

Dead-code removal on four open-source Java projects: the CodeLaser engine against Claude Code

The CodeLaser engine and Claude Code were given the same four open-source Java projects, the same list 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 time, cost, and whether the project still worked afterwards.

ISSUED

2026-09-28

CONTACT

tom.tourwe@codelaser.io

TABLE OF CONTENTS

What the experiments show

What was compared

How the CodeLaser engine removes dead code

Every agent attempt built its own dead-code analyser

What each run removed and what it cost

Where the two sides disagree

The agent costs far more time and tokens

The agent is not reproducible

Errors in the agent's output

Codebase-wide changes need an engine behind the agent

A. How it was measured

B. Why each root is on the list

C. Full figures for every run

IN SHORT

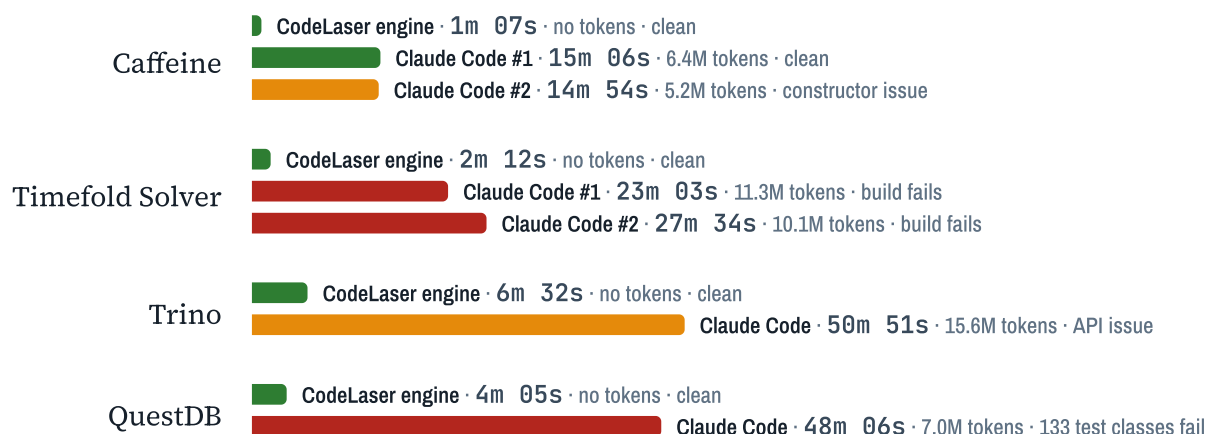
What the experiments show

On four open-source Java projects, the CodeLaser engine removed dead code 8 to 14 times faster than Claude Code, and every result was clean. Claude Code's results compiled, but in five of its six attempts they broke a build, a test suite or a public API.

- **The engine** removed everything unreachable in 1 to 7 minutes per project. Every project still builds, the checks found no fault, the test suites were left intact, and no test that was run regressed.
- **The agent** took 8 to 14 times as long and used 5.2 to 15.6 million input tokens per attempt.
- **The agent's extra removals broke things the compiler does not check.** It disabled every HTTP write in QuestDB: 133 test classes fail. Timefold Solver's own build rejected both attempts, so 855 of its 6,269 tests never ran. On Trino, 885 public classes can now be created from outside. The engine did none of this.
- **The agent's behaviour is non-deterministic.** As an example, two attempts on Caffeine with the same input removed 6 and 21 declarations and agreed on only 5.

Every run is below, on one time axis. Green is a clean result, amber a problem the checks found, red a broken build or failing tests.

EVERY RUN ON ONE TIME AXIS AND HOW ITS RESULT HELD UP



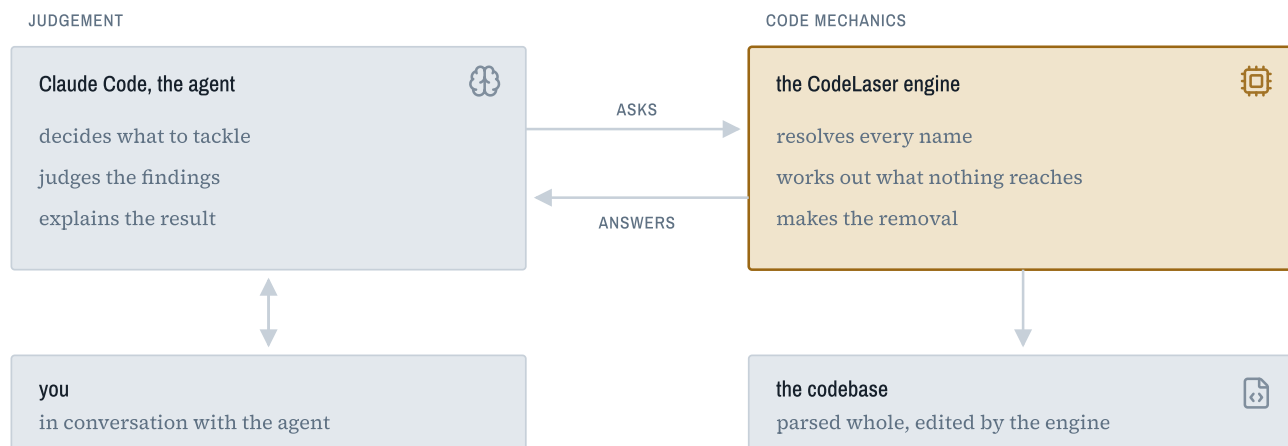
Every result in this document was measured by a predefined test harness, independently of both sides. It parsed each project's source before and after every run, with the same parser for both sides, and compared the two. Neither the engine's nor the agent's own report of what it had done was used.

