Removing dead code: the CodeLaser engine versus Claude Code

The CodeLaser engine and Claude Code were given the same four open-source Java projects, the same definition of what the outside world can reach, and the same instruction: remove everything that is unreachable, and leave a project that still builds. Each was measured on what it removed, what it cost, and what could be established about the result.

ISSUED
2026-09-22

CONTACT
tom.tourwe@codelaser.io

TABLE OF CONTENTS

IN SHORT

What the experiments show

- **What the CodeLaser engine did.** Given the four projects and a list of what the outside world reaches, it removed everything unreachable from each one — from twenty declarations in the smallest to about three and a half thousand in the largest — in between one and a half and six and a half minutes per project, without compiling anything to decide. Every result builds, and none of the checks run afterwards found a problem in any of the four.
- **The agent takes between 15 and 54 times longer than the engine, and spends tokens, which the engine does not.** The CodeLaser engine took between 1m 36s and 6m 25s per project; a single agent attempt took between 30 and 121 minutes, 15 to 54 times longer, and consumed between 10.7 and 47.1 million input tokens. The engine is a compiled program and consumes none.
- **The agent is non-deterministic.** It does not do the same thing twice. Three attempts on Caffeine, from an identical starting point and an identical instruction, removed 18, 40 and 231 declarations. Three on Timefold Solver removed 1,706, 1,478 and 1,317. Each attempt wrote its own analysis program and no two were alike — one wrote none at all and used the compiler as its oracle. None of them reached what the engine does: the same removals on every run, and a rule for each of the ways a declaration stays alive without any Java code referring to it.
- **When the agent runs out of time, there is nothing to show for it.** Three of the eight agent attempts never finished. On Trino the agent spent 196 minutes across two attempts and 35.6 million input tokens, and removed nothing at all: the time went on writing and repairing an analysis program rather than on removing code. The engine finished every project it was given.
- **The agent deletes declarations it was told in writing are reachable.** Both sides were handed the same list of everything the outside world reaches. On QuestDB the agent deleted six entries from that list, among them a test method and five declarations the project's own authors marked as deliberately uncalled. The engine deleted none, on any project.
- **The agent deletes code in ways that still compile and are still wrong.** Four agent attempts left classes with no declared constructor, so Java silently supplies a public one and a class that could not be instantiated now can be. Agent attempts deleted fields whose initialising expression runs code, up to 128 in one attempt. One agent attempt emptied a list of named options that a configuration file selects by name, disabling a feature at run time. Every one of these passes a full build. None of the engine's runs did any of it.

Every count was taken by parsing the source before and after each run, with the same parser for both sides. The sections that follow set out the evidence.

Removing dead code is worth doing, and an agent is a good thing to have driving it. What these experiments set out to establish is where the work should sit. Solving a problem by writing code for it is what an agent does, so an agent handed this task starts by building a dead-code analysis of its own. That is the natural first move and it is the wrong one: an analysis that resolves every name in a Java project is far more than can be built in two hours, and these eight attempts spent most of their time and tokens finding that out.

Give the same agent the CodeLaser engine and the work divides the right way round. The engine does the code mechanics: resolving every name, working out what nothing reaches, and making the removal. The agent does what an agent is good at — deciding what to tackle, judging the findings, explaining the result. Its tokens go on reasoning and on talking to you, not on rediscovering how Java works.

THE SETUP

Experimental setup

Dead code is code that nothing can reach: a method nobody calls, a class nobody names, a field nobody reads. Removing it is useful, tedious and risky, because "nothing reaches this" is a claim about the whole program.

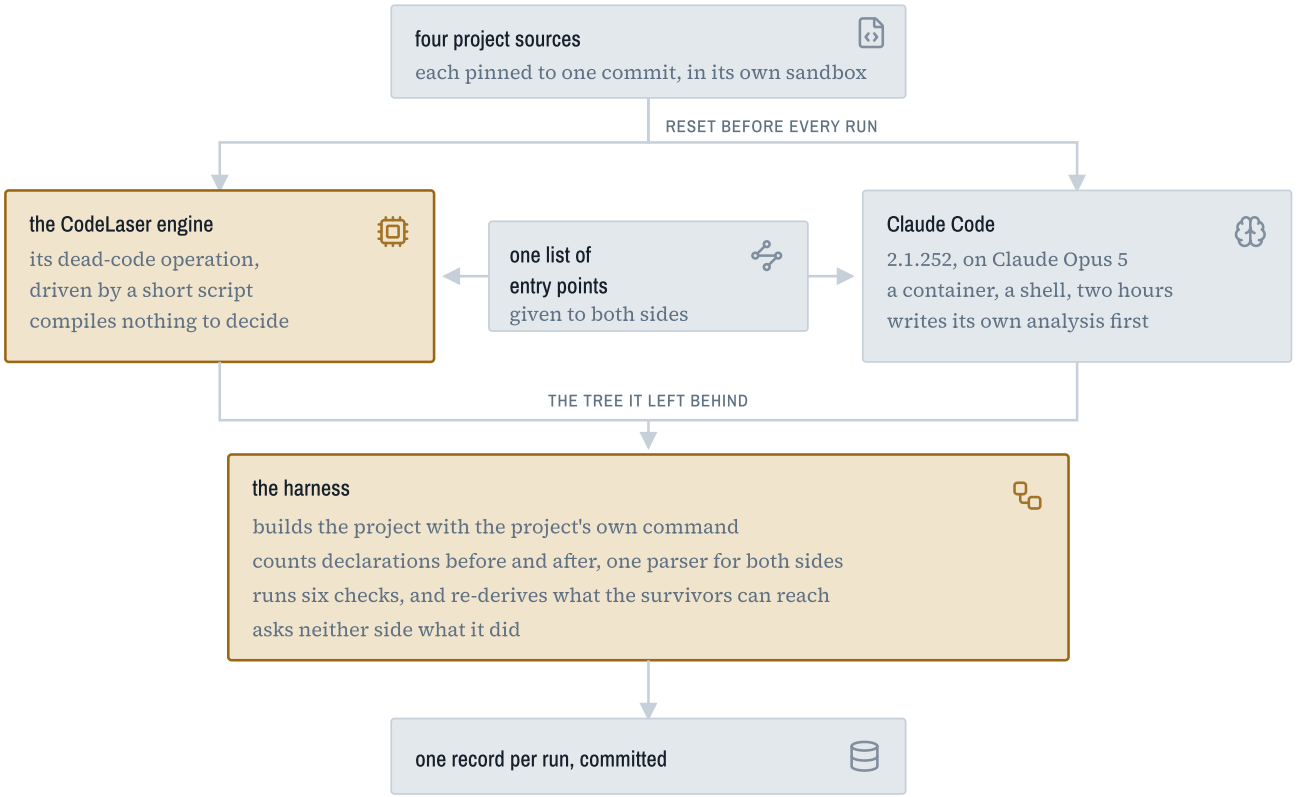
Two approaches were compared on the same task. The first is **the CodeLaser engine**, running its dead-code operation: it parses the project, builds a graph of what refers to what, and deletes what cannot be reached. The second is **Claude Code**, the same coding agent a developer would use interactively, working autonomously with full access to the project and a shell. Version 2.1.252, on Claude Opus 5, the same build and model in all eight attempts. Where this document says **the agent**, that is what it means.

