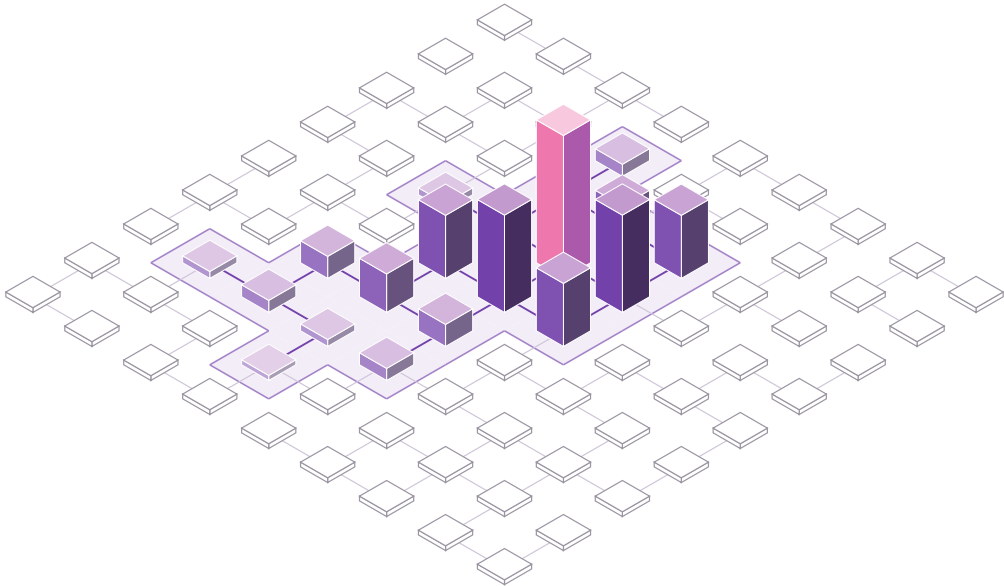




THE CCM BOOK

Code, change, and confidence.

*Algorithms, evidence, and the economics
of software development*



Contents

What a number can promise	4
How the argument develops	4
Three readers, one account	5
A first measurement	6
The inventory before the estimate	7
The short estimate	8
Formats belong to their readers	8
Reading the source tree	9
Language names and policies	9
What the totals preserve	10
Inventory as an audit trail	10
From two snapshots to a task	11
What comparison actually does	11
The model's edge direction	11
A task is more than a changed set	12
A checkout change, calculated	13
Running the complete example	13
Shared and joint work	15
Behavior and evidence	15
Pricing the declared activities	15
The shape of the machine	17
The order of one invocation	18
Few dependencies, explicit tools	18
Selecting work without losing control	20
Ignore rules have scope	20
The walk and the queue	21
Results have no scheduling order	21
Counting languages and mixed documents	22
A declarative catalog	22
One physical line, several languages	23
Typst is a document language	23
Work tokens are not API tokens	24
The mathematics of affected parts	25
Three closure laws	25
Breadth-first evaluation	26
What the closure does not contain	26
When checks become necessary	27
Why positive activation matters	27

Alternatives and conjunctions	27
From necessity to completion	28
Evidence that can survive a change	29
Admissibility and conclusion	29
Residual demand	29
A failure counterexample	30
What reuse can justify	30
Shared work and additional joint work	31
A proof by contribution	31
The checkout calculation	31
What the CLI compares	32
The outcomes evidence leaves possible	33
Complementary evidence	33
An admitted failure can refine cases	34
The relationship to entropy	34
From quantities to an economic scenario	35
A source-based AI workload	35
Residual checks and hosted tools	36
Human work and an optional repeat scenario	36
Overrides and interpretation	37
What the 2026 research profile contains	38
Spending anchors	38
Language effects	38
Prices and time	39
Sensitivity and calibration	39
Experiments we can reproduce	40
Startup is part of the question	40
Three workloads with CCM 0.6.0	41
Where the improvement came from	41
Correctness is another experiment	41
Learning from real development work	43
A useful resource ledger	43
Separate the activities before fitting	44
Prediction error before a confidence story	44
Testing an AI effect	44
Revision as a scientific operation	45
Where the calculus can grow	46
Description length and change description	46
A geometry before a topological interpretation	46
Observers and composition	47
Path observations and distributions	47
Typed graphs before graph spectra	48
The gate back into the runtime	48

A report, a book, and a visual identity	49
One logo, several media	51
The publication pipeline	51
Screenshots as observations	52
The last design rule	52
Contracts and a reading desk	53
The supported surfaces	53
Useful invocations	53
Limits and evidence	54
A small glossary	54
Research and source references	54

What a number can promise

We open a repository because something must change. Perhaps a price is wrong, an interface must move, or a service can no longer satisfy its contract. The files are visible, but the task is not contained in their line count. We must discover what the change can disturb and what evidence would justify accepting it.

A counter can tell us how much text is present. A dependency model can tell us which declared assumptions may be invalidated. An evidence model can tell us which declared checks still need support. An economic scenario can attach resource prices to assumed activities. Each statement is useful precisely because it has a different meaning.

CCM keeps these meanings separate. Its name is Code Counter Machine, but its design begins where ordinary counting becomes insufficient. The familiar table remains the entrance to the system. Beyond that table we find a finite Change and Certificate Calculus, an explicit account of accepted evidence, and a research profile for AI development costs. We can inspect the inputs to every stage.

The word confidence in this book means a justified position within a stated model. It does not name a probability that an arbitrary program is correct. An accepted test may exclude some modeled outcomes while leaving many unmodeled behaviors untouched. A passed check can be useful without becoming a proof of everything we care about.

We will follow a small checkout change throughout the book. Pricing and inventory change together. Checkout depends on both, while a payment boundary remains unchanged. This example is deliberately small enough to calculate by hand and large enough to expose shared checks, additional joint checks, invalidated evidence, and a reusable result. The source fixtures and task model live under `examples/book`. The screenshots are generated by running CCM against those files.

How the argument develops

We begin as readers of a report. We learn the inventory's units, compare two source snapshots, and declare a task model. Then we descend into the implementation: file selection, bounded workers, lexical state machines, graph closure, obligation activation, and evidence transport. The algorithms become easier to assess once we have a concrete result to explain.

The mathematical chapters state assumptions before drawing conclusions. A closure law follows from a fixed directed graph. A residual-work monotonicity law needs additional conditions on evidence. A cost equation follows from declared resource quantities and prices, while the accuracy of those quantities remains an empirical question. These distinctions matter more than the sophistication of the notation.

The later chapters examine estimation, experiments, architecture, and proposed research. They distinguish mechanisms shipped in the executable from candidates in the design records. Tree-sitter extraction, authenticated evidence, richer code property graphs, persis-

tent topology, and imported work ledgers are research or engineering directions, not hidden capabilities of the current CLI.

Three readers, one account

A developer can follow the worked example and then read the algorithms that explain a particular result. A planning team can begin with the report and economic chapters, where every quantity receives a unit and every default receives a status. A domain expert can follow the propositions, counterexamples, evidence interaction, and research gates. We use the same definitions on all three routes.

We use short names in the interface and precise names in the model. An artifact is a modeled part. An obligation is a required check. A certificate is an evidence record admitted by the declared policy. The technical terms remain available because they help us state exact claims; the report uses ordinary language because a result must be understandable before it can guide a decision.

This edition describes CCM 0.6.0. The counting policy is lexical-v5, the preferred task input is model version 2, and the economic profile is the frozen 2026 R&D scenario. Performance measurements identify their own executable and workload. A version number on this book does not retrospectively change an older experiment.

The useful promise is modest and concrete. We can make a measurement reproducible, a finite inference inspectable, and an assumption replaceable. Those are the foundations on which stronger claims can eventually stand.

Vikrant Rathore wrote CCM and this book, with assistance from Ronak Rathore and Kanak Rathore. CCM is free software under the GNU Lesser General Public License, version 3 or any later version.

A first measurement

The first useful run should answer a familiar question: what is in this source tree? We should not need to learn a theory before reading that answer. CCM therefore begins with Language, Files, Lines, Blank, Comments, and Code. The table uses ordinary names and grouped numbers. Color reinforces those names when stdout is a color terminal.

We build the release executable before comparing speed with another tool. A Cargo development build serves debugging and iteration; its runtime is not the release-performance claim. For a local installation we can let Cargo place the optimized executable on the path.

```
cargo build --release
cargo install --path cmd/ccm --offline
ccm examples/book/after
```

The fixture contains four Rust files. Each has three physical code lines, so the inventory has twelve program lines. The program does not execute those files. It reads their bytes, applies the selected lexical policy, and assembles a report.

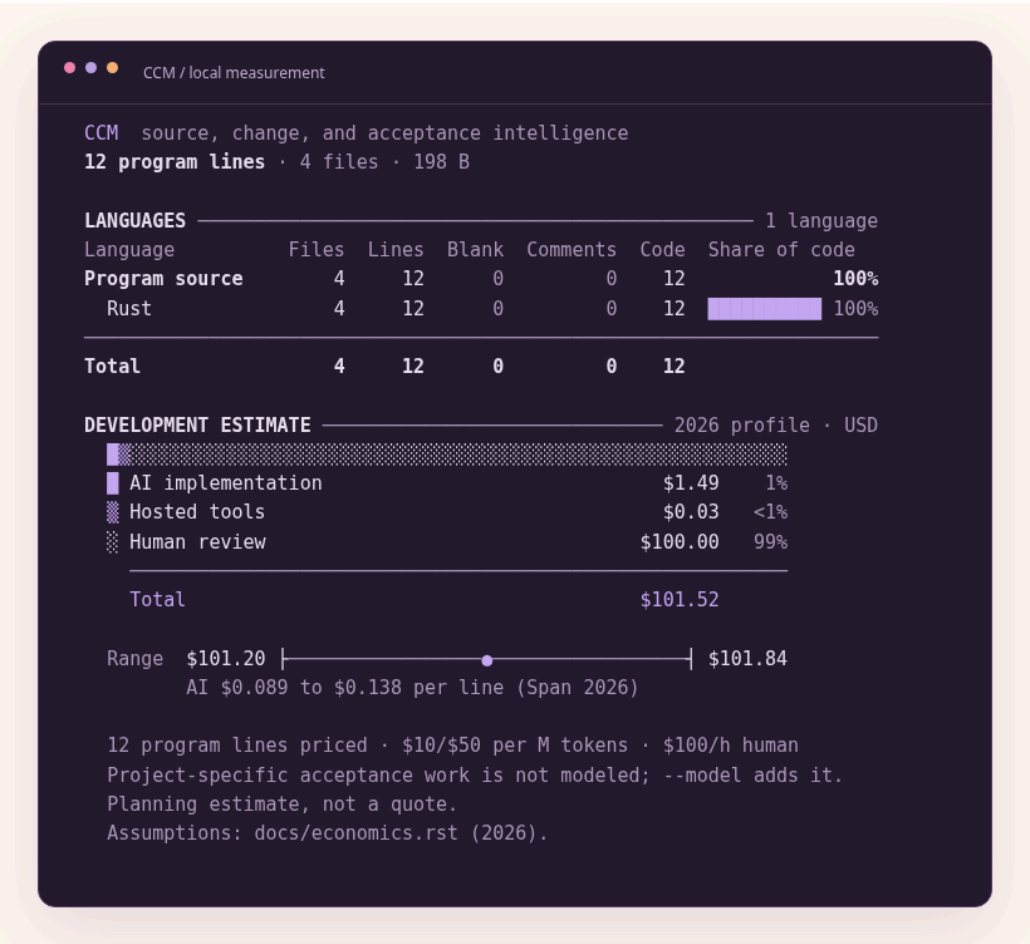


Fig. 1 The executable’s output, captured with the book’s terminal palette. The screenshot is generated from the checked-in fixture rather than drawn as an illustrative result.