Removing dead code is an important and non-trivial task in fighting technical debt. The task is complex and sometimes needs contextual judgement, so having an agent in the loop is a good idea. The finding is about the division of labour: the agent should drive a tool that works out what is dead, not write that tool itself. Handed this task, an agent starts by building a dead-code analysis of its own. All six attempts here did exactly that, unprompted. The analysis has to resolve every name in a Java program and know every

way a declaration stays alive without being referenced. That is a very complex program. An agent cannot be expected to write it on the spot, in answer to a refactoring task such as dead-code removal.

In this benchmark the engine ran on its own, with no agent and no tokens. It can also be handed to an agent. The division of work then becomes more natural. The engine does the code mechanics: resolving every name, working out what nothing reaches, and making the removal. The agent decides what to tackle, judges the findings and explains the result. Its tokens then go on reasoning and on talking to you.

WHO DOES WHAT WHEN THE AGENT DRIVES THE ENGINE



THE SETUP

What was compared

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 CodeLaser engine** runs its dead-code operation on a fully resolved model of the project: it works out what refers to what, and deletes what cannot be reached. **Claude Code** is the coding agent a developer would use interactively, here working on its own with full access to the project and a shell. It was Claude Code 2.1.283 on Claude Opus 5.5 in all six attempts. Where this document says **the agent**, it means Claude Code.

Claude Code was used before the runs to propose the list of roots, described below, that both approaches were given as input. The engine itself ran on its own, without Claude.

Both sides were run and measured by the same harness, which never asks either side what it did. It counts what each run removed, runs checks over the result, and then runs the project’s own tests. Appendix A describes it.

PROJECT	JAVA FILES	LINES	MAVEN MODULES	ROOTS SUPPLIED
Caffeine	704	160,501	1	8,069
Timefold Solver	3,652	354,502	60	22,632
Trino	11,154	2,022,701	110	54,341
QuestDB	6,026	2,286,193	5	48,410

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 110 separately built Maven modules. QuestDB is a time-series database. Each project was pinned to one commit and reset before every run.

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 `main` method started from the command line. Any tool that ignores this will delete working code.

So both approaches were handed the same list of **roots**: every declaration that something outside the code can reach. Whatever no root reaches is dead. Roots are often called entry points. This report says roots, because *entry point* suggests a published API, and only 15,163 of these declarations are part of one. About half the list is tests reaching into the code. The rest is what makes dead code hard: a name in a string, a resource file, a reflection configuration, a service loader. Across the four projects the list holds 133,452 roots. Appendix B gives the reasons.

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"**. Every number in this report comes from a run against the list as it stood on the date of that run. The engine was given the project and this list, and nothing else.

The agent had a written instruction and two hours per attempt

The agent was given a written instruction. It said to remove as much dead code as possible, and to repeat until nothing more is found, because removing one declaration is what makes the next one dead. The project had to build at the end, and the agent could rebuild as often as it liked. It was told explicitly that it could change *any* source file, because deleting a type often requires editing the places that mention it. It was free to write its own programs to do the job, in any language. 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. **No attempt reached the cap.** All six ended when the agent said it was finished.

Because the agent does not give the same answer twice, **two attempts** were run on Caffeine and on Timefold Solver, so that the spread could be seen. Trino and QuestDB are too expensive to repeat that way and received **one attempt** each.

THE CODELASER ENGINE

How the CodeLaser engine removes dead code

Finding dead code has a cheap approximation, while fully removing it requires a high level of precision. Counting references and walking outward from the roots will find most of the obvious dead code, which is why all six agent attempts wrote a program to do exactly that. Deleting declarations is where things get more difficult: every place that referred to one has to be repaired, methods that only it called are now dead too, and imports, overrides and signatures have to be corrected. Each deletion can make another declaration dead, so the work repeats until a pass finds nothing.