Claude is on both sides of this comparison. The engine's side uses a Claude instance as well: it produced the entry-point list both sides were given, and the short script that drives the engine. So this is not a tool measured against a rival. It is the same agent, asked in one case to drive a program built for the job and in the other to build that program from nothing.

Components of the setup

Four: the four project sources, the engine, the agent, and the harness that ran and measured both.

HOW THE PIECES WERE CONNECTED



The project sources. Four public repositories, each pinned to one commit, each in a sandbox of its own, each reset before every run. The upstream repositories were never written to.

- Caffeine, `07c6e370c`
- Timefold Solver, `290c87fc53`
- Trino, `27e3b9c8d62`
- QuestDB, `714b750076`

The engine. The CodeLaser engine and its dead-code operation, driven by a short script. It reads the project, resolves every name, and works out what nothing can reach. It compiles nothing of its own and it asks nothing of a person.

The agent. Claude Code, in an isolated container with network access and a shell, capped at two hours per attempt. What it did with that was its own choice.

The harness. The part that makes this a measurement rather than two anecdotes. It never asks either side what it did. For every run, on either side, it:

- resets the project to its pinned commit, and records what it reset to

- builds it with the project's own command
- runs the arm, engine or agent
- counts the declarations in the tree before and afterwards, with the same parser for both sides, keyed by file, enclosing type and signature
- runs six checks over the result — among them whether a surviving class lost every constructor it declared, whether an enum lost a constant while surviving code still reads constant positions, and whether a file the project declares as generated-only was edited
- re-derives what the surviving code can reach, to see whether anything on the entry-point list was deleted
- writes everything it saw to one directory for that run

PROJECT	JAVA FILES	LINES	BUILD MODULES	ENTRY POINTS SUPPLIED
Caffeine	704	160,501		6,508
Timefold Solver	3,652	354,521		12,804
Trino	11,154	2,022,701	209	38,022
QuestDB	6,026	2,286,193		40,518

THE FOUR PROJECTS, AND WHAT EACH SIDE WAS GIVEN

Caffeine is a high-performance in-memory caching library, small and mature. Timefold Solver is an optimisation engine for problems such as vehicle routing and shift rostering, with a heavy test suite. Trino is a distributed SQL query engine, spread across 209 separately built modules. QuestDB is a time-series database; 161 of its files are a benchmark module that generates code.

Both sides were given the same starting point

Most real projects contain code that no other Java code refers to, and which is nevertheless alive: a test method a test runner discovers, a class a framework creates by name, a handler named in a configuration file, a command-line entry point. Any tool that ignores this will delete working code.

ON THE LIST BECAUSE	COUNT
a test the runner discovers	60,306
a framework annotation marks it	17,586
it is a member of an annotated type	12,387
its own author declared it unused on purpose	2,011
a string in the source names it	1,583
a GraalVM reflection config names it	777
something outside the measured code uses it	637
it is a `main` method	448
a resource file names it	384
a framework selects the enum constant	336
a service loader finds it	185
serialization reaches it	161
a JUnit argument provider supplies it	139
it inherits a test method	96
a JUnit 3 suite names it	5

WHY EACH OF THE 97,041 ENTRY POINTS IS ON THE LIST, ACROSS ALL FOUR PROJECTS

Two thirds of the list is tests. The rest is the part that makes dead code hard: a name in a string, a resource file, a reflection config, a service loader — fifteen different reasons a declaration is alive while no Java code refers to it.

So both approaches were handed the same **list of entry points** — every declaration something outside the code can reach, and the reason it is on the list. That list was produced with Claude Code and refined over several rounds. It is not complete, and does not need to be: **both sides work from the same definition of "alive".**

What each side received

The engine was given the project and the entry-point list, and nothing else.

The agent was given a written instruction: remove as much dead code as possible; removing one declaration is what makes the next one dead, so repeat until nothing more is found; the project must build when you stop, and you may rebuild as often as you like. It was explicitly told it could change *any* source file — that deleting a type often requires editing the places that mention it, and that confining itself to deleting unreferenced files was not the goal. It was free to write its own programs to do the job. The only limits were that it should not change what the project under test does, and should not touch its build configuration.

Each agent attempt was capped at **two hours** of wall-clock time. Because the agent does not give the same answer twice, **three separate attempts** were run against Caffeine and against Timefold Solver, so that the spread could be seen. Trino and QuestDB are too expensive to repeat that way and received **one effort each**; for those two the figure reported is the whole of the time and tokens that effort consumed, and the spread is unknown.

THE TOOL

How the CodeLaser engine removes dead code

Four things about the CodeLaser engine explain the results that follow.

First, the task is two problems rather than one. Finding the dead code is the easier of the two; taking it out is where the difficulty is.

Finding dead code has a cheap approximation. Counting references and walking outward from the entry points gets most of the way, which is why every agent attempt wrote code to do exactly that. What a reference count cannot find is a declaration whose reachability is decided outside the Java source. Two of the four silent failures below are of that kind: an enum whose constants a configuration file selects by name, and six declarations the agent deleted although the entry-point list it had been given named them as reachable.

