns. This means that effective tools produce cost savings that are **quadratic** in the number of iterative self-correction steps they eliminate. This is clearly seen in the cost savings provided by `bronto-refactor`. The savings only grow with larger codebases. A 14-26x cost reduction was only for 50,000 lines of code; for a 1,000,000-line codebase, we would expect a significantly larger cost reduction ratio.

Conclusion

The results of this study clearly demonstrate that providing LLMs with deterministic, purpose-built tooling significantly shifts the economics of software refactoring. By offloading complex code analysis and transformation to a tool like `bronto-refactor`, we bypass the inherent inefficiencies of LLM-based approaches; specifically, the high token consumption associated with iterative trial-and-error, context window bloat, and the overhead of managing multiple subagents.

Our experiment showed a 14-26x cost reduction when using the `/bronto:refactor` plugin compared to unaided model performance. Perhaps more importantly, the plugin enabled smaller, more efficient models like Haiku to successfully complete tasks that were otherwise infeasible for them.

Ultimately, the most effective architecture for AI-assisted engineering is not just about reducing token spend; it's about constraining the agent's probabilistic nature with deterministic tooling. This hybrid provides the reliability required for production-grade engineering, while still enjoying the benefits of agentic development. As the industry looks to scale agentic workflows across larger codebases, this strategy, where LLMs orchestrate the execution of specialized tools, will be essential to moving from experimental prototypes to sustainable, production-grade automation.