Accurate detection based on a fully resolved model. Before it decides anything, the engine builds a model of the whole project in which every name is resolved to what it denotes: which declaration a call reaches, which method overrides which, which type a name means. The model is built primarily from the source code, and uses compiled code mainly for references into libraries. All of the engine's reasoning happens on this model, never on bytecode. Reachability is computed over it, outward from the roots. This is what separates it from counting references. The first example under *Errors in the agent's output* is a case where the difference disables a database's write path.

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. The engine carries a rule for each such mechanism. One of them: a call to `valueOf` or `values()` on an enum marks **every** constant in that enum as used, because which one will be asked for is not decided until the program runs.

The engine declines what it cannot show is safe. Where it cannot show that a removal is safe, it does not make it, and says why. On QuestDB, for example, it removed eleven plain constants from one class and kept the field beside them whose initialiser registers a logger, because deleting that field would change what the program does.

The engine does not use the compiler as a source of feedback. It decides what to remove without building the project to test its guesses. The build afterwards only verifies the result.

THE AGENT

Every agent attempt built its own dead-code analyser

This section is reconstructed from the six session transcripts and the programs each attempt left behind. All six took the same route, unprompted. Each wrote a reachability analysis of its own, from scratch, and ran it in a loop:

1. **Build the project, then read the compiled class files.** Three attempts used the ASM library, three the JDK's own class-file API. In a compiled class every call already names what it reaches, so this is the quickest place to start.
2. **Walk outward from the roots.** Each matched the list of roots to the compiled code, followed calls, field reads and object creation, and reported whatever nothing reached.
3. **Cut it out of the source.** Four attempts wrote code that uses a Java parser to find each declaration in the source and delete it. The two Caffeine attempts used short Python scripts.
4. **Rebuild. On failure, change a rule.** When a removal broke the build, no attempt repaired the calling code. It went back to its last good state, changed a rule in its analyser, and ran again.
5. **Stop when a pass finds nothing.** All six said they stopped because they had run out of dead code, not out of time.

Most of the time went on discovering aspects of the code that are not available from class files alone. For example, the compiler copies constants into the classes that use them, drops code behind `if (debug)`, and keeps some annotations only in the source. The attempts ran into these gaps one at a time and added a rule for each. None handled names in strings or configuration files; for those, all six relied on the list of roots. The engine's model is built primarily from the source, so what the compiler copies away or leaves out of the class files is still in it. All its reasoning happens on that model, never on bytecode.

Deleting the constructors was a decision the agent made. All six attempts deleted private constructors of helper classes. The first Caffeine attempt put them back, because the project's build tooling warned about it. The other five kept the deletions. Three of them, both Timefold Solver attempts and Trino, described the consequence only in their closing message: the classes can now be created from outside. They called it "a judgement call" and "easy to revert", and left it in place.

The real task was building a dead-code engine under a clock. The instruction was to remove dead code. What filled the time was writing a program to decide what dead code is, and every attempt wrote its own, from scratch. The analysers ran from 476 to 1,747 lines; Appendix C lists them. Reading class files was a reasonable choice, because the alternative is a Java parser that resolves every name, which is more than an afternoon's work. But it meant rediscovering, one rule at a time, what ordinary Java already implies: an override of a default method, a private constructor, an enum constant selected by name at run time, a field

whose initialiser does work. The attempts reached some of these rules and missed others. The engine carries them all. An agent that calls the engine does not have to rediscover them.

RESULTS

What each run removed and what it cost

This section compares the two sides project by project: how long each run took, what it spent, and how its result held up. The chart in the summary shows every run at a glance. The full figures for every run are in Appendix C.

Caffeine

Input data: 704 files, 160,501 lines, 8,069 roots supplied to both sides.

- **The engine** removed 8 declarations in 1m 07s. The result builds and the checks found nothing.
- **The agent** needed about 15 minutes per attempt, 13 times as long. Its two attempts removed 6 and 21 declarations and agreed on only 5. The second left 14 classes without a constructor, 7 of them public.
- **No tests were run on Caffeine.** Its suite multiplies every test by every cache configuration, and its own CI splits the result into 40 shards of up to an hour each. That does not fit this benchmark, so on Caffeine the build and the checks are the only evidence.

Timefold Solver

Input data: 3,652 files, 354,502 lines, 22,632 roots supplied to both sides.