Removing dead code has no approximation at all. Deleting a declaration means repairing every place that referred to it, and following the consequences: methods that only it called are now dead too, imports and overrides and signatures have to be corrected, and the code left behind can often be simplified further. Each deletion can make another declaration dead, so the work repeats until a pass finds nothing. The other two silent failures are here: the code really was unreachable, the agent was right to find it, and deleting it still changed what the program does.

The engine resolves every name before it decides anything. The project is parsed once and every name is resolved to what it denotes: which declaration a call reaches, which type a name means. Reachability is computed over that, outward from the entry points. What it costs to work from compiled output instead is set out under *Four repairs caused by working from compiled classes*.

The engine knows where reachability is decided outside the code. A name in a configuration file, a class loaded from a string, an enum constant selected by text at run time: none of these is a reference the compiler can see, and counting references will not find them. The engine carries a rule for each such mechanism. One of them is set out in full in the fourth example below: a call to `valueOf` or `values()` marks every constant in that enum as used, because which one will be asked for is not decided until the program runs. That single rule protects the ten strategy classes an agent attempt deleted.

The engine refuses rather than guesses. Where it cannot show that a removal is safe, it declines to make it and reports why. The clearest case is a field whose initialising expression calls a method: showing that the call only computes a value means following every call it makes, through code that may not be available to read. The engine reports such a field as unused and leaves it in place. That caution costs it real removals — four on Caffeine, where the expressions turned out to be harmless. The third example below is the same situation going the other way, where an agent attempt deleted 128 such fields and broke the tests that depended on them.

The engine does not need to compile anything to decide. It determines reachability from the source directly. The build afterwards is verification, not part of the method, which is why the engine's **builds run** figure is zero on all four projects while the agent's is between 7 and 25.

RESULTS

What each run removed and what it cost

One table per project follows. Counts are of **declarations** — individual types, methods, constructors, fields and enum constants — and were measured independently by parsing the project before and after each run, not taken from either side's own report of what it did. The five kinds sum to the total in every row.

Caffeine

704 files, 160,501 lines, 6,508 entry points supplied to both sides.

WHAT WAS MEASURED	THE CODELASER ENGINE	CLAUDE ATTEMPT 1	CLAUDE ATTEMPT 2	CLAUDE ATTEMPT 3
types	**0**	0	0	19
methods	**13**	14	14	87
constructors	**0**	1	18	35
fields	**3**	3	3	72
enum constants	**4**	0	5	18
declarations removed	**20**	18	40	231
lines removed	**72**	94	117	1,357
builds run	**0**	10	7	25
time	**1m 36s**	30 min	34 min	74 min
input tokens	—	12.9M	15.1M	47.1M
output tokens	—	103k	119k	178k
outcome	🟢 **compiles**	🟠 compiles, with identified issues	🟠 compiles, with identified issues	🟠 compiles, with identified issues

WHAT EACH RUN REMOVED AND WHAT IT COST ON CAFFEINE

Timefold Solver

3,652 files, 354,521 lines, 12,804 entry points supplied to both sides.

WHAT WAS MEASURED	THE CODELASER ENGINE	CLAUDE ATTEMPT 1	CLAUDE ATTEMPT 2	CLAUDE ATTEMPT 3
types	**117**	85	83	88
methods	**1,116**	1,300	965	966
constructors	**115**	178	128	102
fields	**214**	131	283	142
enum constants	**29**	12	19	19
declarations removed	**1,591**	1,706	1,478	1,317
lines removed	**11,052**	11,733	11,933	9,546
builds run	**0**	24	19	17
time	**2m 14s**	121 min (cut off)	34 min	35 min
input tokens	—	17.9M	10.7M	12.7M
output tokens	—	136k	103k	116k
outcome	🟢 **compiles**	🔴 does not compile	🟠 compiles, with identified issues	🟠 compiles, with identified issues

WHAT EACH RUN REMOVED AND WHAT IT COST ON TIMEFOLD SOLVER

Trino

11,154 files across 209 modules, 38,022 entry points supplied to both sides.

WHAT WAS MEASURED	THE CODELASER ENGINE	CLAUDE CODE
types	**64**	0
methods	**1,090**	0
constructors	**33**	0
fields	**247**	0
enum constants	**44**	0
declarations removed	**1,478**	0
lines removed	**16,138**	0
builds run	**0**	—
time	**6m 25s**	196 min, two attempts
input tokens	—	35.6M
output tokens	—	286k
outcome	🟢 **compiles**	🔴 nothing removed

WHAT EACH RUN REMOVED AND WHAT IT COST ON TRINO

QuestDB

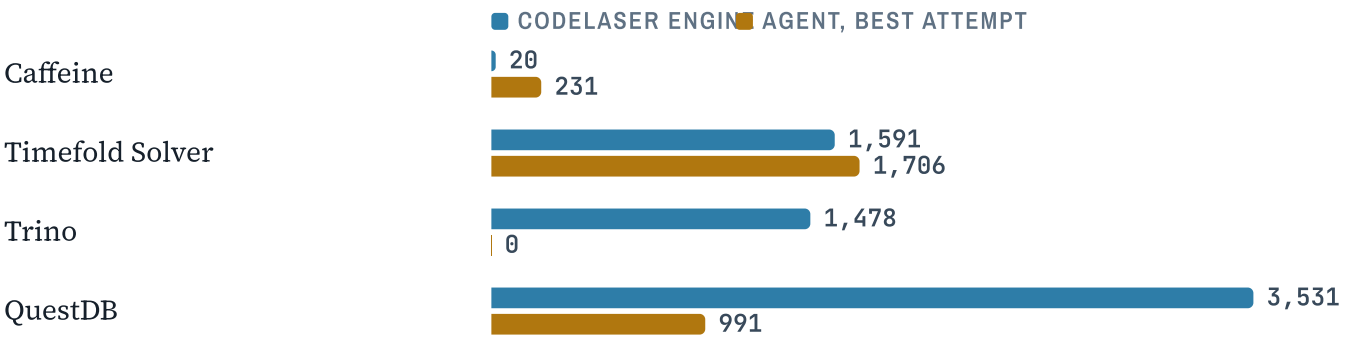
6,026 files, 40,518 entry points supplied to both sides.