The inventory before the estimate

Lines is the physical total under the policy’s line-ending rules. Code, comment-only, and blank form a disjoint partition of that total. A line containing both an expression and a trailing comment belongs to Code. The Code column also follows the established counter convention for documentation and data: it contains their noncomment content. The language row supplies the context needed to interpret that convention.

Program lines is a narrower aggregate. It selects policies classified as program source and leaves documentation and data outside that total. The AI implementation scenario narrows the scope again by excluding recognized generated files. These quantities need not be equal, and a difference is not an arithmetic failure. We will keep their selection rules visible as the example grows.

A normal skip is different from a failed measurement. A binary image, a symlink, or an explicitly excluded directory does not require a warning for each path. An unreadable input,

an invalid task model, or a resource limit needs a diagnostic. A detected lexical coverage failure produces an incomplete report and exit status 2. This distinction lets the useful table remain readable while preserving failures for automation.

The short estimate

Below the inventory we see a compact development scenario. It separates AI implementation, hosted tools, human review, and any additional modeled activities. Zero optional activities do not occupy rows in the ordinary report. Unknown project-specific acceptance work is described as unmodeled rather than priced as a known zero. The assumptions reference points to the economic account instead of reproducing a research essay in the terminal.

The default figure estimates a counterfactual rebuild of selected, nongenerated program source with an AI coding agent. It is not a reconstruction of the repository's historical spending. It is not the purchase price or asset value of the software. The human review component uses an explicit per-file assumption, and the agent component uses the editable 2026 profile. We will derive both quantities later.

For a counting-only view, we remove the scenario. For a compact price and coverage explanation, we request verbose output. For the full expert account, we request the technical report.

```
ccm --no-cost examples/book/after
ccm --verbose examples/book/after
ccm --technical examples/book/after
```

Formats belong to their readers

The terminal report is for a person making a quick inspection. HTML carries the same measured bundle into a standalone document with collapsible assumptions and no runtime scripts or remote assets. JSON preserves numeric components, unknown scope, policy identifiers, and model results for agents. CSV and NDJSON serve tabular and record-oriented consumers. We do not ask an agent to recover structured facts by scraping the human table.

```
ccm examples/book/after --html -o report.html
ccm examples/book/after --json -o report.json
ccm examples/book/after --format csv -o files.csv
```

Existing output files are protected unless we supply `--force`. Forced replacement uses the output writer's atomic replacement path rather than truncating the previous report before rendering succeeds. A rendering budget also bounds the complete report before publication.

A color terminal normally receives color automatically. A pipe normally receives plain text. A nonempty `NO_COLOR` variable disables automatic color, and `--color=always` is the explicit override. The labels retain their meaning in monochrome, in a narrow terminal, and in a PDF screenshot. Color is a reading aid, not an additional data channel.

Reading the source tree

A source inventory is an observation under a selection policy. We can change the policy and obtain another valid inventory of the same checkout. The important question is whether the report makes that policy recoverable.

Consider a dependency lockfile. It can contain many thousands of lines and still describe data rather than implementation work. CCM gives Cargo and JSON lockfiles their own rows. It counts their physical content by default, but those rows do not inflate Rust or JSON source groups and do not become program lines in the AI implementation workload. A comparison with a tool that excludes lockfiles must first align the selected files. Otherwise the apparent disagreement combines selection and classification.

The same reasoning applies to generated files. A generated Rust file is Rust source for the physical inventory. Its origin matters to a rebuild estimate, so recognized generated files are excluded from the default implementation driver. The scanner looks for declared markers in a bounded header; it does not infer authorship from style. `--no-generated` changes selection and removes those files from the inventory itself.

Language names and policies

The registry contains 658 language and format names and aliases in this edition. Sixty-seven native policies cover common source forms, assembly in its common dialects, Starlark, Roff, and mixed documents. The wider declarative catalog handles literal comment and quote definitions. An alias can select the same native policy as another name, so the registry size is not a count of distinct parsers.

Native and declarative are implementation tiers. Neither means that CCM compiles the language or proves semantic equivalence. The distinction tells us where specialized handling exists and where a generic delimiter machine is being applied. The machine-readable language list exposes that policy identity.

```
ccm --languages
ccm --languages --json
ccm --count-as custom:Rust path/to/project
```

Exact native filenames and TypeScript declaration suffixes have priority. Catalog filenames, compound suffixes, native extensions, and catalog extension aliases follow the documented selection order. Extension matching is case-insensitive; exact filename matching retains case. Where several languages share an extension, the default is deterministic rather than omniscient. An explicit override is more honest than silently claiming to recognize every dialect from its contents.

The catalog's compatibility manifest records CLOC and scc names and SLOC extensions. Those checks establish name and path coverage. They do not establish that every language

construct has been independently validated or that different tools assign every line to the same class. The policy chapter will show why literal syntax definitions and specialized state machines both have a place.

What the totals preserve

CCM recognizes LF and CRLF physical lines and counts a final nonempty segment without a newline. An empty file has zero lines. A leading UTF-8 byte-order mark is ignored for classification while its bytes remain in the byte total. Text in a legacy encoding such as Latin-1 is counted by its ASCII structure, since every delimiter CCM recognizes is ASCII. NUL-containing inputs and known binary asset extensions are ordinary binary skips under the relevant selection rules.

The underlying classes retain their definitions through aggregation. Adding language rows produces the same physical totals as adding file reports. Every lexer counts a whitespace-only line as blank, even inside a multiline string or a block comment. So the blank count of a file equals its number of whitespace-only lines, a fact anyone can check without CCM. We do not need another counter to settle such a question: the definition fixes the answer, and a short program checks it on every file. `counting.rst` states this and the other properties that the counts guarantee, with the checks behind them. These details are part of the versioned policy, `lexical-v5`.

Python docstrings are another useful example. A triple-quoted string that begins a line is documentation by convention: the language keeps it as the object's `__doc__`, and no code reads it otherwise, so CCM counts its lines as comments. A triple-quoted string inside an expression, such as an argument or an assignment, stays code. We must record such conventions before comparing Code or Comments, while the physical line total remains a direct cross-check.

Inventory as an audit trail

A JSON file record includes its path, language, purpose, generated marker result, counts, and language-policy identifier. The enclosing report records selection limits, skip counters, notices, and whether the selected supported text was counted completely. Unknown types remain visible in the coverage summary. They can hide additional work, so they do not become zero-sized software components by implication.

The inventory cannot reveal missing requirements or code that has not yet been written. It cannot establish the value of a feature or the skill needed to change it. It supplies a physical coordinate system for later reasoning. That coordinate system becomes useful when we keep it attached to its policy and resist asking it to answer a different question.

From two snapshots to a task

A snapshot tells us what is present. A comparison tells us which selected files have different bytes. Neither statement yet tells us why a change is being made. The task model supplies that missing context.

Our checkout example changes two values. The pricing function returns 100 before the edit and 110 afterward. The inventory function returns 10 before the edit and 9 afterward. The checkout and payment source files retain their bytes.

```
ccm --compare examples/book/before examples/book/after
```

CCM reports modified pricing and inventory files. It also distinguishes added and removed files. A comment-only edit is still a byte change, even if it leaves program behavior untouched. This is deliberate: the snapshot comparison is an exact file-byte operation, not a semantic equivalence test.

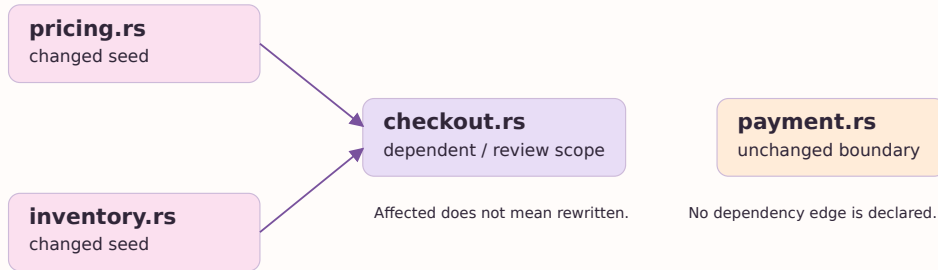
What comparison actually does

The project adapter first measures both snapshots under the same selected options. It matches root-relative file identities and compares the retained files' bytes, with bounded reads and mutation checks. Equal line counts are insufficient to establish equal contents. Two files can have the same number of code lines while implementing different functions.

The comparison retains before and after counts alongside its file status. It does not compute a general patch algebra or estimate the implementation effort of individual changed statements. In the current economic adapter, a comparison selects whole changed source files in the current snapshot for the rebuild scenario. A removed file has no current implementation lines, but it can still contribute review work. This scope is conservative about what a file-level observation can justify.

The model's edge direction

We now declare that checkout depends on pricing and inventory. The graph records the direction in which invalidation travels: an assumption points to its dependent. A pricing edit therefore reaches checkout. A checkout edit does not automatically reach pricing through the same edge.

Two edit seeds. Three affected parts.

Pricing and checkout evidence is invalidated. Payment evidence survives.

Required check units: 7. Remaining check units: 6. Shared: 1. Additional joint: 3.

Fig. 2 The declared dependency model for the running example. The payment part has no dependency edge in this model.

This graph is supplied by the model author. The lexical language registry does not extract it. Broad counting support and automatic semantic dependency extraction are separate engineering problems. Tree-sitter, code property graphs, compiler data, and patches can become future adapters, but a source counter does not obtain their guarantees merely by recognizing an extension.

A task is more than a changed set

The model also declares required checks and the evidence already available. A pricing edit activates a pricing contract check. An inventory edit activates stock availability. Either edit activates checkout compatibility. Their combination activates an additional combined checkout test. A payment boundary review is standing work, active even with no edit seeds.

The model's context identity binds these declarations to a task, source snapshot, environment, and policy by author assertion. The identity is required, but the current executable does not authenticate it or cryptographically bind it to source bytes. A precise string is a useful discipline; it is not a substitute for a provenance mechanism.

We can let comparison paths supply the seeds when those paths match declared artifacts. We can also supply explicit seeds with repeated `--changed` options. Explicit names are particularly useful when artifacts represent interfaces or environmental assumptions rather than files. An abstract artifact can participate in exact finite reasoning while remaining unmapped to an implementation line count. The estimate must then preserve the unpriced scope.

The next chapter makes these declarations executable. The result will explain three different sets: the files whose bytes changed, the parts reached by invalidation, and the checks that remain open. Keeping those sets distinct is the first step toward an intelligible change report.

A checkout change, calculated

We now have enough context to write the task model. The preferred format begins with `ccm-model` and version 2, separated by a literal tab. Records use tab-separated fields; comments begin with `#`. Names must be declared before they are referenced. The checked-in file is `examples/book/checkout.ccm`.

Four artifacts name the source parts: pricing, inventory, checkout, and payment, with their `.rs` suffixes. Two edges point from pricing and inventory to checkout. The declarations deliberately omit a dependency edge to payment. The payment boundary remains relevant through a standing check and surviving evidence rather than through reachability.

Table 1 Required checks in the example

Check	Units	Activation
Pricing contract	1	Pricing is affected.
Stock availability	1	Inventory is affected.
Checkout compatibility	1	Pricing or inventory is affected.
Combined checkout test	3	Pricing and inventory are affected.
Payment boundary review	1	Always active.

Trigger syntax keeps alternatives and conjunctions explicit. A comma joins members of one conjunction; a semicolon separates alternatives. A dash denotes the empty conjunction, which is true and therefore activates standing work. Weights are nonnegative policy units. They do not become hours until an economic assumption supplies a conversion.