- **The engine** removed 311 declarations in 2m 12s. The result builds, the checks found nothing, and Timefold Solver's full test suite ran on it: 6,269 tests in one run, the same as on the untouched project, with no failure.
- **The agent** needed 23 and 28 minutes, 10 to 12 times as long, and removed about 400 declarations in each attempt. Both attempts made Timefold Solver's own build fail, by deleting the private constructors of two public classes (the second example under *Errors in the agent's output*).
- **The agent's clean test record is misleading.** No test failed on its results because 855 of the 6,269 tests never ran. The failing build stopped Maven before it reached the modules that hold them. Those tests were not deleted. They never saw the agent's changes, so nothing is known about what they would have found.

Trino

Input data: 11,154 files, 2,022,701 lines, 110 Maven modules, 54,341 roots supplied to both sides.

- **The engine** removed 835 declarations in 6m 32s. The result builds, the checks found nothing, and no test failed because of the removal.

- **The agent** needed 51 minutes, 8 times as long, and removed 2,070 declarations, two and a half times as many. Most of the extra are constructors. It left 957 classes without a declared constructor, 885 of them public, so those 885 can now be created from outside. That is a change to Trino's public API (the third example under *Errors in the agent's output*).
- **Trino's tests did not catch it.** No test failed on either result. Trino's full suite takes hours, so each result was tested on the modules its run had changed (Appendix A).

QuestDB

Input data: 6,026 files, 2,286,193 lines, 48,410 roots supplied to both sides.

- **The engine** removed 1,547 declarations in 4m 05s. The result builds, the checks found nothing, and QuestDB's test suite ran on it, 41,587 tests, with no failure caused by the removal.
- **The agent** needed 48 minutes, 12 times as long, and removed 2,138 declarations. Its result compiles, but 133 of QuestDB's test classes fail on it: the agent disabled every HTTP write path (the first example under *Errors in the agent's output*). It also left 115 classes without a constructor, 102 of them public.

How to read these results

What a clean result means. The project builds, the checks found no fault, and no test failed because of the removal. Whether a removal is *right* is not something this benchmark decides. A run could have deleted something it should have kept in a way nobody has looked for. Neither side's output has been audited line by line, including the engine's.

The number of declarations removed is not a score. A run that removes more may have removed things it should have kept, which is what the next two sections are about.

RESULTS

Where the two sides disagree

Both sides were measured against the same parse of the same code, so their removals can be compared one declaration at a time. Nearly everything the engine removed, the agent removed too. The agent also removed a great deal more. Those extra removals are where the damage is:

PROJECT	ENGINE REMOVED	AGENT REMOVED	BOTH REMOVED	AGENT ALONE	ENGINE ALONE
Timefold Solver, attempt 1	311	405	308	97	3
Timefold Solver, attempt 2	311	399	306	93	5
QuestDB	1,547	2,138	1,395	743	152
Trino	835	2,070	819	1,251	16

REMOVALS THE TWO SIDES AGREE ON AND WHAT EACH REMOVED ALONE

On Trino, 983 of the 1,251 declarations the agent removed and the engine did not are constructors. Removing them changes the public API of the classes that lose them, which is the third example below. Appendix C breaks the extra removals down by kind.

OBSERVATION

The agent costs far more time and tokens

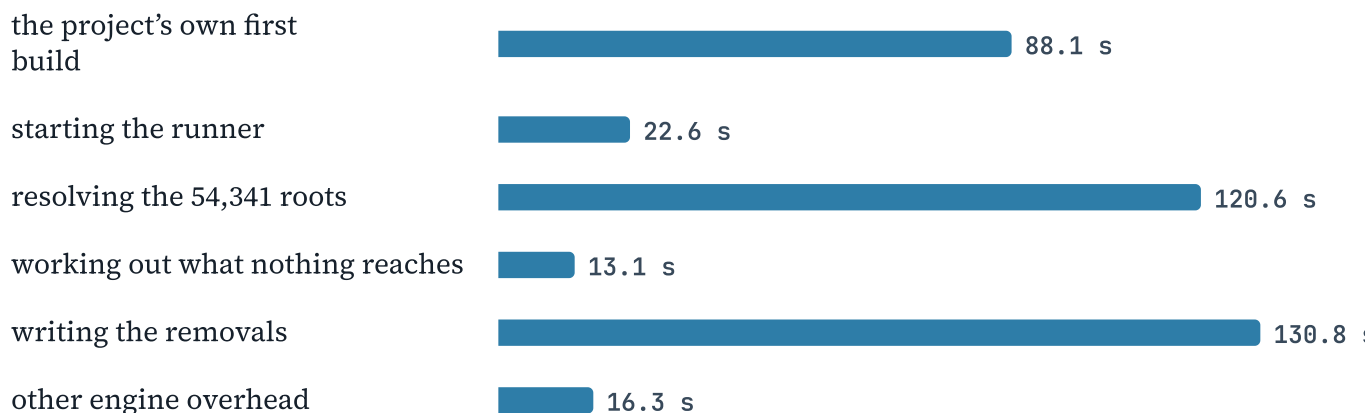
PROJECT	THE CODELASER ENGINE	CLAUDE CODE, PER ATTEMPT	HOW MUCH LONGER
Caffeine	1m 07s	14m 54s – 15m 06s	13.3× – 13.5×
Timefold Solver	2m 12s	23m 03s – 27m 34s	10.5× – 12.5×
QuestDB	4m 05s	48m 06s	11.8×
Trino	6m 32s	50m 51s	7.8×