WHAT WAS MEASURED	THE CODELASER ENGINE	CLAUDE CODE
types	**662**	43
methods	**2,235**	779
constructors	**228**	45
fields	**396**	124
enum constants	**10**	0
declarations removed	**3,531**	991
lines removed	**50,321**	22,174
builds run	**0**	—
time	**4m 10s**	90 min
input tokens	—	32.0M
output tokens	—	178k
outcome	**compiles**	compiles, with identified issues

WHAT EACH RUN REMOVED AND WHAT IT COST ON QUESTDB

> Trino received one effort across two attempts. The first ran 76 minutes and was killed by an expired OAuth token, which is an infrastructure failure rather than the agent failing at the task. The second ran to completion in 120 minutes. The 196 minutes and the token figure cover both, which is why they exceed the two-hour cap on a single attempt.

DECLARATIONS REMOVED, THE CODELASER ENGINE AGAINST THE AGENT'S BEST ATTEMPT



The agent removed more than the engine on two of the four, and the observations that follow are about what it removed to get there.

QuestDB is the agent's best result in this benchmark. It is the only attempt on any project that stopped because it had run out of things to find rather than because its time ran out, the only one that left no class without a constructor, and the only one that went looking for names written as text before anything drew its attention to them. It removed 991 declarations against the engine's 3,531, in twenty-one times the time. Its own account of the difference is that it deliberately left 273 methods that override one another across a hierarchy and 184 protected constructors, categories it chose not to touch, not ones it failed to find.

On Trino the agent removed nothing, for a reason worth setting out. Late in the effort it did have a project that built cleanly with 3,531 methods and 357 fields removed. It then checked its own work, found it had deleted a method that is only ever named by a piece of text elsewhere in the project, undid everything, and spent its remaining time building a safeguard against that mistake. The clock stopped it while it was applying the removals again. It traded a finished answer for a correctness fix and ran out of time, which is a result about the approach, not an accident.

Builds run counts compilations performed during the removal work itself. Both sides additionally get one build before they start, so that a compiled project is available, and one build afterwards to verify the result. The engine performs none of its own: it reads the source and decides in a single pass.

Input tokens include all context the agent re-read on each step, which is most of the figure. Each row is one attempt, except Trino's, which covers both of its attempts. **The CodeLaser engine consumes no tokens when it runs**: it is a compiled program. It is not free of AI involvement, though: the short script that drives it, and the entry-point list both sides were given, were produced with Claude Code beforehand. That is a one-off preparation cost, paid once rather than on every run, and it is not measured here.

Outcome records only what can actually be established. Whether a project compiles can be checked. Whether a removal is *right* is not something this benchmark can decide: a run could have deleted something it should have kept, or kept something it should have deleted, in ways nobody has looked for. Neither side's output has been audited line by line — including the engine's.

So the outcome is one of three things. **Compiles** means the project builds and the harness's own checks — not the project's test suite — found nothing. **Compiles, with identified issues** means it builds but at least one of those checks found a specific, confirmed problem. **Does not compile** means the build fails. On Trino the outcome records something different again: an effort that ended before producing anything.

The number of declarations removed is not by itself a score. A run that removes more may simply have removed things it should not have, which is what happened in every one of the six attempts on Caffeine and Timefold, and is the subject of the third observation below. Nor is a low number automatically a poor result: on Trino the agent removed nothing precisely *because* it stopped to correct a mistake of that kind.

OBSERVATION

The agent costs far more, in time and in tokens

On Caffeine the CodeLaser engine finished in 1m 36s. The agent took between 30 and 74 minutes. On Timefold Solver the engine took 2m 14s; the agent took 34 minutes at best, and in one attempt used the entire two-hour cap without finishing. Timefold's three attempts are both ends of the range quoted above: 15 times the engine at best and 54 times at worst. It is not explained by the agent doing more work: on Timefold its best attempt removed slightly *less* than the engine.

On QuestDB the CodeLaser engine took 4m 10s and removed 3,531 declarations. The agent took 90 minutes and removed 991: twenty-one times longer, for under a third of the result. That is its *best* showing, the one attempt that finished because it had nothing left to find.

Put the two sides on one axis, project by project, and the shape of the benchmark is the whole argument. The engine's bar is the shorter one every time, it costs no tokens, and it is the only one that ends clean. The agent is shown at its best on each project; where there were three attempts, the other two were worse.

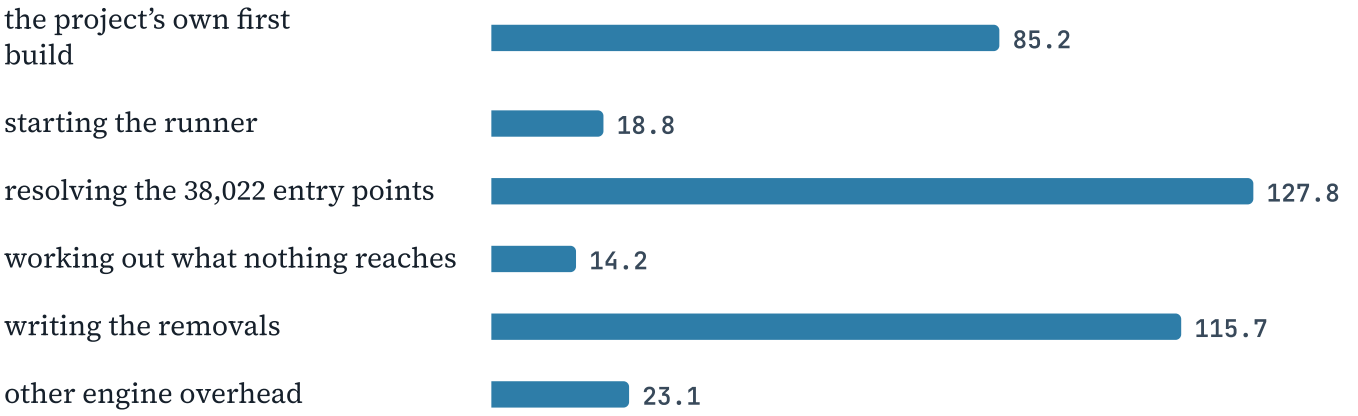
WHAT EACH PROJECT COST EACH SIDE, AND HOW THE RUN ENDED. THE AGENT ROW IS ITS BEST ATTEMPT ON THAT PROJECT

On Trino the agent removed nothing, while the CodeLaser engine finished in 6m 25s and removed 1,478 declarations. Claude spent 196 minutes and 35.6 million tokens across its two sessions. Late in the second it found it had deleted a method that is named only in a string, undid all of its removals, and was applying them again when the two-hour cap stopped it.