Running the complete example

```
ccm examples/book/after --compare examples/book/before \
--model examples/book/checkout.ccm
```



Fig. 3 One change report connects file differences, affected parts, evidence, and resource assumptions.

The seeds are pricing and inventory. Reachability includes each seed and their dependent checkout, producing three affected parts. Payment is outside the closure. All five checks activate, with total weight seven.

The previous pricing test depends on pricing, so it is invalidated. The previous checkout test depends on checkout, so it is also invalidated. The payment review depends only on payment and survives. It supports the standing boundary check, leaving six policy units pending across four checks. These are results of the declared finite model; no checker is executed by this invocation.

Shared and joint work

A pricing-only batch activates the pricing, compatibility, and payment checks. An inventory-only batch activates stock, compatibility, and payment. The compatibility check is shared work. The combined checkout test appears only for the union and contributes three additional joint units. Standing payment work is removed from the interaction comparison.

The shared weight is one and the additional joint weight is three. Their difference is two. A positive net interaction here means the combined activation creates more joint work than it shares. It is not a statement that batching is always undesirable, and it is not yet a monetary result. Later we will derive the identity behind this calculation.

Behavior and evidence

The example declares four hypotheses with four different observation labels: accepted checkout, wrong price, wrong stock, and payment timeout. Each admitted evidence item supplies an explicit compatibility filter over those hypotheses. After the edit, only payment evidence survives. Its filter leaves three hypotheses and three distinct observations compatible. The displayed distinction capacity is therefore approximately 1.585 bits.

Without changed seeds, all three accepted records survive. Their filters intersect to one declared hypothesis and one observation, giving zero distinction bits within this finite universe. That result is not a proof of arbitrary checkout correctness. It is a statement about the hypotheses the author chose to enumerate.

Pricing the declared activities

The review assumption is 0.25 hours for each affected artifact. Three affected artifacts therefore contribute 0.75 review hours. Six remaining policy units, at one human hour per unit, contribute six more hours. At 100 USD per loaded hour, these human activities total 675 USD.

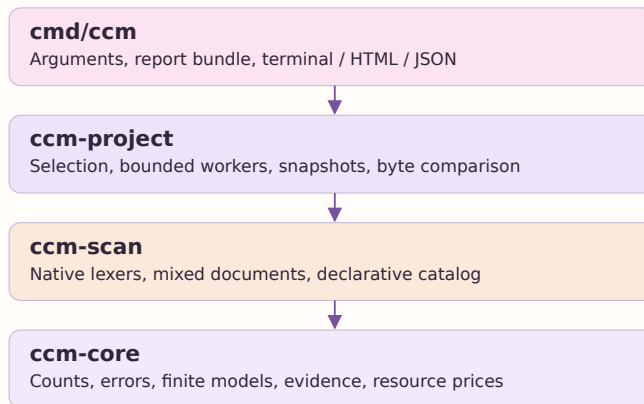
Implementation scope attaches to the two seed source files, which contain six program lines in total. The 2026 profile prices their inferred Rust workload at about 0.74424 USD. The additional agent check budget contributes 0.084 USD and one assumed hosted session contributes 0.03 USD. With no additional repeat probability, the expected total is about 675.85824 USD, displayed as 675.86 USD.

The report rounds displayed components independently and rounds the total after summation. A one-cent difference between the sum of displayed components and the displayed total can therefore occur. The JSON report retains the underlying numeric components for reconciliation. The quantities are useful because their origin is visible, not because the small example validates a general development forecast.

The shape of the machine

A command-line application is easier to reason about when its core can be used without a terminal. CCM's workspace follows that rule. The finite mathematics lives in a library that does not read files or print reports. Filesystem policy and presentation surround it as separate adapters.

Four crates. Dependencies point downward.



The CLI and project adapter also use ccm-core directly. The core never reads files.

Fig. 4 Compile-time dependencies point toward a smaller, filesystem-independent core.

ccm-core owns counts, structured errors, finite models, behavioral cases, and resource-price equations. Its functions borrow declared inputs and return owned results or diagnostics. This boundary lets a unit test evaluate a three-artifact graph without creating a checkout or redirecting stdout. It also lets another application reuse the finite evaluator without adopting CCM's CLI vocabulary.

ccm-scan turns bounded UTF-8 input into physical counts under a selected language policy. It knows lexical states and mixed-file boundaries. It does not decide which repository paths should be selected, which artifacts depend on one another, or how many dollars an activity should cost.

ccm-project owns those filesystem observations that a scanner should not make. It walks paths, applies ignore rules, feeds bounded workers, detects generated headers, retains stable file identities, and compares snapshots. It returns a report containing data and notices rather than printing them. The application can therefore render the same observation as terminal text, HTML, or JSON.

cmd/ccm parses options, assembles the current and optional baseline inventory, loads a declared model, maps seeds, asks the core for results, applies explicit economic assumptions, and renders one bounded report. A bundle holds these observations together. The

bundle does not erase the distinction between an absent task model and a model whose pending work is zero.

The order of one invocation

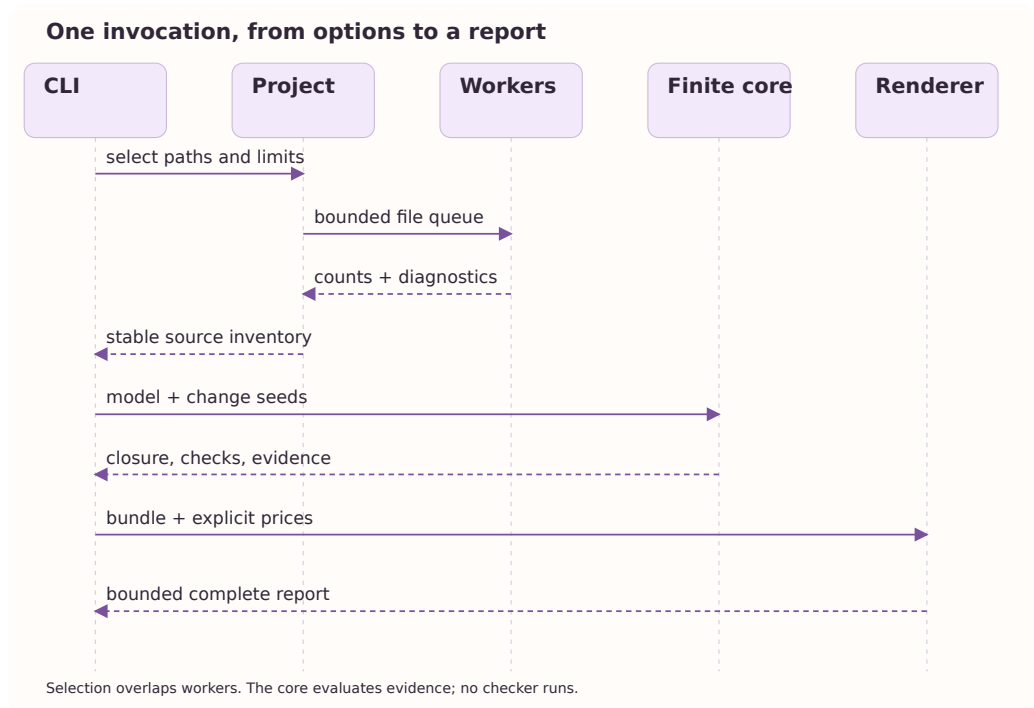


Fig. 5 The high-level call sequence. Worker execution overlaps selection, while model evaluation follows the completed inventory.

The application validates assumptions before expensive filesystem work. Selection and lexical workers then overlap through a bounded queue. The project adapter collects results and sorts them into a stable report. If a baseline was requested, byte comparison determines changed file identities. If a task model was requested, parsing and validation precede finite evaluation. Pricing and rendering come after these inputs have explicit status.

The renderer receives a complete bundle rather than asking the filesystem questions of its own. This keeps format changes from becoming measurement changes. HTML branding, a table's spacing, and a JSON field can evolve without changing the graph closure algorithm. The output writer receives a completed string only after the rendering budget has been respected.

Few dependencies, explicit tools

The runtime workspace uses the Rust standard library and its own small modules. It has no external crate dependencies. This choice is useful for distribution, auditability, startup, and controlling the amount of code needed to understand a result. It does not require

reimplementing a full compiler for every language. The policy tiers expose where generic definitions end and specialized handling begins.

The book and artwork are development tools, not runtime services. `ccm-art` emits deterministic SVG with a small standard-library writer. The documentation task invokes Guidedog, Typst, and Chromium to build the website, typeset the PDF, and capture live screenshots. CeTZ and Fletcher draw the book's monochrome cover illustration. Those tools are not launched when a developer runs `ccm` against a source tree.

A future tree-sitter or compiler adapter should preserve the same direction of dependence. It can supply richer typed edges, source bindings, or checked evidence to the core. It should not make the core depend on a network service or a human report format. The architecture's value is the boundary it preserves as the sources of evidence become richer.

Selecting work without losing control

Before a byte can be classified, the project adapter must decide whether to read it. This stage often determines both the meaning and the runtime of a report. A very fast lexer does not help if every ignored path triggers unnecessary work. A complete-looking table is not useful if the selection rules cannot be explained.

CCM begins with explicit roots and a bounded walk. It normalizes retained file names relative to each root and avoids duplicate paths across overlapping roots. Where no Git repository or ignore file governs a tree, as in an unpacked archive, common build and vendor directory names are excluded by default. In a governed tree the project's own ignore rules decide, because a name such as `build` can hold hand-written source, as it does in Chromium. Explicit literal exclusions, extension filters, generated-file selection, and scoped ignore files refine that initial policy. Symlinks are ordinary skips and their targets are not intentionally followed. Measurements assume stable input checkouts; mutation detection is best effort rather than an atomic filesystem snapshot.

Ignore rules have scope

A root or ancestor can contribute rules, while a nested directory can add later local rules. CCM reads `.gitignore`, `.ignore`, and `.ccmignore`. An ordinary repository can also contribute `.git/info/exclude`. The current implementation does not interpret global Git configuration or resolve an external Git directory referenced by a worktree's `.git` file.

Rule order matters because a later inclusion can reverse an earlier exclusion within the supported scope. Anchored rules apply relative to their base. Basename rules compare a candidate's name; path rules compare its relative path. A negation cannot restore a child of an excluded directory unless the parent directory is itself reenabled and traversed. This is why selection cannot be implemented as an unordered set of filename suffixes.

The rule engine supports escaped literals, stars, double stars, question marks, bracket classes, and ASCII POSIX classes. Its general matcher uses a bounded dynamic program rather than recursive backtracking. For a pattern with p tokens and a path with n bytes, that state machine uses work proportional to p times n and linear row storage in n . A separate work budget prevents an adversarial collection of patterns from demanding unbounded matching.

The common case can be cheaper without changing the rule. Literal prefixes and suffixes reject impossible matches early. A single star surrounded by literal text has a direct matcher, with the same slash-crossing rule as the full state machine. Tests compare these paths with the full matcher over generated path strings. Scope-relative paths are resolved once for a group of local rules instead of once per rule.

The walk and the queue

Directory enumeration discovers which ignore filenames are present. The loader can then avoid negative metadata lookups for every absent file in every directory. Candidates rejected by ignore rules do not need the same retained-file metadata as candidates sent to a worker. These changes matter on repositories with many small files, where filesystem and selection overhead can dominate lexical counting.

The walker takes each entry's type from the directory listing, so it does not call `stat` for every file. It sends selected files through a synchronous queue with 64 slots per worker. A full queue applies backpressure to selection. Workers reuse byte buffers rather than allocate a new buffer for each file. The default uses one worker per hardware thread, at most sixteen, with an explicit CLI range from one to sixty-four. Increasing that number is not automatically an improvement: scheduling, storage, file size, and the serial selection stage all matter.