WHAT EACH PROJECT COST EACH SIDE

Every time above is the tool’s own work. The clock starts when the project’s first build starts and stops when the tool finishes. Checking the result afterwards, by compiling it again, counting it, and running the checks and the test suite, is excluded on both sides.

Most of the engine's 6m 32s on Trino is not analysis. Working out what nothing reaches, across eleven thousand files, takes **13.1 seconds**:

WHERE THE ENGINE'S 6M 32S ON TRINO WENT



Deciding what is dead is the cheap part. Resolving every name in the program and then making the edits is where the time goes. An agent has to do both as well, and starts with nothing resolved.

Tokens show the same pattern. A single attempt consumed between 5.2 and 15.6 million input tokens and between 64,000 and 150,000 output tokens. The engine consumed none. Because one attempt does not predict the next, getting a usable result can take several attempts, each at that cost.

OBSERVATION

The agent is not reproducible

Two attempts were run on Caffeine and two on Timefold Solver, from an identical starting point and an identical instruction.

On Caffeine they did not agree. One removed 6 declarations, the other 21. They agree on 5. Both used the same method: an analyser over the compiled class files, built with the ASM library. Both deleted 15 private constructors. The first attempt then put them back, because the project's build tooling warns about a helper class without a private constructor. The second kept the deletions. That one judgement call accounts for most of the difference between the two answers.

On Timefold Solver they very nearly did agree: 405 and 399 declarations, of which 398 are the same. Both wrote an analysis program that reads compiled class files. Both arrived at much the same answer, including the same mistake on the same two classes, which is the subject of the second example below.

So the agent's answer depends on judgement calls it makes along the way, even when its method is the same. The engine performs the same computation on the same input and gives the same answer.

OBSERVATION

Errors in the agent’s output

Every one of these compiles, so no amount of rebuilding would find them. The harness’s checks found the second and third. QuestDB’s test suite found the first. Timefold Solver’s own build found the second as well.

First example: QuestDB can no longer accept an HTTP POST or PUT

QuestDB decides whether an HTTP handler accepts a given request method by asking the handler. The question is declared once, with a default answer:

```
// io/questdb/cutlass/http/HttpRequestProcessor.java
default short getSupportedRequestTypes() {
    return METHOD_GET;
}
```

Three interfaces override it, because handlers of those kinds accept more than a GET :

```
// io/questdb/cutlass/http/HttpPostPutProcessor.java
public interface HttpPostPutProcessor extends HttpRequestProcessor {
    @Override
    default short getSupportedRequestTypes() {
        return METHOD_POST | METHOD_PUT | NON_MULTIPART_REQUEST;
    }
    ...
}
```

The agent deleted all three:

THE OVERRIDE IT DELETED	WHAT THAT OVERRIDE ANSWERED
HttpPostPutProcessor.getSupportedRequestTypes()	METHOD_POST METHOD_PUT NON_MULTIPART_REQUEST
HttpMultipartContentProcessor.getSupportedRequestTypes() ()	METHOD_POST METHOD_PUT MULTIPART_REQUEST
processors/RejectProcessor.getSupportedRequestTypes()	ALL INVALID

THE THREE OVERRIDES THE AGENT DELETED FROM QUESTDB

The third belongs to the handler QuestDB uses to **produce** a rejection. It accepts every method so that it can answer any request no other handler will take. The code still compiles, because deleting an override is always legal: calls fall through to the declaration above, which answers METHOD_GET .

QuestDB checks that answer on every request, and rejects any method a handler does not list.

So every POST and PUT handler in QuestDB now declares that it accepts `GET` only, and the server answers `405 Method not supported`. CSV import, the settings endpoint and the line-protocol write endpoint all stop working: a time-series database that can no longer be written to over HTTP. QuestDB's own tests for all three now fail with `405`.

Why the agent removed them. Its analyser does follow calls through interfaces. But where several interfaces supply a default for the same method, it keeps whichever one it meets last, not the most specific one. So `processor.getSupportedRequestTypes()` resolved to the base declaration, and the three overrides looked unreachable. **Why the engine keeps them.** It resolves the call to every declaration it can reach at run time, which includes all three overrides. The engine's run did not touch any of the three files.

