

TRUST-BENCH: A Synthetic Japanese Financial-Group Benchmark for Enterprise Document Access in Coding Agents

Shamik Kundu Anand Mekala Kazutaka Yamasaki

Trust Co.,Ltd.

{shamik_kundu, anand_mekala, kazutaka_yamasaki}@trust-partner.co.jp

Abstract

In large financial institutions, what code *should* do is rarely recoverable from code alone: the authoritative rules live in internal business documents that are largely unreachable to coding agents. Benchmarks built on public repositories therefore cannot measure what reaching those documents is worth. We present TRUST-BENCH, a fully synthetic benchmark that measures the effect of enterprise document *availability* and of the *channel through which it is delivered*: the codebase of a fictional Japanese financial group—1,105 programs and 287k lines of code across five languages, a legacy COBOL/PL/I/RPG core of record surrounded by the Java and C periphery paired with 149 Japanese business documents and 93 maintenance tasks. Tasks are graded two ways: an LLM judge scores answers for correctness and completeness, and generated code is compiled, run, and checked against curated test cases.

We describe the dataset-generation procedure in detail and report results over four context conditions: code only (the baseline); code with the raw documents on disk; code with documents behind a hybrid retrieval tool; and code with documents behind a knowledge-graph tool. Graph-mediated document access raised retrieval-task correctness from 0.68 to 0.94 while the strict build-task pass rate rose from 27% to 91%.

1 Introduction

Benchmarks for coding agents are usually of this form: take a real repository, a real issue, and check whether the agent’s patch passes the repository’s tests (Jimenez et al., 2024). The recipe has driven rapid progress, but it assumes the task statement plus the code contain everything needed to act correctly—an assumption that fails inside most large enterprises, where the authoritative answers are typically *not* in the code but in internal documents that coding agents cannot reach.

TRUST-BENCH addresses this problem with a new design: a *fully synthetic* enterprise codebase in which document access—and the channel that delivers it—is the experimentally controlled variable. Its central design idea is that “where the decisive fact resides” is itself the object of measurement: each task is anchored to a decisive fact whose placement is recorded in a *visibility taxonomy* (Section 3.3).

Our main contributions are:

- **Benchmark:** a fully synthetic Japanese banking codebase along with multi-modal business documents and maintenance tasks, dual-graded by compiling and running tests (30 build tasks) and by an LLM judge (63 retrieval tasks). We release the merged codebase, documents, and the task set.¹
- **Dataset-generation methodology:** Authoring a frozen decision oracle first, then generating code, documents, and build-task tests against it. The recipe is deterministic where it can be and reusable for building contamination-free enterprise benchmarks beyond finance.
- **An initial baseline study** under four context conditions (Figure 1): a code-only control and three that add the documents—as raw files; behind a hybrid-retrieval tool; or behind a knowledge-graph tool (the latter two via the Model Context Protocol, MCP). The primary metric is the gain Δ of each treatment over control.

2 Related Work

SWE-bench (Jimenez et al., 2024) and its descendants (Chowdhury et al., 2024; Yang et al., 2024a)

¹<https://github.com/shamik-13/trust-bench-public>

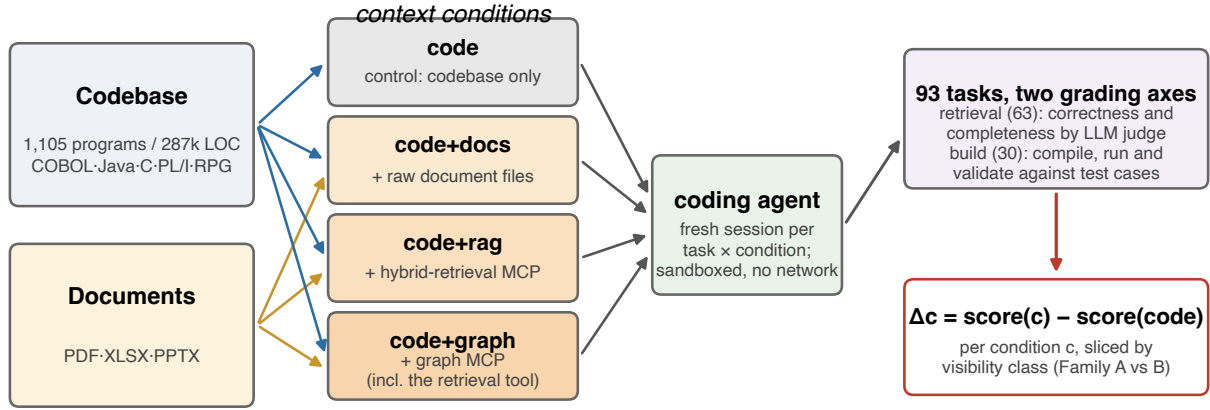


Figure 1: The TRUST-BENCH experiment.