The worker opens each file and reads its size and modification time from the open file. It applies the per-file and snapshot byte budgets, performs a bounded read, and compares the open file's metadata before and after the read. A detected change becomes an error rather than a silently inconsistent snapshot. This is a useful guard under the stable-checkout assumption, not a proof against every filesystem race.

Results have no scheduling order

Workers can finish in any order. The project adapter sorts retained files and notices before aggregation and presentation. This makes repeated JSON reports deterministic for stable inputs and options, even when worker counts differ. Determinism belongs to the result, not to a claim that threads executed the same interleaving.

Limits constrain the walk, selected files, per-file bytes, total snapshot bytes, nesting, ignore records, and rendering. The ordinary defaults allow 1,000,000 selected files, 64 MiB per file, 64 GiB per snapshot, and traversal depth 128, enough for a Linux kernel checkout. A limit failure is explicit. The CLI does not publish a normal success table after silently discarding work needed to remain within a budget.

The design lesson is broader than a counter. A bounded pipeline makes the amount of admitted work part of the contract. That contract gives later claims about speed, completeness, and memory a stable object to refer to.

Counting languages and mixed documents

A lexical counter needs enough structure to distinguish a comment marker from the same bytes inside a literal. It does not need to answer every question a compiler answers. The useful boundary is a documented policy whose results we can reproduce, including the cases it cannot safely classify.

The native source lexer keeps explicit states for normal text, nested block comments, quotes, and delimited literals. Language policy selects line-comment markers and enables specialized constructs. Ordinary spans are scanned together until a relevant special byte appears. The lexer preserves a disjoint physical-line classification as it crosses those states.

Rust provides a good example of why delimiter tables alone are insufficient. A raw string can contain quote characters and apparent comment markers until its particular closing quote-and-hash sequence appears. A lifetime apostrophe is not an open character literal. Nested block comments need a depth counter and a bound. CCM handles these cases natively rather than pretending that every apostrophe begins the same sort of string.

Shell syntax poses another problem. A heredoc delimiter governs later physical lines, while a here-string does not open the same deferred region. The characters used for a bit shift inside arithmetic must not become a heredoc opener. The shell policy therefore tracks heredoc queues and bounded arithmetic nesting in addition to ordinary quotes. JavaScript regex character classes, Python triple strings, Go backticks, Lua long delimiters, and markup comments require their own documented distinctions.

A declarative catalog

The wider catalog separates literal syntax definitions from the algorithm that applies them. A definition can supply several block-comment pairs, several quote pairs, line-comment markers, case-insensitive matching, escape behavior, and declared nesting. The bounded machine can then count many language families without importing a large parser dependency for each one.

Catalog presence does not establish compiler-level fidelity. Some languages require fixed-column rules, preprocessing, interpolation grammars, or content-based extension disambiguation that a generic delimiter machine does not implement. CLOC's executable regular-expression filters are not imported into CCM. The registry's name and extension checks establish compatibility coverage, while focused fixtures establish particular lexical behavior. The JSON policy tier keeps those claims distinguishable.

This is also why a count difference is not resolved by copying another counter's total. We first align selected files and physical lines. Then we locate the policy choice or unsupported

construct responsible for the classification difference. Only a language specification and a stated counting convention can determine which change to make.

One physical line, several languages

A Vue component can contain a template, TypeScript, and CSS in one file. A closing script tag and an opening style tag can even share one physical line. Counting each extracted region independently and adding its line totals would count that line twice.

CCM's mixed-document adapter records a small classification mask for each physical line. It locates supported script and style regions outside markup comments and quoted attribute examples. The appropriate lexer classifies the embedded source, while markup classification handles the surrounding document. The adapter combines their code and comment flags at the original line position. A code flag wins over a comment-only flag, and each physical line contributes once.

Vue and Svelte use this mechanism for supported JavaScript, TypeScript, CSS, and SCSS blocks. An unsupported declared embedded language produces a coverage diagnostic. Arbitrary preprocessors and template-expression grammars remain outside this native subset. The adapter's precision comes from preserving the shared physical coordinate system, not from claiming to compile the component.

Typst is a document language

A prose quote in Typst is not necessarily a code string. A fenced raw example can contain an unmatched quote or an unfinished comment because those bytes are examples, not active Typst syntax. Applying a conventional source-string lexer to the entire document caused valid repository documents to be rejected.

Typst nests three modes: markup, code, and math. A `"` opens a string only in code and math, so the lexer must know which mode it is in at every byte. It keeps a stack of open frames: content blocks, code blocks and groups, equations, and expressions embedded with `#`.

An embedded expression is the delicate part, because nothing marks where it ends. Typst ends it by rule, and the lexer follows the same rules. A statement such as `#let` or `#set` runs to a newline or `;` outside any delimiter, so a header that spans several lines inside parentheses stays code. A control expression such as `#if` ends after a body block that no `else` follows. Any other expression is a chain such as `#text(red)[body].len()`, which ends at the first space.

An earlier scanner forgot an open statement at each line end. In this book's own template, the header of `#let book(...) = {` spans three lines, so the scanner returned to markup too early, and the next `rgb("#...")` opened a string that never closed. The mode stack removes that failure, and a test keeps it removed. A raw example still protects its contents, and a link such as `https://a//b` in markup is text, not a comment. The cover drawing of this book is also written in Typst; CCM counts it like any other Typst file.

Work tokens are not API tokens

An optional source-work counter collapses recognized literals and counts lexical words and punctuation outside comments. Its policy identity is source-work-v1. It is collected for technical inspection and is not the billing-token quantity used by the AI estimate. No tokenizer library can turn this lexical measure into observed historical model usage without a separate empirical mapping.

Native scanning is linear in ordinary input traversal with bounded language-state overhead, while delimiter and catalog dispatch add costs determined by the selected policy. Mixed-file mapping adds storage proportional to the physical line count. The implementation bounds file size, delimiters, and nesting rather than relying on a universal grammar-complexity claim. The resulting measurement is a physical one, useful because its algorithm and limits are stated.

The mathematics of affected parts

The finite calculus begins with a graph and a question. Which declared parts can be reached by invalidation from the parts we changed? We do not need a probabilistic model to answer that question once the graph is fixed. We do need to state what its edges mean.

Let A be a finite set of artifacts and E a directed dependency relation. An edge from a to b means that changing a may invalidate an assumption used by b . It is a conservative propagation declaration, not a statement that b 's source must be rewritten. Let S be the set of changed seeds.

We define the affected closure by reachability, including paths of length zero. Every seed therefore belongs to its own closure.

$$D(S) = \{b \in A \mid \text{some } a \in S \text{ reaches } b\}.$$

For the checkout example, the seeds are pricing and inventory. Each reaches itself and checkout. There is no declared path to payment, so payment is not affected by this calculation. An omitted real-world dependency would make the model incomplete; it would not make the reachability algorithm nondeterministic.

Three closure laws

The first law is extensivity: S is a subset of $D(S)$. The zero-length path proves it immediately. The second is monotonicity: adding seeds cannot remove an already available path. The third is idempotence: starting from every already reachable vertex cannot reach anything that was not reachable from an original seed. Concatenating the two paths supplies the proof.

$$\begin{aligned} S &\subseteq D(S), \\ S \subseteq T &\Rightarrow D(S) \subseteq D(T), \\ D(D(S)) &= D(S). \end{aligned}$$

These laws explain the word closure. They are properties of ordinary graph reachability, not a new discovery about software effort. Their usefulness comes from giving later activation and evidence rules a precise set to inspect.

A fixed graph also gives a union law. A path begins at a seed in the first batch or at a seed in the second batch. Those two possibilities exhaust the union.

$$D(S \cup T) = D(S) \cup D(T).$$

This equality does not say that the required checks of a combined task are just the union of the separate checks. A check can require affected parts from both sides before it activates. The next chapter makes that distinction explicit.

Breadth-first evaluation

The implementation validates every artifact index and builds adjacency lists. It sorts and deduplicates neighbors, sorts and deduplicates seeds, marks each seed seen, and enters it into a queue. When a vertex leaves the queue, unseen dependents are marked and appended. The first discovery records a predecessor.

A predecessor chain is a witness that explains why a part was reached. With unit graph edges and breadth-first traversal, first discovery gives a shortest edge-count path from the seed set. Stable ordering makes the selected witness deterministic when several shortest paths exist. The witness is not a causal proof that a dependent will fail; it is a reproducible path in the declared invalidation graph.

After adjacency construction, the traversal uses work proportional to the number of artifacts and edges. Sorting contributes the corresponding neighbor-ordering cost before traversal. Storage includes adjacency, a seen vector, a queue, and one optional predecessor per artifact. The evaluator checks explicit object, edge, and membership budgets before admitting this work.

What the closure does not contain

A closure is not a patch, a schedule, or a risk probability. It can include an unchanged file whose assumptions deserve review. It can omit an important component if the author omitted its edge. It is exact as a finite inference and conditional as a model of a real system.

This separation gives us a useful design rule. A source adapter may improve the graph, but it must not silently change the graph's meaning. A lexical call-like token, a compiler-resolved dependency, and an explicitly reviewed assumption edge have different evidential strength. A future typed graph should preserve those distinctions instead of merging them into one unexplained complexity score.

When checks become necessary

The affected closure describes propagation. Acceptance work needs another declaration: which conditions make a check necessary? The same affected set can require different checks under different task policies. A library update, a security repair, and a behavior-preserving cleanup need not share one acceptance contract.

Each check q has a nonnegative weight and a collection of trigger terms. A term is a set of artifacts that must all be affected. Several terms provide alternatives. This is a positive disjunctive normal form: an OR of ANDs, without negated affectedness tests.

Let the trigger family of q be $T(q)$. The check is active when at least one term is contained in the affected closure.

$$q \in O(S) \iff \exists t \in T(q) : t \subseteq D(S).$$

An empty term is always contained in the closure. It therefore represents standing work. A check with no trigger terms would have no way to activate, which is different from a check with one empty term. The input parser's dash notation supplies the latter.

Why positive activation matters

Suppose we enlarge S to T . The closure law tells us that $D(S)$ is contained in $D(T)$. Any trigger term already contained in the smaller closure remains contained in the larger one. Active checks are therefore monotone under added seeds for a fixed graph and trigger model.

Define required demand as the sum of active check weights. The runtime uses checked nonnegative integer policy units; the algebra also makes sense for arbitrary nonnegative real weights. Within the supported integer range, its reported weight sum is exact.

$$R(S) = \sum_{q \in O(S)} w_q.$$

Nonnegative weights preserve monotonicity of this demand. The statement would fail for an activation language with arbitrary negation or for a sum containing negative weights. Those alternatives can be useful in another model, but they need different laws. The implementation does not quietly adopt them.

Alternatives and conjunctions

Checkout compatibility has two singleton terms: pricing and inventory. Either edit activates it. The combined checkout test has one two-member term containing both parts. It activates only after the combined closure contains both. These declarations distinguish shared work from newly joint work without guessing from source size.

A larger trigger is not necessarily a larger check weight. The author can give a small joint check a weight of one and a broad standing review a weight of ten. The units encode a declared policy scale. They remain separate from counts of source files, minutes, tokens, or dollars.

Evaluation visits trigger terms and their memberships against the closure's seen vector. It accumulates active indices and checked weight sums. The model bounds total memberships so a compact-looking input cannot conceal unbounded conjunction work. An overflow is a diagnostic rather than a wrapped total.

From necessity to completion

An active check is necessary under the model. It is not necessarily unfinished. Evidence can already support it, and that evidence can either survive or be invalidated by the current change. The distinction between active and pending is the reason we need a certificate calculus rather than a list of tests.