Second example: Timefold Solver's own build rejects a change to its published API

Both Timefold Solver attempts deleted the private constructors of `ConstraintCollectors` and `Joiners`, two public classes in `ai.timefold.solver.core.api`:

```
public final class ConstraintCollectors {
    ...
    private ConstraintCollectors() {
    }
}

public final class Joiners {
    ...
    private Joiners() {
    }
}
```

A class that declares only a private constructor cannot be created from outside itself. It is the standard way to write a class of helper functions. The author chose it deliberately. Delete the constructor and Java supplies one automatically, **with the same visibility as the class**. So `new ConstraintCollectors()` compiles where it did not before, and a class that was never meant to be instantiated is now part of the API anybody can call.

Timefold Solver checks for exactly this, in its own build, with Revapi:

```
[ERROR] Failed to execute goal org.revapi:revapi-maven-plugin:0.15.1:check (check)
on project timefold-solver-core: The following API problems caused the build to fail:
[ERROR] java.method.visibilityIncreased: method void
ai.timefold.solver.core.api.score.stream.ConstraintCollectors::<init>(): visibility increased
[ERROR] java.method.visibilityIncreased: method void
ai.timefold.solver.core.api.score.stream.Joiners::<init>(): visibility increased
```

Both attempts did this to the same two classes. It is also the reason 855 of the project's tests could not be run on either of the agent's results. The engine's run left every declared constructor in place.

Third example: 957 classes on Trino left with no declared constructor

The same kind of removal, at scale. On Trino it was a rule the agent wrote: delete the private constructor of every helper class, applied in a single round. Each such class can then be created from outside, the way `new RemovalListeners()` now compiles in Caffeine where it did not before.

Sometimes the author has written down that this matters. Caffeine's `References` carries `@SuppressWarnings("PMD.MissingStaticMethodInNonInstantiatableClass")` on the class, three lines above the constructor the agent deleted: the author stating in the source that the class is deliberately not instantiable.

RUN	CLASSES LEFT WITH NO CONSTRUCTOR	OF THOSE, PUBLIC
Caffeine, attempt 1	0	0
Caffeine, attempt 2	14	7
Timefold Solver, attempt 1	48	43
Timefold Solver, attempt 2	48	43
QuestDB	115	102
Trino	957	885
the CodeLaser engine, all four projects	0	0

CLASSES LEFT WITH NO DECLARED CONSTRUCTOR AT ALL

On Trino one run made 885 public classes creatable from outside. The agent's closing message calls this "a judgement call". Every one of them compiles, and Trino's tests found no failure caused by the removal. Timefold Solver noticed only because it runs an API-compatibility check of its own.

WHY "IT COMPILES" IS NOT A SUFFICIENT CHECK

All three examples 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 would be reported as a success. Finding them needed knowing in advance what to look for. The harness runs four checks that can fail a run, listed in Appendix A. They found something in five of the six agent attempts and in none of the four engine runs.

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. All six attempts wrote their own analysis programs, unprompted, each of them several hundred lines of working bytecode analysis.

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 three problems:

- **It is slow and costly.** Claude took 8 to 14 times as long as the engine on all four projects, and consumed 5.2 to 15.6 million input tokens per attempt. The engine consumes none. See [The agent costs far more time and tokens](#).
- **It gives a different answer each time.** Two attempts on Caffeine, from the same starting point and the same instruction, removed 6 and 21 declarations, agreed on 5, and split on a single judgement call. See [The agent is not reproducible](#).
- **Its mistakes pass the build.** In five of six attempts it broke a build, a test suite or a public API, and every one of those results compiles. See [Errors in the agent's output](#).

The agent was not careless. Every attempt ended when it judged itself finished, well inside the two-hour cap. On Timefold Solver the two independent attempts agreed on 398 of about 400 declarations. The problems come from asking it to build, in 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.

In other words, this experiment shows what an agent is not good at, despite being an excellent coder: tasks that require a spotless view of how pieces of code connect and behave, and the ability to make fundamental changes to the code without introducing any errors.

The CodeLaser engine provides that. It gives the same answer every time. 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. It 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.

APPENDIX

A. How it was measured

The projects. 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 file counts are the harness's own. The line counts are `.java` lines outside `target/` and `build/`, counted separately, because the harness does not record them.

The agent's environment. Claude Code ran in a container with network access and a shell, holding a copy of the project and nothing else of the harness. No restriction was placed on what tools it could use: it could write and run any program it liked, in any language on the image.