Every time in this report includes the project's first build. For both sides the clock starts when that build starts and stops when the tool finishes its own work. What the harness does afterwards to check the result — compiling it again, counting it, running the invariants — is excluded from every figure. Engine times are given to the second; agent times run to tens of minutes and are rounded to the minute.

Most of the engine's 6m 25s on Trino is not analysis. Working out what nothing reaches, across eleven thousand files, takes **14.2 seconds**. The rest is the project's own first build, resolving the entry points, and writing the edits. In the order they happen:

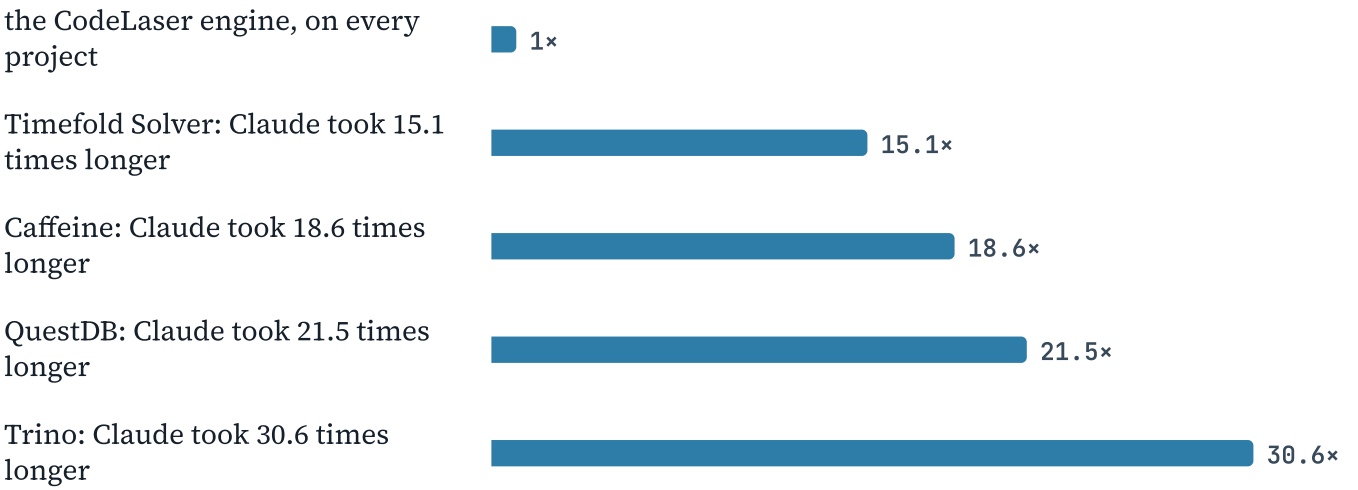
WHERE THE CODELASER ENGINE'S 6M 25S ON TRINO WENT, IN THE ORDER IT WAS SPENT



Deciding what is dead is the cheap part. Resolving every name in the program, and then making the edits, is where the engine spends its time. An agent has to do both as well. It starts from the source files alone, with no resolved representation of the program.

The agent's method is to compile and look at what broke. It removes some code, rebuilds the project, reads the errors, and repeats: between 7 and 25 builds per attempt on Caffeine and Timefold Solver. Without a representation of what reaches what, trial and error is the only method available. It is also the part that grows worst with project size. Trino's build takes about ninety seconds, and a loop that rebuilds after every round pays that every round.

HOW MUCH LONGER CLAUDE TOOK THAN THE CODELASER ENGINE, AT ITS FASTEST ON EACH PROJECT



Tokens show the same pattern. A single attempt consumed between 10.7 and 47.1 million input tokens; the engine consumed none. That is the cost of one attempt at one project. Because one attempt does not pre-