The checkout example activates seven units of checks. The surviving payment review supports one of them. We therefore see seven required units and six pending units. The difference is meaningful only because the report names the evidence and its support. A bare total could not explain whether the reduction came from reuse, omission, or an arbitrary discount.

Evidence that can survive a change

A previous result is useful only under assumptions that still hold. We represent those assumptions as the support of an evidence record. Support can name source parts, interfaces, environmental conditions, or other modeled artifacts. The author is responsible for choosing a support set that is adequate for the claim.

Let $\sigma(c)$ be the support of evidence c . In the finite runtime, an admitted record survives the change when its support is disjoint from the affected closure.

$$c \in C_S \iff c \text{ is admitted and } \sigma(c) \cap D(S) = \emptyset.$$

This is a transport rule within a context. It does not authenticate the evidence, execute a test, or prove that the declared support is sufficient. The context identity and evidence policy say what an author claims to have admitted. A future provenance adapter could strengthen that claim without changing the set calculation.

Admissibility and conclusion

Model version 2 distinguishes a record's admission from its conclusion. A passed record can support named checks. An admitted failed record can remain useful as an observation, but it cannot discharge acceptance. An unverified assertion does neither. The validator rejects an accepted assertion class rather than treating an unchecked statement as a passed check.

The evaluator first decides which admitted records survive. It then records favorable coverage and surviving failures for each named check. An active check is pending if favorable coverage is absent or if a surviving failure names it. A surviving failure therefore wins over simultaneous favorable coverage in the acceptance result. This conservative conflict rule is separate from the behavioral compatibility calculation.

In the running example, pricing evidence supports pricing and checkout evidence supports checkout. Both supports intersect the affected closure and are invalidated. Payment evidence supports only payment and survives. The report can therefore explain the reusable result without pretending that all old tests remain valid after any edit.

Residual demand

Let $P(S)$ be the active checks that remain pending after transport and the conclusion policy. Residual demand is their weighted sum.

$$U(S) = \sum_{q \in P(S)} w_q.$$

For the pure favorable-evidence case, adding change seeds can only activate more checks and invalidate more favorable records. A previously pending active check cannot become

supported by losing favorable evidence. Under these conditions, pending checks and residual demand are monotone. This is a conditional theorem, not a general property of every evidence policy.

A failure counterexample

The version 2 failure rule makes the condition important. Consider one standing check q with weight one. A passed record with empty support names q , and an admitted failed record supported by artifact b also names q . With no changed seeds, both survive. The failure keeps q pending, so residual demand is one.

Now change b . The failed record is invalidated while the empty-support passed record survives. The standing check is now supported with no surviving failure, so residual demand becomes zero. The seed set grew, but pending demand decreased.

Nothing in this example changes graph closure or positive activation. The change is in which signed observations survive. A book about the calculus must preserve this distinction rather than carry a favorable-only monotonicity proof into a richer implementation without its hypothesis. The evaluator implements the conclusion policy directly; an operator should inspect a conflict and its provenance before interpreting the smaller residual as a real-world improvement.

What reuse can justify

A reusable result saves only the activities it actually supports under the declared policy. It does not grant a universal discount on implementation work. It does not make a dependent source file unchanged in a byte comparison. It does not convert a failed test into an acceptance certificate because the failure was informative.

This is why the economic adapter prices residual check demand separately from implementation scope. A favorable record can remove modeled verification activity while leaving the source workload unchanged. The effect has a named mechanism and explicit units. That is a stronger account than multiplying an entire project price by an unexplained evidence score.

Shared work and additional joint work

Two changes can share a check and still create another check that neither needed alone. A single net number can conceal both effects. The calculus therefore keeps shared and additional joint work before taking their difference.

We compare two disjoint seed batches S and T in a fixed model. The implementation rejects overlapping batches in this comparison API. Standing checks are removed from the interaction terms so that work present before either batch is not mistaken for sharing created by the edits.

Let H contain nonstanding checks active for both batches. Let J contain checks active for their union but for neither batch separately. Their weights are the shared and additional joint demand.

$$H = (O(S) \cap O(T)) \setminus O(\emptyset), \quad J = O(S \cup T) \setminus (O(S) \cup O(T)).$$

The interaction report returns both sets and their sums. It also returns the signed net quantity J minus H in policy units.

$$\begin{aligned} \Delta R &= R(S \cup T) - R(S) - R(T) + R(\emptyset) \\ &= \sum_{q \in J} w_q - \sum_{q \in H} w_q. \end{aligned}$$

A proof by contribution

We can prove the identity one check at a time. A standing check contributes once to every required-demand term and therefore contributes zero after the baseline correction. A nonstanding check active in only one separate batch remains active for the union and contributes zero. A check active in both separate batches contributes one copy for the union and loses two separate copies, leaving minus its weight. A newly joint check contributes only for the union, leaving plus its weight. Positive activation rules exclude a check that is active separately but absent from the union. These cases exhaust the possibilities.

The proof describes required-demand activation. It does not claim that evidence transport or human execution will have the same interaction. A reused test, a conflict, a shared environment setup, and a repeated agent attempt can add further resource interactions that need their own model.

The checkout calculation

Pricing alone needs pricing, compatibility, and the standing payment check, for three units. Inventory alone needs stock, compatibility, and payment, also three units. Their union needs all five checks, for seven units. The empty batch has the standing payment unit.

The baseline-corrected difference is therefore seven minus three minus three plus one, which is two. We can also read it as three joint units minus one shared unit. The combined checkout test supplies the joint work, while compatibility supplies the shared work. Payment supplies neither term in the normalized interaction.

A zero net interaction would not establish independence. A shared check of weight one and a newly joint check of weight one cancel numerically while both mechanisms remain present. The report retains their names so an expert can inspect this case.

What the CLI compares

For several CLI seeds, the current interaction report compares the first ordered seed with the remaining seeds. It is not an exhaustive report of every possible partition or every pairwise interaction. The public core method can compare other declared disjoint batches. The technical report and JSON identify the split used by the CLI.

A scheduling decision needs more than the sign of this number. Batching may share setup, delay feedback, concentrate review, or create integration work outside the declared checks. The interaction calculation supplies an inspectable part of that decision. It becomes valuable when we keep the named sets beside the signed sum.

The outcomes evidence leaves possible

The previous chapters ask which checks are required and supported. A different question asks which task outcomes remain compatible with the evidence. A failed observation can help answer that question while leaving acceptance unresolved. The finite behavioral model exists to keep those two uses of evidence distinct.

The author enumerates a finite hypothesis universe Π . Each hypothesis has an observation label under a declared observer g . Different hypotheses can share the same label when the task treats them as indistinguishable. Each admitted evidence item can supply a compatibility subset of Π . The evaluator intersects the subsets belonging to surviving evidence.

$$K(S) = \bigcap_{c \in C_S} K_c, \quad N(S) = |g(K(S))|.$$

Here the intersection is taken within the declared universe, with no selected filter leaving the full universe available. The runtime reports both the surviving hypothesis count and the count of distinct observation labels. The distinction matters: ten hypotheses can all produce one task observation.

The displayed distinction capacity is the base-two logarithm of the positive observation count. The observation count is exact within the supported finite model; the logarithm is a rounded floating-point display.

$$B(S) = \log_2 N(S), \quad N(S) > 0.$$

This is log-cardinality, not Shannon entropy. No probability distribution is assigned to the cases. If no hypothesis survives, the result is inconsistent evidence within the declared universe and the logarithm is unavailable. It is not zero uncertainty. One surviving observation gives zero distinction bits, which is a different statement.

Complementary evidence

Suppose four hypotheses a, b, c, and d have two observations. Hypotheses a and b produce the first observation; c and d produce the second. One evidence filter retains a, b, and c. Another retains a, b, and d. Each filter alone leaves both observations possible, but their intersection leaves only the first. The evidence is complementary for this observer even though neither item changes the observation count by itself.

The implementation can measure signed evidence interaction against a background set. Writing B for background distinction bits, B_1 and B_2 for the separate additions, and B_{12} for their joint addition gives:

$$I = B_1 + B_2 - B - B_{12}.$$

In the example, the four values are one, one, one, and zero in the corresponding expression, producing positive interaction of one bit. For redundant filters that each remove the same observation, the interaction can be negative. If an intersection is inconsistent, the method returns no numeric interaction rather than applying a logarithm to zero.

An admitted failure can refine cases

Model version 2 allows a failed test to be an admitted observation. Its compatibility filter can exclude hypotheses that would not have produced the failure. The same record does not discharge acceptance and can keep a named check pending. A model can therefore become more informative while its acceptance report becomes less favorable. That is a sensible description of learning from a failure.

The current CLI reports evidence-conditioned hypothesis and observation counts. The public core also exposes the evidence-interaction method. The CLI does not automatically search for an optimal experiment or claim that its declared cases cover real program behavior.

The relationship to entropy

If a future measurement supplies a probability distribution over the distinct outcomes, Shannon entropy becomes available. It would sum the negative probability-weighted logarithms rather than merely count labels. With N possible outcomes its maximum is $\log_2 N$, attained by a uniform distribution. The current runtime has no basis for choosing those probabilities, so it reports capacity and keeps the interpretation narrower.

This restraint is useful for software evidence. A finite universe often begins as an explicit set of requirements examples or mutation hypotheses, not a representative random sample of production behavior. Treating it as a calibrated probability distribution would add a claim that the input does not support. The exact finite calculation is already useful without that claim.

From quantities to an economic scenario

A measured source tree is not an invoice. A required check is not an hour. A distinction bit is not a dollar. To obtain a development scenario we must declare resource quantities and the prices attached to them. The theory gives a disciplined place for those declarations, while empirical work must determine how well they predict a particular task.

The current adapter separates implementation, additional agent acceptance work, hosted sessions, human activities, and optional extra spending. Implementation quantities attach to selected source files or matched change seeds. Human review attaches to the affected closure when a model is supplied. Additional check activity attaches to pending policy units after evidence transport. These choices preserve the calculus's separation between a description of change, its propagated effects, and its remaining assurance demand.

Price declared resources; preserve their units.

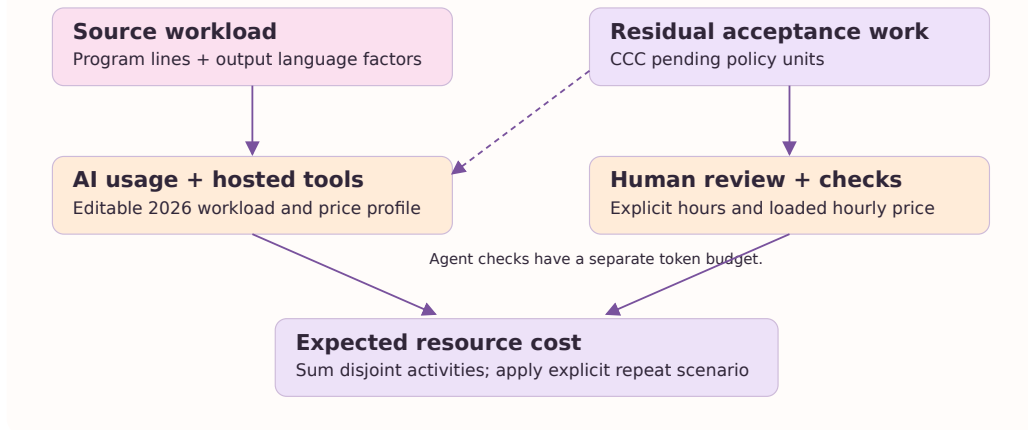


Fig. 6 The economic adapter prices declared activities after the finite calculations. It does not convert semantic information directly into money.