The harness. 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 engine or the agent
- counts the declarations in the project before and afterwards, with the same parser for both sides, keyed by file, enclosing type and signature
- runs its checks over the result (below)
- only then runs the project's own test suite, so that each side's time is its own work
- writes everything it saw to one directory for that run

The checks. Four can fail a run: a surviving class that lost every constructor it declared; an enum that lost a constant while surviving code still reads constant positions; a class emptied of every member that nothing names any more; and an edit to a file the project declares generated-only. Two more are counts recorded beside them, not faults. One of those counts is classes emptied of their members that surviving code still names, as a type in a signature, a supertype or a `X.class`. Such a class could not have been deleted, so emptying it is the correct outcome. Runs on both sides have them: 18 in the Trino agent run, 2 in the Timefold Solver engine run, 1 in the Trino engine run.

Parameters. The counter records each method parameter as a declaration of its own, so a three-parameter method that goes counts as four. Nearly every removed parameter belongs to a method or constructor that was itself removed. Without parameters, the QuestDB totals are 949 for the engine and 1,322 for the agent.

Files the parser cannot read. The counter parses with JavaParser, which cannot parse 901 of Trino's files, 146 of QuestDB's and 13 of Caffeine's, and none of Timefold Solver's. No declaration in those files is counted for either side. On Trino, both sides edited three of those files into a form JavaParser can read. That adds declarations to the count after the run and takes none away, so no removal count is affected.

Tokens. Input tokens are the whole context the agent read on each step, nearly all of it re-read from cache. The engine consumes none, because it is a compiled program. It is not free of AI involvement: the script that drives it and the list of roots were produced with Claude Code beforehand. That preparation is paid once, not on every run.

Tests. Every test count in this report comes from one run of the project's tests on a run's final result. Tests are counted as JUnit reports them: a parameterised test counts once per parameter set, and tests a project itself marks as skipped are included (in the untouched projects, 7 on Timefold Solver and 470 on QuestDB). On Timefold Solver that run is the whole suite, 6,269 tests in 608 test classes, the same as on the untouched project. On QuestDB it is the whole suite of 42,044 tests in 1,915 classes, except `IODispatcherTest` and `PGJobContextTest`, 457 tests between them, which stall under the harness. That leaves 41,587 tests on both sides. On Trino the full suite was not run. The harness runs every test class of each Maven module in which the run changed at least one file. The agent edited 1,402 source files to the engine's 367, and removed something from 1,211 classes to the engine's 273. So 2,301 test classes were selected for the agent and 2,026 for the engine. On Caffeine no suite is run, for the reason given with its results.

APPENDIX

B. Why each root is on the list

ON THE LIST BECAUSE	COUNT
it is a test method, which the test runner finds and calls	60,591
a framework annotation on it means the framework calls it	17,718
it is part of the project's published API, which code outside the project calls	15,163
it is a member of a type that carries a framework annotation	13,860
a test writes its class literal, for example <code>Foo.class</code>	6,947
a reflective lookup in the source names it	3,814
a test runner finds it through a supertype it extends	3,530
its own author wrote that it is deliberately uncalled, for example <code>@SuppressWarnings("unused")</code>	2,011
a string in the source spells its name	1,604
a configuration property is bound to it by name	1,379
it is a subclass of an annotated type	1,367
a framework selects this enum constant by name	900
something outside the analysed code uses it	897
it inherits a test method from a superclass	885
a framework creates the class through its no-argument constructor	593
it is a <code>main</code> method	448
a framework reads or writes the field through this accessor	411
a resource file names it	391
a GraalVM reflection configuration file names it	240
a service loader finds it	193
serialization reaches it	161
a JUnit argument provider supplies it	145

ON THE LIST BECAUSE	COUNT
a template file names it	80
it is declared in a package the project publishes as a library	75
a framework converts a string into it	24
a framework reads it as a bean property	20
a JUnit 3 suite names it	5

WHY EACH OF THE 133,452 ROOTS ACROSS THE FOUR PROJECTS IS ON THE LIST

60,591 of these are test methods. With every test-related reason included (a class literal in a test, a supertype a runner finds, an inherited test method, an argument provider, a JUnit 3 suite) it is 72,103, or 54 per cent.

APPENDIX

C. Full figures for every run

Counts are of **declarations**: types, methods, constructors, fields, enum constants and method parameters, measured by parsing the project before and after each run. A removed method's parameters are counted with it, so the totals are larger than the number of types, methods and fields removed. Both sides are counted the same way. Appendix A gives the totals without parameters.