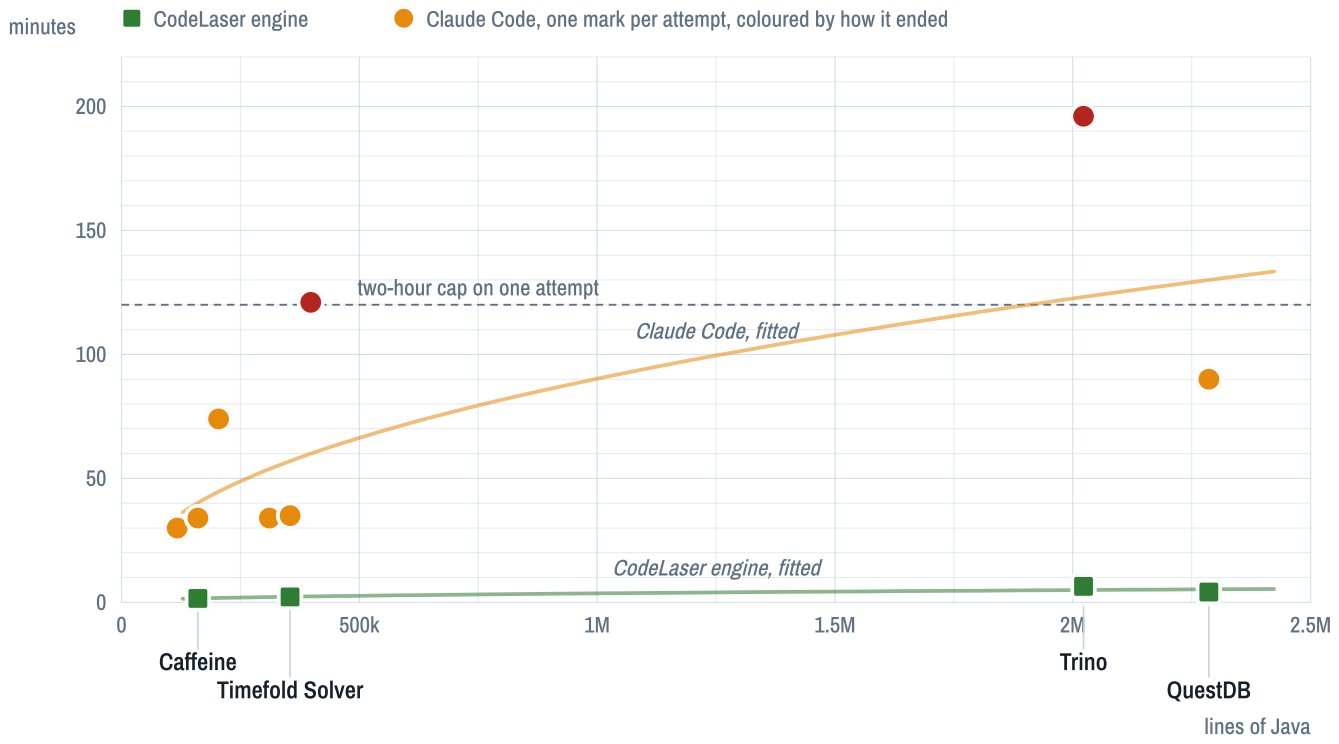
dict the next, getting a usable result can take several, each at that cost.

Three of the eight attempts did not finish. One on Timefold was still working when the cap stopped it, and the code it had produced by then does not compile. The Trino effort ended with nothing removed. In each case the time was spent and nothing usable came of it. Only the QuestDB attempt stopped because a pass found nothing more to remove.

Caffeine is 160,000 lines and Timefold Solver 355,000. Trino and QuestDB are over two million lines each. QuestDB has more lines with half as many files, because its files average 379 lines against Trino's 181.

Neither side's time grows in proportion to size. QuestDB has fourteen times as many lines as Caffeine; it took the engine 2.6 times as long and the agent's best attempt 3.0 times as long. The chart below fits a curve to each side's runs, and both come out close to the square root of project size: the engine's time grows as size to the power 0.45, the agent's as size to the power 0.44. The two sides grow at much the same rate, so the gap between them holds at fifteen to thirty times. The agent's curve is the looser fit of the two, because its attempts scatter so widely: its three on Caffeine took between 30 and 74 minutes. What size does change is whether the agent reaches an answer at all. On Trino its two sessions together ran past the cap for a single attempt, and it removed nothing.

PROJECT SIZE AGAINST TIME, ONE MARK PER RUN, WITH A CURVE FITTED TO EACH SIDE. THE AGENT'S TRINO MARK IS WHERE THE CAP STOPPED IT



OBSERVATION

The agent is not reproducible

Three attempts were run against Caffeine and three against Timefold Solver, from an identical starting point and an identical instruction. They did not produce the same answer, and they did not use the same method. Trino and QuestDB were not repeated. Measuring the spread on projects of that size was not considered: a single effort on Trino already took more than three hours. For those two the spread is unknown, which is not the same as zero.

On Caffeine the three attempts removed 18, 40 and 231 declarations, a factor of thirteen between the smallest and the largest. On Timefold Solver they removed 1,706, 1,478 and 1,317. The engine was run twice on QuestDB, three days apart, and removed the same 3,531 declarations both times. It performs the same computation on the same input, so it gives the same answer.

The variation comes from the agent choosing a different approach each time. Across the six attempts, three distinct strategies appeared:

- **Working by hand.** Read files, judge each case, edit directly. Slowest per declaration and the most conservative: on Caffeine this attempt removed the fewest.
- **Writing an analysis program.** Four of the six attempts wrote their own tooling: a program that reads every source file, builds a map of which declaration refers to which, walks outward from the supplied entry points, and lists what was never reached. Some attempts went further and wrote a second program to perform the deletions, a third to tidy up the leftover imports, and a script to run the whole cycle repeatedly until nothing more was found. One attempt also kept a list of declarations it had learned not to touch, after a previous deletion broke the build.
- **Using the compiler as the oracle.** One attempt wrote no analysis program at all. It removed code that looked unused, rebuilt, read the errors, and put back whatever the compiler complained about — seventeen times.

The agent chose among these methods itself; the instruction did not name one. So a result from one run says little about what the next run will do, and the quality of the outcome depends on which method the agent happened to pick. On Caffeine the same instruction produced both the smallest removal, 18 declarations, and the largest, 231.

OBSERVATION

Errors in the agent's output

Three of the checks run over every result are the subject of this section. The first asks whether anything on the supplied entry-point list was deleted. The second asks whether a surviving class lost every constructor it declared. The third asks whether a field was deleted whose initialising expression calls a method. All three are cheap and specific. **None of them is an audit**: each looks for one known failure pattern and says nothing about anything else.

Those three checks alone found problems in **every agent attempt that removed anything**, and in none of the four engine runs. What follows are examples of places where an attempt is known to have gone wrong.

The loud failure: it does not compile

One attempt produced a project that does not build — the Timefold attempt that ran out of time. This kind of failure is at least obvious: anyone running the build discovers it immediately, and nothing further needs checking.

The other four examples are silent. In each of them the code compiles and the build succeeds. The result is still wrong. The last of the four was not found by any of the three checks — it was found by reading what one attempt had deleted. It is included because it is the clearest case, and because it shows what the checks do not cover.

First example: deleting a test

The QuestDB attempt deleted six declarations that were *on the entry-point list it had been given*. Five are methods the project's own source marks `@SuppressWarnings("unused")` — the author stating in writing that nothing calls them and that they are to stay. The sixth is a test:

```
@Test
public void testBadJsonExtract() throws Exception {
    testBadJsonExtract(ColumnType.BOOLEAN, "false");
    testBadJsonExtract(ColumnType.SHORT, "0");
    testBadJsonExtract(ColumnType.INT, "null");
    ... eleven column types in all ...
}
```

The whole method was removed. The helper it called, which takes the column type and the expected result, was left in place and is now called by nothing. The project compiles, the remaining tests pass, and eleven cases that were being checked no longer are.