A source-based AI workload

Let L_l be selected nongenerated program lines in language l . Let f_l be that language's output-workload factor and m a user-supplied workload multiplier. The profile supplies inferred input and output tokens per reference program line, t_I and t_O . We calculate implementation token equivalents as:

$$I = mt_I \sum_l L_l, \quad O = mt_O \sum_l f_l L_l.$$

Only the output quantity receives the language adjustment in this profile. Unmeasured languages have a neutral factor of one. The factor is an explicit transferred prior, not a compiler-derived measure of semantic difficulty. We will examine its 2026 evidence in the next chapter.

With input and output prices p_I and p_O in USD per million tokens, the implementation scenario is:

$$C_{\text{implementation}} = (p_I I + p_O O) / 10^6.$$

The prior includes generic exploration, iteration, and verification by an agent. We must not add another default iteration multiplier for the same activity. Project-specific acceptance beyond that generic work has a separate explicit budget.

Residual checks and hosted tools

Let U be pending policy units and b an assumed token budget per unit. The default is 1,000 additional agent tokens per pending unit. Let s be the input share implied by the two per-line token coefficients, using 0.9 when both coefficients are zero. The additional agent acceptance scenario is:

$$C_{\text{checks}} = mbU\{sp_I + (1 - s)p_O\} / 10^6.$$

The coefficient b is an R&D guess. A policy unit is not intrinsically a duration or a token quantity. If the implementation prior already includes these specific checks, we can set b to zero rather than charge twice. Human check hours represent separate human work; fully automated checks can have zero human hours in a locally calibrated scenario.

Hosted tools use a second declared throughput assumption. The default assumes one 1 GB, twenty-minute sandbox session per fifty delivered program lines, rounding the selected scope upward to whole sessions. The workload multiplier m also scales this session count. It prices each session at 0.03 USD in the frozen card. This throughput is a guess, not an observed average from the development study. A zero-line scope incurs no default session charge even when declared check work remains. Additional CI, sandbox use for checks, hardware, storage, search services, deployment, and subscription allocations are excluded unless explicitly budgeted. There is no invented fee for each local tool call.

Human work and an optional repeat scenario

Human hours add review hours per affected part, check hours per pending unit, and explicitly supplied implementation hours. The loaded hourly price can represent resource and opportunity cost rather than only salary. The default per-file or per-artifact review assumption is sensitive to granularity and is not a fitted labor model. Splitting one module into ten files can change this assumption without changing functionality.

Let H be those human hours, p_H their loaded price, A explicit AI/tool spending, S the assumed hosted sessions, and p_S their session price. By default A is zero; an explicit A replaces the automatic implementation, check, and session terms, which are then zero. An optional stationary repeat probability r gives:

$$C = \frac{p_H H + A + C_{\text{implementation}} + C_{\text{checks}} + p_S S}{1 - r}.$$

The geometric multiplier assumes comparable repeated attempts with a stationary independent repeat probability. A real attempt can change the task, evidence, environment, or agent strategy, so this is a scenario rather than a universal process law. The CLI accepts r from zero through 0.95 and defaults to zero additional repetition.

The default and HTML breakdowns multiply every displayed component by the same expected-attempt factor as the total. Machine output also retains per-attempt quantities. Money is calculated with finite floating-point inputs and displayed to cents; it is not exact rational accounting.

Overrides and interpretation

--agent-cost replaces all automatic AI and hosted-tool spending for one attempt. --implementation-hours adds explicit human implementation work. --cost-profile changes token coefficients, prices, session assumptions, workload, year, model label, and registered language factors in argument order. Prices and resource assumptions remain independent inputs.

The resulting number is a resource-cost scenario for a stated scope. It does not estimate market value, willingness to pay, a delivery date, or the number of people required. A richer schedule or staffing model would need capacity, dependencies between activities, delays, and empirical service-time information. We keep those missing quantities visible instead of deriving them from source size by convention.

What the 2026 research profile contains

The first profile is a reproducible starting point for research and development. It uses development evidence published in 2026 and provider prices frozen on 2026-10-23. It does not use foundation-model training expenditure or the historical COCOMO labor equation. The purpose is to approximate development with an AI agent under visible assumptions.

The important distinction is between an observed spending example and a measured token workload. The public evidence we used supplies the former more directly than the latter. We therefore preserve the conversion as an inference.

Spending anchors

[Span's July 2026 study](#) describes 103 engineering teams over twelve weeks of development activity from May through July. Its prompt-clarity examples price fifty merged AI lines at 6.90 USD and 4.45 USD under different conditions. These examples imply 0.138 USD and 0.089 USD per merged line. We choose their midpoint, 0.1135 USD, as a provisional planning prior. It is our chosen midpoint, not a reported population mean or a confidence interval. The observational examples do not establish a causal effect or a universal greenfield rate.

The public summary does not provide a universal input/output token count per delivered line. We assume a 90 percent input share and a 10 percent output share under reference prices of 10 and 50 USD per million tokens. Those choices convert the dollar prior into token-equivalent quantities that can be repriced.

$$t = \frac{0.1135 \times 10^6}{0.9(10) + 0.1(50)}, \quad t_I = 0.9t, \quad t_O = 0.1t.$$

The result is approximately 7,296.43 input tokens and 810.71 output tokens per reference program line. These are inferred billing equivalents, including a broad agent trajectory under the prior. They are not a tokenizer's measurement of a line of Rust or tokens observed in a particular repository run. The source-work-v1 lexical counter is a different quantity and does not enter this equation.

Language effects

[Wu, Anderson, and Guha's July 2026 investigation](#) compares coding-agent output-token use across matched programming problems. Table 7 supplies Java, Rust, and OCaml ratios relative to Python for two proprietary models. Averaging each pair gives provisional output factors of 1.24, 1.26, and 1.29. The study's effects are task-level effects, not measured per-line price differences. Applying them to a line-driven workload and treating the mixed-cohort dollar prior as a reference-language anchor are additional unvalidated transfers.

The profile keeps those choices editable. Other languages receive factor one rather than an invented fitted coefficient. A neutral prior does not claim equal difficulty; it records the absence of a more specific calibration in this first profile.

Prices and time

The [official OpenAI pricing page](#) supplied the snapshot card: GPT-6 Astra standard input at 10 USD and output at 50 USD per million tokens, with reasoning included in output. The hosted shell/code-interpreter reference was 0.03 USD for a 1 GB twenty-minute session. The profile uses uncached standard rates. The CLI does not contact a provider or update these values at runtime.

A price change and a workload change are different revisions. `--ai-input-price` and `--ai-output-price` reprice the same inferred workload. They do not establish that another model will use the same tokens or achieve the same accepted result. A model label remains the reference label unless we change the profile explicitly. This is useful for a controlled scenario and insufficient for a model-quality comparison.

Sensitivity and calibration

The JSON report scales the implementation midpoint by the lower and upper spending-anchor ratios. These are sensitivity scenarios, not statistical bounds. Task clarity, repository readiness, accepted quality, cache use, language, and human intervention can all produce spending outside them.

A future profile should link accepted changes to billed input, cached input, visible output, reasoning, tool invoices, and human activity. It should separate implementation from project-specific verification before estimating both coefficients. Until such a calibration exists, an honest estimate is one whose assumptions can be located and replaced. The versioned file in `examples/ai-costs-2026.conf` makes that replacement concrete.

Experiments we can reproduce

A performance claim begins with a workload, an executable, and a measurement procedure. The name of a competitor is not enough. If one tool reads lockfiles and the other excludes them, their runtimes describe different work even when their tables look similar. We align input before interpreting the timing.

The recorded Ensodoc-derived experiment uses 821 regular files, 187,402 physical lines, and 7,754,367 bytes. A manifest records relative paths, byte counts, and SHA-256 hashes. The source checkout's revision and the measured CCM 0.4.0 release executable hash are retained in `docs/performance.rst`. The snapshot contains an empty `.git` directory as a repository boundary so it does not inherit the enclosing project's ignore policy. It is a local benchmark fixture, not source copied into published benchmark data.

The comparison uses installed `scc` 4.1.0. Both tools produce terminal output into a null sink, with cost models disabled and `scc`'s optional traditional complexity calculation disabled. The driver checks equal file, byte, and physical-line totals before admitting a named corpus. It does not force their code/comment classifications to agree. Those classifications retain the language-policy differences explained earlier.

```
cargo build --release
cargo xtask bench 30 target/bench-ensodoc-v1
```

Startup is part of the question

We measure a monotonic wall clock around the child process. The observation includes startup, traversal, reading, counting, aggregation, and terminal rendering. That is the practical operation a developer invokes. A microbenchmark of a single scanner loop would answer a narrower question.

The driver performs two warm-ups per tool and then thirty paired measurements with alternating order. For each pair it divides `scc` time by CCM time. The summary uses the median of those paired ratios, which need not equal the ratio of the separate medians. It bootstraps the paired ratios five thousand times with a fixed deterministic generator and reports the percentile interval of the median. The procedure does not establish independence between successive warm samples.

The recorded median runtimes are 12.621 ms for CCM and 13.629 ms for `scc`. The median paired speedup is 1.077x, with a 95 percent interval from 1.052x to 1.095x under that resampling procedure. Thirty additional runs per tool record median peak resident memory of 3,886 KiB and 20,858 KiB respectively. The raw time and memory CSV files are retained with the input manifest.

These are historical measurements of the identified 0.4.0 executable. We do not relabel the older experiment as a measurement of every later binary or of every language. A new runtime claim needs its own executable and inputs.

Three workloads with CCM 0.6.0

For 0.6.0 we measured a new executable on three inputs: the synthetic fixture, the same Ensodoc snapshot, and the 40,821 files of the Zig source tree.

Table 2 Median paired speedup over scc 4.1.0 (above one, CCM is faster)

Workload	Size	Speedup, 95% interval	Peak RSS, CCM / scc
Fixture	2,004 files, 6.9 MB	1.607x [1.576, 1.649]	10,000 / 20,792 KiB
Ensodoc	821 files, 7.8 MB	1.195x [1.152, 1.231]	4,470 / 20,864 KiB
Zig	40,821 files, 411 MB	0.780x [0.750, 0.806]	70,618 / 107,916 KiB

On small projects CCM is faster, because startup and traversal dominate. On the large tree scc is faster, and CCM uses about a third less memory. When we gave both tools 4, 8, and 16 threads, both gained about the same factor. The gap is therefore the cost per byte of the lexer, not the way work is scheduled. `docs/performance.rst` records the executable hash, the raw samples, and how the classifications of the two tools compare.

Where the improvement came from

The first implementation was slower on this workload. Measurements pointed toward repeated ignore matching and filesystem work, not merely the arithmetic in the lexer. Resolving a rule scope once, adding semantics-preserving glob fast paths, discovering available ignore files during enumeration, avoiding unused work-token collection, and normalizing generated headers once all reduced admitted work. The final reported measurements follow that engineering process rather than an independent preregistered confirmation.

The complete selected Ensodoc inventory contains 828 files, including additional lock-files and configuration data. It counts all 56 Typst files and 123 Vue components with no coverage warnings and exit status zero. That useful complete scan is distinct from the equal-input timed corpus. A wider inventory and a faster aligned count are two different results, and both deserve their own evidence.

Correctness is another experiment

Small exhaustive graph checks compare closure against a fixed-point implementation. Lexical fixtures exercise named language constructs. Catalog tests check recorded names and extensions. Glob tests compare optimized paths against the full matcher and use Git-backed selection checks. Report tests cover escaping, output limits, replacement behavior, and machine-readable components.

A passing test is evidence about the property it exercises. It is not an empirical validation of the cost prior or a proof of semantic extraction completeness. The book keeps mathematical proofs, finite test coverage, runtime experiments, and economic calibration in different paragraphs because they establish different things.