Caffeine

WHAT WAS MEASURED	THE CODELASER ENGINE	CLAUDE ATTEMPT 1	CLAUDE ATTEMPT 2
types	0	0	0
methods	2	3	3
constructors	0	0	15
fields	1	1	1
enum constants	4	1	0
parameters	1	1	2
declarations removed	8	6	21
lines removed	21	20	59
files changed	3	4	18
time	1m 07s	15m 06s	14m 54s
input tokens	none	6.4M	5.2M
output tokens	none	64,222	72,249
classes left with no declared constructor	0	0	14, of which 7 public
outcome	builds, checks clean	builds, checks clean	builds, one check failed

WHAT EACH RUN REMOVED AND WHAT IT COST ON CAFFEINE

The engine and the first attempt overlap on 4 declarations. The engine also removed four constants of the `@CacheSpec` test-configuration enums (`CacheScheduler.SYSTEM`, `CacheScheduler.THREADED`, `InitialCapacity.ZERO`, `InitialCapacity.ONE`). The attempt also removed the single constant of a private enum, and one method. Neither difference has been adjudicated.

Timefold Solver

WHAT WAS MEASURED	THE CODELASER ENGINE	CLAUDE ATTEMPT 1	CLAUDE ATTEMPT 2
types	12	12	9
methods	229	274	271
constructors	7	55	55
fields	10	12	13
enum constants	2	0	0
parameters	51	52	51
declarations removed	311	405	399
lines removed	1,595	2,025	1,938
files changed	50	107	106
time	2m 12s	23m 03s	27m 34s
input tokens	none	11.3M	10.1M
output tokens	none	84,796	87,989
classes left with no declared constructor	0	48, of which 43 public	48, of which 43 public
tests broken by the removal	0	0	0
tests run on the result	6,269 of 6,269, in all 608 test classes	5,414 of 6,269, in 361 test classes	5,414 of 6,269, in 361 test classes
tests that could not be run	none	855, in 247 test classes	855, in 247 test classes
outcome	● builds, checks clean	● project build fails	● project build fails

WHAT EACH RUN REMOVED AND WHAT IT COST ON TIMEFOLD SOLVER

Trino

WHAT WAS MEASURED	THE CODELASER ENGINE	CLAUDE CODE
types	35	26
methods	396	507
constructors	14	995
fields	43	79
enum constants	26	26
parameters	321	437
declarations removed	835	2,070
lines removed	5,090	8,128
files changed	367	1,402
time	6m 32s	50m 51s
input tokens	none	15.6M
output tokens	none	150,056
classes left with no declared constructor	0	957, of which 885 public
tests broken by the removal	0	0
test classes run on the result (changed modules only)	2,026	2,301
tests run on the result (changed modules only)	27,015	28,119
outcome	builds, checks clean	builds, two checks failed

WHAT EACH RUN REMOVED AND WHAT IT COST ON TRINO

QuestDB

WHAT WAS MEASURED	THE CODELASER ENGINE	CLAUDE CODE
types	44	39
methods	671	949
constructors	29	155
fields	195	176
enum constants	10	3
parameters	598	816
declarations removed	1,547	2,138
lines removed	8,036	9,827
files changed	380	610
time	4m 05s	48m 06s
input tokens	none	7.0M
output tokens	none	75,605
classes left with no declared constructor	0	115, of which 102 public
tests run on the result	41,587 of the suite's 42,044	41,587 of the suite's 42,044
tests broken by the removal	0	133 test classes
outcome	builds, checks clean	133 test classes fail

WHAT EACH RUN REMOVED AND WHAT IT COST ON QUESTDB

What the agent removed that the engine did not

By kind of declaration. The Timefold Solver row is attempt 1.

PROJECT	CONSTRUCTORS	METHODS	PARAMETERS	FIELDS	TYPES
Timefold Solver, attempt 1	48	45	1	3	0
QuestDB	128	312	249	46	8
Trino	983	112	120	36	0

WHAT THE AGENT REMOVED THAT THE ENGINE DID NOT

What the engine removed and the agent did not is much smaller: 3 declarations on Timefold Solver's first attempt and 5 on its second, 16 on Trino and 152 on QuestDB. Among the QuestDB ones are 13 whole types and 65 fields that the agent left in place.

The programs each agent attempt wrote

ATTEMPT	WHAT IT WROTE
Caffeine, attempt 1	a 636-line analyser that reads class files; three small scripts
Caffeine, attempt 2	an 884-line analyser that reads class files, saved where the project's git ignores it; short Python scripts for the edits
Timefold Solver, attempt 1	a 770-line analyser that reads class files; a 281-line program that makes the edits; five small helpers
Timefold Solver, attempt 2	two analysers that read class files, 686 and 653 lines; a 350-line program that makes the edits through the Java compiler's own parser; two small helpers
QuestDB	a 476-line analyser that reads class files, and a variant of it; a 335-line program that makes the edits through the Java compiler's parser; eight small helpers
Trino	a 1,747-line analyser that reads class files and parses source with the Java compiler's parser; two small Java helpers; two shell scripts

THE PROGRAMS EACH AGENT ATTEMPT WROTE FOR ITSELF