This is the failure that matters most in practice. Making everything compile is not the same as being correct, and an approach that leans on the compiler to tell it what is safe will delete anything the compiler cannot see — which is precisely what a test runner, a framework, a configuration file or a reflective lookup reaches.

This was the agent's best attempt: the only one that converged, and one that checked for names written as text. It still deleted a test that was on its entry-point list.

Second example: removing the last constructor of a class

A class that declares only a private constructor cannot be created from outside itself. This is the standard way to write a class that exists only to hold helper functions, and it is a deliberate design decision: the author is preventing instantiation.

If every declared constructor of such a class is deleted, the class is not left without one. Java supplies a constructor automatically, **with the same visibility as the class itself**. A public class whose only constructor was private becomes publicly creatable by anyone. The code compiles and the tests pass. The author's restriction is gone without any error, and the class's public API has changed.

Four of the six attempts did this. The counts below are what the check found; they are not a measure of how wrong each result is overall:

RUN	CLASSES LEFT WITH NO CONSTRUCTOR	OF THOSE, PUBLIC
Caffeine, Claude attempt 2	17	9
Caffeine, Claude attempt 3	17	9
Timefold, Claude attempt 1	51	44
Timefold, Claude attempt 3	5	3
CodeLaser engine, all four projects	0	0

CLASSES LEFT WITH NO DECLARED CONSTRUCTOR, BY RUN

The attempt that used the compiler as its oracle still produced five of these. No amount of rebuilding can catch this kind of error, because the code is valid at every step.

Third example: removing a field whose initialising expression calls a method

A field can be declared together with an expression that is evaluated when the class is loaded:

```
public static final ParameterSpec keySpec = ParameterSpec.builder(kTypeVar, "key").build();
```

Suppose no code anywhere reads `keySpec`. Is the field safe to delete? Deleting it also deletes the call on the right-hand side, and that call is not necessarily a plain computation. A method can write a file, start a thread, register itself somewhere, or change state that the rest of the program depends on. Nothing in the line above settles which it is. Answering requires knowing what `builder` and `build` do, and what every method they call in turn does, including inside libraries whose source is not part of the project being analysed.

The CodeLaser engine does not remove such a field. It reports the field as unused, refuses the edit, and states the reason: the initialising expression contains a method call. This is deliberate caution — the engine declines whenever it cannot show the call is harmless, rather than judging that it probably is.

The agent removed these fields. Its criterion was that nothing reads them; it did not examine what the initialising expression does.

RUN	FIELDS DELETED WHOSE INITIALISING EXPRESSION CALLS A METHOD
Caffeine, Claude attempt 1	2
Caffeine, Claude attempt 2	2
Caffeine, Claude attempt 3	4
Timefold, Claude attempt 1	0
Timefold, Claude attempt 2	128
Timefold, Claude attempt 3	1
CodeLaser engine	0

FIELDS DELETED WHOSE INITIALISING EXPRESSION CALLS A METHOD, BY RUN

The zero on the engine row is not a finding. The engine refuses this edit by design, so it cannot appear in this row.

Deleting such a field can be harmless or it can break the program, and both happened here. In the Caffeine attempts the deleted expressions build immutable values — the call computes a result and does nothing else — so removing the fields changed nothing, and the engine was needlessly cautious. Those are real removals the engine left on the table.

The 128 fields removed in the second Timefold attempt are the other case. Most of them held a test-harness object that the testing framework collects and uses to assemble the application the tests are written to exercise. There the expression does not merely compute a value; it establishes the environment the surrounding tests depend on. Removing the field compiles cleanly, and the tests it configured no longer have the application they were written against.

The agent was right in the first set of cases and wrong in the second, and in neither did it establish which it was in. That is the substance of the disagreement. Showing that a method does nothing beyond returning a value means following every call it makes, and every call those make, through code that may not be available to read. It is a hard property to prove and an easy one to get wrong, and the cost of the two errors is not symmetric: refusing these edits gives up some genuine removals, four of them here. Making them without checking gains those four and produces the 128 as well. On this evidence the conservative choice is the right default.

Fourth example: emptying an enum whose constants a configuration file selects by name

Caffeine includes a cache simulator. Which strategies the simulator runs is not stated in the Java source. It is stated as text in a configuration file, `simulator/src/main/resources/reference.conf`:

```
strategy = [
  simple,
  correlation,
  trust-region-ewma,
  ...
]
```

Those strings are converted into Java values when the simulator starts. Each one is upper-cased, its hyphens become underscores, and the result is looked up by name:

```
public ImmutableSet<HillClimberType> strategy() {
  return config().getStringList("hill-climber-window-tiny-lfu.strategy").stream()
      .map(strategy → strategy.replace('-', '_').toUpperCase(US))
      .map(HillClimberType::valueOf)
      .collect(toImmutableEnumSet());
}
```

So `trust-region-ewma` in the file becomes the name `TRUST_REGION_EWMA`, and `HillClimberType.valueOf` returns the option carrying that name. `HillClimberType` is a Java enum — a fixed list of named options — and each option holds a small function that builds the class implementing that strategy:

```
public enum HillClimberType {
  SIMPLE((_, config) → new SimpleClimber(config)),
  CORRELATION((_, config) → new CorrelationClimber(config)),
  TRUST_REGION_EWMA((_, config) → new TrustRegionEwmaClimber(config)),
  … six more, ending
  INDICATOR(IndicatorClimber::new);
```

The third Caffeine attempt reduced it to this:

```
public enum HillClimberType {
  ;
```

All ten options removed, and the ten strategy classes they build deleted as files.

`valueOf` raises an error when no option carries the name it was given. With no options left, every name in that configuration file fails, and the policy cannot be built at all. The simulator's hill-climbing feature is gone. The project still compiles and the build still passes, because no Java code anywhere writes those names. The only place they are written is the configuration file, which the compiler never reads.