Learning from real development work

An estimate becomes more useful when it can learn from the work it is intended to describe. The first economic profile is deliberately replaceable. Replacing it responsibly needs a ledger that connects accepted changes, resource consumption, evidence, and the task context. A new coefficient without that connection only changes the appearance of precision.

We begin by choosing the unit of observation. A trajectory can contain several failed implementations, tool interactions, tests, and human interventions before one change is accepted. A merged line is an output convention, not a complete account of those activities. A repository snapshot is further removed: it does not identify which trajectory produced each line or what requirements were absent from the snapshot.

A useful resource ledger

For a measured task we retain the task and source identities, acceptance policy, selected agent and model version, and the relevant environment. Usage should distinguish uncached input, cached input, output, and billed reasoning where the provider exposes them. Hosted sessions, search, CI, storage, and other tool charges need their own invoice-backed quantities. Human implementation, review, intervention, and verification are separate activity observations.

The ledger should identify the accepted patch and the checks that justified acceptance. It should also retain failed attempts rather than charge their resource use to an invisible denominator. A model’s narrative statement that it used a certain number of tokens is not a substitute for a usage record. The current CCM CLI does not import or authenticate this ledger automatically.

Table 3 A proposed calibration record

Coordinate	Observation to retain
Task and context	Requirements, source identities, environment, policy version, and acceptance boundary.
Source change	Accepted patch, selected lines, language, generated status, and formatting policy.
Agent resources	Billed token categories, model identity, cache policy, prices, and invoices.
Tool resources	Session counts and durations, CI use, search, storage, and explicit allocations.

Coordinate	Observation to retain
Human activities	Implementation, review, verification, intervention, and waiting time where measured.
Assurance	Activated checks, evidence support, failures, reusable records, and acceptance outcome.

The schema is proposed research infrastructure. It should not be confused with a current CLI input format merely because it appears in a table. The runtime’s supported task format and cost-profile format remain the smaller contracts described in the reference chapter.

Separate the activities before fitting

The implementation prior already includes generic agent iteration. A task-specific verification coefficient must exclude those same activities or explicitly replace that part of the prior. Human check hours must describe human work rather than duplicate automated runtime. If a sandbox invoice covers a shared session, allocating the full session to each file would overcount the resource. The accounting boundary must be settled before statistical fitting.

Language effects also need a task boundary. Comparing tokens per physical line across two languages can reward verbosity or punish compact syntax. Comparing matched accepted tasks answers another question. A hierarchical calibration can retain language effects while controlling for task family, repository, acceptance criteria, and intervention, but its transfer claims still need held-out evaluation. The first profile’s neutral factors record missing calibration rather than asserting equal effort.

Prediction error before a confidence story

We hold out tasks, and where possible repositories or later time periods, before estimating a forecast’s error. Training and test leakage can occur through dictionaries, near-duplicate tasks, generated code, shared trajectories, or repeated accepted patches. A random split at the line level would rarely protect against those dependencies.

Absolute error and appropriately scoped relative error should accompany a point estimate. Percentage error needs care when true spending is zero or very small. An interval needs a stated procedure and evaluation of its coverage on held-out tasks. The current low and high scenarios are sensitivity anchors, so the CLI must not relabel them as statistical prediction intervals.

Testing an AI effect

A claim that an agent improves development is a causal claim, not a consequence of the source inventory. A useful design can randomize access or pair comparable tasks, define accepted quality before timing, and record experience, environment readiness, and human intervention. The outcome should include resource cost and quality, not only the quantity

of accepted text. A fast solution that changes the acceptance criteria is not the same task completed more efficiently.

CCC can help describe the acceptance boundary and reusable evidence in that experiment. It does not remove selection bias merely by adding graph features. A larger surviving hypothesis count can reflect a broader declared universe rather than a worse tool. A smaller residual check sum can reflect changed policy weights rather than greater productivity. Context identities make these alternatives inspectable.

Revision as a scientific operation

When new evidence supports a better workload or price profile, we version the inputs and retain the previous profile for reproducibility. We change workload coefficients because of development observations and prices because of price observations. We record the date, source, population, task boundary, and transfer limits of each revision. The CLI's flat profile file makes that operation small enough to perform without changing the finite engine.

The aim is not to defend the first guessed coefficient indefinitely. It is to make the next coefficient easier to justify, test, and replace. That is how an R&D scenario can mature into a useful local forecasting tool.

Where the calculus can grow

A research program needs more than a list of attractive mathematical subjects. Each candidate must define an observable object, a computable quantity, a computational cost, and a claim that an experiment can reject. The design record applies that rule to description length, topology, observation abstraction, path entropy, and typed dependency graphs. The present executable implements bounded finite instances of the calculus, not every proposed extraction or analysis.

The distinction protects the program from its own ambition. A novel-looking scalar can be easy to calculate and difficult to interpret. We should first ask which decision the scalar would improve and which cheaper baseline it must outperform. The following constructions are research candidates under that discipline.

Description length and change description

A repeated source file is easy to compress. That observation does not establish that the file is easy to change safely. A small edit to a repeated schema can invalidate many consumers, while a larger isolated edit can require little cross-boundary review. Description and propagation therefore remain separate coordinates.

A reproducible compression feature must name its canonicalization, codec, framing, dictionary, and resource limits. If a dictionary is trained, its cost and identity must remain visible. A two-part description charges for the dictionary as well as the encoded units.

$$L_{\text{two}}(X, D) = L(D) + \sum_{x \in X} L_c(x \mid D).$$

A conditional patch codec can similarly describe a new source snapshot given an earlier one. The reconstruction rule must be explicit: the patch and prior snapshot must recover the claimed new bytes or canonical representation. A short description is then a precise coding result under that scheme. It is not exact Kolmogorov complexity, intrinsic semantic difficulty, or proof that review will be short.

A useful experiment would compare these lengths with token counts, clone features, and source changes at fixed time and memory budgets. Training a dictionary on the evaluated unit would contaminate held-out novelty. Random identifiers, generated boilerplate, minimification, small-file overhead, and compact effectful algorithms are necessary controls. The current CLI does not run these codecs as a hidden complexity engine.

A geometry before a topological interpretation

For a topological experiment we first need a finite state sample and an encoder into a specified metric space. The chosen metric determines which states are near one another. A Vietoris-Rips construction can connect points whose distances are within a scale and add

simplices for compatible cliques. Persistent homology can then ask which features remain across scales.

A persistent loop in this chosen space is not automatically a bug, race, or architectural cycle. The interpretation requires a justified relationship between the state encoder, the task observation, and the feature being measured. Changing the encoder can change the geometry while leaving the program's externally relevant behavior unchanged. A research adapter must expose that dependence rather than presenting an unlabeled topology score.

The computational cost is also part of the candidate. A full clique complex can grow exponentially with the number of sampled states. A bounded implementation needs declared dimension, sample, simplex, and memory budgets, together with an explicit approximation policy. A stability theorem applies under its own metric and filtration assumptions; it does not validate an arbitrary source-derived geometry by association.

Evidence persistence is a different construction. We can examine the unresolved task distinctions after successive evidence acquisitions at an increasing verification budget. With a fixed universe and retained compatible evidence, the surviving observation set can only shrink. An area under its log-cardinality curve would have units such as bits-times-hours when the budget is measured in hours. It would describe the persistence of unresolved distinctions, not their business value or probability of failure. Changing context, invalidating evidence, or admitting an inconsistent intersection requires separate treatment.

Observers and composition

An observer maps programs or hypotheses to the distinctions a task cares about. Two objects are equivalent for that observer when they receive the same observation. The quotient groups those indistinguishable objects. Our finite behavioral model already uses an explicit version of this idea through repeated observation labels.

Category-theoretic language can help state how observation abstractions should respect composition. Its useful question is whether an intended mapping preserves a specified operation or diagram. The existence of a mapping alone establishes no program-correctness result. A certificate transported across an abstraction needs the corresponding preservation assumptions and proof obligations. The present runtime does not infer those assumptions from an arbitrary dependency edge.

This research direction is especially relevant at service boundaries. A boundary contract can hide internal implementation details while exposing the outcomes a dependent task must distinguish. A successful abstraction could reduce the model without losing those distinctions. The validation obligation is to show what the abstraction preserves for the declared task, not to claim universal semantic compression.

Path observations and distributions

A bounded trace experiment can record execution observations up to a declared horizon. Counting possible observed words gives a capacity measure. Assigning empirically justified probabilities enables Shannon entropy or entropy-rate questions. The horizon, state abstraction, input process, and observation policy must all remain explicit.

We cannot infer uniform input probabilities from source syntax. Nor can an observed low-entropy trace establish worst-case safety. A useful comparison would examine path capacity, empirical entropy, branch counts, and transition coverage under the same task and horizon. The current CLI reports no execution-derived distribution and performs no such execution by default.

Typed graphs before graph spectra

A code property graph can combine syntax, control-flow, value-dependence, call, effect, and certificate-support relations. These relation types must remain distinguishable. An AST edge is not a data-flow proof, and a syntactic call is not necessarily a resolved runtime dependency. A future adapter should retain source provenance and a policy for unresolved boundaries.

A spectral experiment can choose a nonnegative symmetric adjacency matrix from selected layers and declared weights. With degree matrix D and adjacency W , the ordinary Laplacian is D minus W . Its quadratic form measures differences across the chosen weighted edges.

$$L = D - W, \quad f^T L f = \frac{1}{2} \sum_{i,j} W_{ij} (f_i - f_j)^2.$$

This identity follows by expanding the square and using symmetry. It says something exact about that matrix model. It does not make a weak graph cut intrinsically good or bad software design. Symmetrizing also discards direction, so the original directed relations must remain available for invalidation and effects.

Degree summaries, cut conductance, and direct closure are cheaper baselines than a full eigendecomposition. A solver adds tolerances, convergence, residual checks, and input-dependent iteration costs. Nonisomorphic graphs can share spectra, so a spectral feature is not a unique program identity. These costs and ambiguities belong in the research gate rather than outside the headline result.

The gate back into the runtime

A candidate earns an implementation slot by answering a specified question with a bounded method and improving a held-out decision beyond simpler measurements. Sound abstract interpretation, slicing, compositional logic, separators, and factorized finite inference can all contribute under their stated contracts. None automatically supplies every guarantee to every language in the lexical registry.

The combined calculus is the research proposal: task-relative description, propagated impact, obligations, evidence transport, behavioral distinctions, and resource accounting with explicit units. Reachability, partition lattices, coding theory, graph spectra, and queueing theory are established foundations. The book does not claim their invention. The experiment must determine whether the construction built from them is useful for software development.

A report, a book, and a visual identity

A measurement is only useful after someone can read it. CCM's interface therefore treats presentation as an explanation of the bundle rather than a second analysis engine. The same inventory, finite model, and economic inputs can produce a short terminal view, an expert view, a standalone report, or a machine record. Their different levels of detail should not produce different quantities.

The terminal report leads with the number most readers want, program lines, and then lists the languages grouped by purpose, each group with a subtotal. Its labels and fill patterns carry the meaning in monochrome; one accent color, bold, and dim text only reinforce the hierarchy, and yellow or red appear only for warnings and failures. The estimate is drawn as a stacked bar with one pattern per component, a legend with amounts and shares, and the calibration range. A change is drawn as a flow from changed files to affected parts to checks. `--format plain`, or a locale that is not UTF-8, gives the same report in ASCII. The compact view omits zero optional activities and points to the assumptions; `--verbose` lists every item, and `--technical` adds the full derivation. This balance lets a planning reader find the result without losing the expert's audit trail.

The HTML report is one self-contained file that the `ccm` binary writes with all of its styles, scripts, and logo embedded. It needs no network and no companion files. Without scripts it shows a static summary; with scripts it lets a reader drill down from languages and directories to single files, and from the estimate and the acceptance plan to each formula, artifact, and check.