Why the engine leaves this alone. By reference count these options are unused, and the engine sees that. It keeps them because it has a rule for this Java mechanism: when it sees a call to `valueOf` on an enum, it

marks *every* option in that enum as used. The reason is that `valueOf` is handed a name as a piece of text, and which text it will be handed is not decided until the program runs. The engine cannot tell which option is meant, so it keeps them all. The same rule covers `values()`, which hands out the whole list at once. Here the call is written as `HillClimberType::valueOf`, a reference to the method rather than a call spelled out in full; the rule covers that form too.

Keeping the options alive then keeps the ten classes alive, because each option's definition contains `new TrustRegionEwmaClimber(config)` and so on. One rule about one language feature protects the whole group.

The agent counted references and found none, which is correct: no Java code refers to these options. Counting more carefully would not help. What is needed is to know in advance that this construct is resolved by name at run time, and that deleting its members is not safe whatever the count says.

> WHY "IT COMPILES" IS NOT A SUFFICIENT CHECK
>
> All four silent examples above pass a full build. If the only verification is that the project still compiles — which is the natural check to ask for, and what the agent was asked for — every one of them would be reported as a success. Detecting them required knowing in advance what to look for. Three such checks were written, and they found problems in every one of the seven attempts that removed anything; the fourth example was found by reading the output, not by any check.

OBSERVATION

The agent's real task is to build a dead-code engine under a clock

The instruction was to remove dead code. The work that filled the time was writing a program to decide what dead code is. The task cannot be done without such a program. The engine is one; the agent had to write its own.

On Trino the agent wrote nineteen files before removing anything: a reader for compiled Java classes, two scanners built on the compiler's own parser, an index of every library on the classpath, a reachability pass, two planners, a text editor that removes a declaration along with the comment above it, an import fixer, a snapshot-and-rollback, a repair loop that reads compile errors, and an outer loop that repeats the cycle until a pass finds nothing. More than an hour of the effort went into building that, and another 47 minutes into repairing it, before a single declaration was removed.

Four repairs caused by working from compiled classes

On Trino the agent worked from compiled classes rather than source, because in a compiled class every call already names the method it reaches. But compilation also erases or relocates information. Each gap showed up as a group of declarations the agent wrongly believed dead, and it had to repair its analysis for each of these four:

- **Constants the compiler copies.** A `static final` number or string is copied into every class that uses it, so the class that declares it looks unreferenced.
- **Lambdas and method references.** These compile to an `invokedynamic` instruction. The method that does the work is named only as an argument to a bootstrap method, never as a call, so a reader that follows calls sees it as uncalled.
- **Generic type arguments.** They are erased from the compiled instructions and survive only in the class file's `Signature` and `LocalVariableTypeTable` attributes, so a type used only as a type argument looks unreferenced.
- **Sealed types.** A sealed class records its permitted subclasses in a separate attribute, which the agent's class reader ignored until it was repaired.

None of these is a problem for the engine, because it reads source with every name resolved to what it denotes. Nothing was erased, so there is nothing to recover. The agent did not choose compiled output carelessly. The alternative is a Java parser that resolves names, and that is not a two-hour job. That trade is the clearest statement of what the CodeLaser engine is: that parser, plus the accumulated knowledge of when reachability is decided by something other than a reference in the code.

On Caffeine and Timefold Solver the agent produced an answer in fifteen to fifty-four times the engine's time, and some of those answers contain damage a build does not reveal. On Trino it produced no answer at all, having spent its time writing and repairing a program that does what the engine already does. On QuestDB it worked the way a careful engineer would: it built its analysis on the compiler's own machinery, repeated until a pass found nothing, and checked its own work. It still removed under a third of what the engine removed. On the way it deleted a test that was on its entry-point list.

The rules the agent never reached concern ordinary Java: configuration files, field initialisers, constructors. Each requires knowing something the code does not state at the point of removal: that a name in a configuration file selects an enum constant, that an initialising expression may do work beyond computing a value, that removing a class's last constructor makes it instantiable. The engine carries these rules already. An agent that calls the engine does not have to rediscover them.

CONCLUSION

Codebase-wide changes need an engine behind the agent

An AI coding agent is good at writing new code within a bounded scope: a function, a feature, a test, a fix. These experiments show that too. Most attempts wrote their own analysis programs without being asked to — readers for compiled classes, reachability passes, planners, repair loops.

Removing dead code is a different kind of task. It changes the whole codebase at once, every decision in it depends on the rest of the program, and it has to be right in thousands of places. On that task the experiments show four problems:

- **It is slow and costly.** Claude took 15 to 54 times as long as the engine on Caffeine, Timefold Solver and QuestDB, and consumed 10.7 to 47.1 million input tokens per attempt. See The agent costs far more.
- **It does not reach an answer on the largest codebase.** On Trino, two million lines, it spent 196 minutes and 35.6 million tokens and removed nothing.
- **It gives a different answer each time.** Three attempts on Caffeine, from the same starting point and the same instruction, removed 18, 40 and 231 declarations. See The agent is not reproducible.
- **Its mistakes pass the build.** It deleted a test that was on its own entry-point list, made classes publicly instantiable, deleted fields whose initialiser set up the tests, and emptied an enum whose constants a configuration file selects by name. Every one of these compiles. See Errors in the agent's output.

These problems do not come from a careless agent. On QuestDB it worked the way a careful engineer would, and still removed under a third of what the engine removed. They come from asking it to build, within reasonable time, what a codebase-wide change depends on: every name in the program resolved, and a rule for each way Java code stays alive without being referenced. See The agent's real task is to build a dead-code engine under a clock.

That is what the CodeLaser engine provides. It also gives the same answer every time: run twice on QuestDB, it removed the same 3,531 declarations.

Used together, each does the part it is suited to. An agent that calls the engine leaves the code mechanics to it — resolving names, working out what nothing reaches, making the edits — and spends its own time and tokens on deciding what to tackle, judging the findings and explaining the result.

The complete record of every run, from diffs and build logs to session transcripts, is available to anyone evaluating CodeLaser.