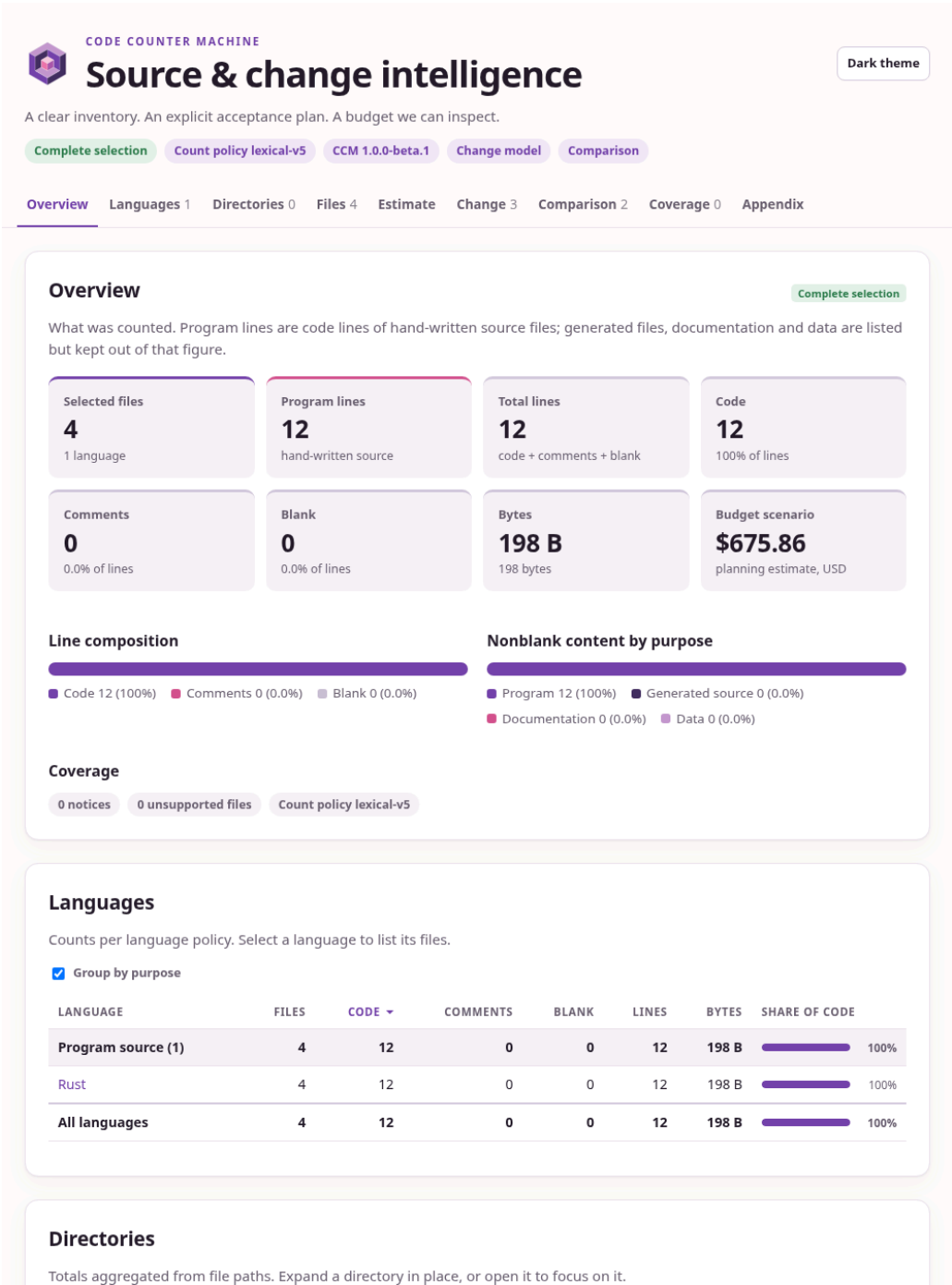


Fig. 7 The live HTML report for the running example. Its logo, styles, and scripts are embedded; the document needs no network asset or companion file.

One logo, several media

The mark is an abstract solid generated by the Rust tool `ccm-art`. It is an isometric cube whose near corner is cut away, with a small pink cube in the inner corner of the cut. Seen along the cube diagonal it reads as three nested hexagons: the outer ring, the hollow notch, and the change at the center. The solid stands for a system, the notch for the region a change affects, and the pink cube for the change itself. Every polygon follows from a few named sizes and the standard isometric projection, and one light direction shades the faces. A monochrome variant uses tints of a single ink.

The same canonical SVG is embedded in standalone reports and staged into the book and website. We do not maintain an unrelated hand-edited mark for each surface. Regenerating the asset therefore changes the visible identity through one reviewed source tool. The tool uses the standard library's formatting and filesystem support, with no external Rust SVG crate.

The book cover tells the same story at a larger scale. It draws a nine-by-nine isometric lattice of modules. One pink module changes, and the modules it reaches through dependency edges rise and fade with their distance from it. A tinted floor region marks the bounded closure, the part of the system that the change can affect. The custom Typst template draws the lattice with `CeTZ 0.3.4`, and a fixed integer hash chooses the edges, so every build draws the same picture.

The publication pipeline

```
guidedog ssg build docs --out docs/_build/site
guidedog build pdf docs/book
```

When we edit generated artwork or the executable screenshots, we refresh those inputs and ask the same installed Guidedog CLI to assemble the result.

```
cargo xtask assets
cargo xtask docs
cargo xtask docs --no-pdf
```

The complete task regenerates vector assets, renders the cover preview, stages shared files, builds CCM, captures live report screenshots, and asks Guidedog to assemble the landing page, HTML book, PDF book, and design-record catalog. The build invokes the installed `guidedog` executable on `PATH`; `--guidedog PATH` selects another installed executable. Guidedog generates the PDF through its normal compiler support; the book configuration does not override it with an external executable. The separate Typst CLI is needed only for SVG artwork and PNG previews, while a headless Chrome, Chromium, or Edge captures the report screenshots. The screenshots are committed; without a browser the task keeps them and finishes the build. `cargo xtask assets` checks for the Typst CLI first and reports an Elm-style error with an installation hint if it is missing. We can still build the book through Guidedog from the prepared assets. These are publication tools, not dependencies invoked by ordinary measurement.

The landing page introduces the same three questions as the book: what is present, what a declared change affects, and what evidence or resources remain necessary. It links directly to the HTML book, downloadable PDF, examples, and design records. The assembled site is a local artifact until we deliberately publish it.

Screenshots as observations

The CLI screenshots begin with actual ANSI output from the executable. A small renderer displays that transcript in the book's terminal palette, and a headless browser captures the result. The report screenshot begins with CCM's own HTML output for the same fixture. Neither image is a manually invented table. The generated JSON beside them makes the quantities independently inspectable.

The fixture and its source model keep screenshots from becoming ornamental examples that no longer match the software. We can regenerate them after a CLI change, inspect overflow and contrast, and compare the underlying numeric result. A useful screenshot is another reproducible view of a measurement.

The last design rule

A beautiful interface does not need an unexplained score. It needs hierarchy, readable names, stable units, and a visible route to detail. The prose, diagrams, and artwork in this book follow the same rule as the executable: a claim should have a place where its inputs and limits can be found.

Contracts and a reading desk

The earlier chapters explain the design in sequence. This chapter gathers the contracts we most often need while using or extending the program. The executable's `--help` remains the authority for the complete current option list. The source records retain the rationale and unresolved questions behind an interface.

The supported surfaces

Table 4 Runtime interfaces

Surface	Contract in this edition
Human reports	Terminal, plain, standalone HTML, verbose detail, and technical detail.
Agent and table reports	ccm.report.v3 JSON, file CSV, and record-oriented NDJSON.
Language registry	ccm.languages.v2 JSON with native-lexical-v5 and declarative-v1 policy tiers.
Task model	Preferred tab-separated ccm-model version 2; version 1 keeps its legacy failed-evidence interpretation.
Cost profile	Bounded UTF-8 flat key=value entries, applied in argument order.
Development assets	Rust SVG generator, custom Typst cover, Guidedog publication, and actual-report screenshots.

The report identifies lexical-v5 as its aggregate count policy. A file also identifies its own language-policy tier. The current finite model is author-declared; `semantic_guarantee` remains unavailable. Machine consumers should preserve null and unknown values rather than replace them with zeros.

Useful invocations

```
ccm .
ccm --no-cost .
ccm --color=always .
ccm --by-file --sort code .
ccm --technical --json .
ccm --format csv --include-ext rs,py .
ccm --no-generated --exclude vendor .
ccm --cost-profile examples/ai-costs-2026.conf --verbose .
```

The `--model` and `--changed` options connect a declared task to explicit seeds. A comparison can supply seed paths when names match the model. `--fail-on-pending` makes open active checks or inconsistent modeled observations an exit-status condition. It does not execute those checks.

Table 5 Exit status

Status	Meaning
0	Selected supported text completed; pending checks remain report data unless the explicit failure mode is enabled.
1	Arguments, I/O, model validation, resource admission, or output failed.
2	A report was produced with a detected lexical coverage failure.
3	Explicit fail-on-pending mode found an open active check or inconsistent modeled observations.

Limits and evidence

The task file is bounded to 4 MiB and 100,000 physical record lines. The finite evaluator admits at most 100,000 artifacts, checks, certificates, or seeds of the corresponding kind, with one million edges and one million relevant memberships. The behavior model admits 100,000 cases and filters and bounds case-filter work to twenty million steps. Physical selection and output limits have their own options. The ordinary renderer budget is 256 MiB.

These are admission contracts, not targets every ordinary run must approach. A library error carries a category, plain explanation, hint, and optional source location. Libraries do not print it themselves. The application renders a diagnostic appropriate to the output surface and preserves failure status for automation.

A small glossary

An artifact is a modeled part, not necessarily a source file. A seed is a part declared changed in the current task. The affected closure contains every part reached through declared invalidation edges. An obligation is a check activated by positive trigger terms. A policy unit is its declared nonnegative weight. Evidence support names the assumptions on which an admitted record depends. A surviving record has support disjoint from the affected closure under the finite rule. Pending demand is the weighted sum of active checks left open by the conclusion policy. A behavioral hypothesis is one enumerated candidate, while an observation label identifies the task distinction that candidate exposes.

Research and source references

The founding design is CMD 0002, a change and certificate calculus for code measurement. CMD 0003 adopts the finite runtime and activity budgets. CMD 0004 develops competitive counting and evidence-conditioned reports. CMD 0005 records the 2026 AI development profile, and CMD 0006 records the broad catalog and compact report. CMD 0007 records

the Guidedog publication pipeline and shared visual identity. Their source files live under `docs/cmd/records` and their web catalog is mounted beside the book. The founding research record remains a discussion; implemented follow-up records do not automatically close its empirical gates.

The 2026 spending source is [Span's Beyond the Model study](#). The language source is [The Best Programming Language for Tokenmaxxing](#) by Wu, Anderson, and Guha. The price reference is the [official provider pricing page](#), with the profile frozen on 2026-10-23. The chapter on the research profile explains the transfer assumptions before those sources are used for estimation.

Language-catalog provenance and metadata hashes are retained in `docs/benchmarks/language-catalog-2026.json`. The [CLOC catalog](#), [scc syntax metadata](#), and [SLOC definitions](#) have different roles and fidelity limits in CCM. The bundled MIT notices are available through `ccm --licenses`. The counting algorithm and finite economic model are separate implementations.

The cover uses the [CeTZ drawing package](#) and [Fletcher](#) with explicit pinned versions. [Typst](#) typesets the PDF, while Guidedog builds the web and book editions from the same prose. All of these publication sources remain inspectable in the repository.