grade agent patches by executing repository tests—the standard we adopt for build tasks—but their public-repository provenance invites contamination and memorization (Liang et al., 2025; Aleithan et al., 2024), which SWE-Bench Pro (Deng et al., 2025) and SWE-smith (Yang et al., 2025) counter with private code and synthetic bugs. TRUST-BENCH instead synthesizes the *entire* codebase, documents, and tasks: no benchmark artifact was collected from a pre-existing repository. On the legacy axis, COBOLEval (Bloop AI, 2024) and MainframeBench (Dau et al., 2024) are snippet-scale and Legacy-Bench (Tchourakov et al., 2026) is execution-verified but document-free and largely closed; none place the agent in a multi-language codebase cross-referencing COBOL, PL/I, RPG, Java, and C.

CRAG (Yang et al., 2024b) and CodeRAG-Bench (Wang et al., 2025) measure how retrieved context changes performance; CRAG’s finding that naive retrieval can *hurt* motivates our multiple conditions. Benchmarks of outdated or contradictory evidence (Ouyang et al., 2025; Hou et al., 2024) study conflict in web text, which TRUST-BENCH embeds as superseded rate sheets and stale design guides whose resolution is part of the task. Simulated enterprises such as TheAgentCompany (Xu et al., 2024), TheMCPCompany (Esfandiarpour et al., 2025), and τ -bench (Yao et al., 2024) come closest, but none vary document access as the experimental treatment over a fixed codebase.

FinBen (Xie et al., 2024), Hirano (2024), and EDINET-Bench (Sugiura et al., 2025) evaluate models on public financial text, and a recent wave of finance-agent benchmarks (Bigéard et al., 2025; Zhu et al., 2026; Cao et al., 2026) adds tool use—

but still over *public* APIs and text with context fixed. TRUST-BENCH instead makes the internal-document channel the controlled variable over an institution’s own legacy code, graded by execution. We are unaware of an existing benchmark that jointly combines a multi-language legacy codebase, synthetic Japanese enterprise documents, controlled document-access conditions, and execution-graded maintenance tasks. The closest prior on/off context study (Dillon and Varanasi, 2026) uses an agentic harness, but over eight tasks with LLM-scored compliance and neither legacy code nor execution-based grading.

3 The TRUST-BENCH Benchmark

3.1 Setting: one combined codebase

TRUST-BENCH simulates the codebase of *Mirai Financial Group*, a fictional Japanese financial group with six companies: a trust bank, a securities firm, a card issuer, a code-payments provider, an insurer, and a shared-platform subsidiary. Each contributes coherent subsystems for 34 scenarios in total. Crucially, the artifact is *not* 34 separable folders: all scenarios are merged into one flat codebase, so locating the right program among 1,105 candidates is part of task difficulty (Table 1). The estate is also deliberately not uniformly legacy: COBOL, PL/I, and RPG programs form the core of record, surrounded by Java services and C components that share the core’s datasets and call chains.

Although each scenario is authored as a self-contained model, the scenarios are not completely isolated. All scenarios are written against a shared convention layer—a common program-naming grammar, a fixed taxonomy of the group’s companies and their subsystems, and a shared set of

Scenarios (six group companies)	34
Programs	1,105
COBOL / Java / C	339 / 524 / 156
PL/I / RPG	54 / 32
Copybooks, headers, includes	418
Lines of code	287k
Documents	149
PDF / PPTX / XLSX	126 / 15 / 8
Decisive facts (visibility-tagged)	92
Document–fact placements	100
Fact–program implementations	98
Tasks (retrieval / build)	93 (63 / 30)

Table 1: Dataset statistics.

document genres. Scenarios belonging to the same company reuse the same subsystems and the same canonical datasets. The boundaries between scenarios therefore dissolve: one subsystem is built up by several scenarios, related programs share the files they read and write, and some call chains run across scenario boundaries. A final merge step (Section 4) then flattens every scenario into one namespace.

The documents in the benchmark are the genres that govern real maintenance: design guides, record layouts, rate sheets, approval minutes, incident reports, migration decks, and operations manuals. Filenames are opaque (DOC-12186.pdf); documents cross-reference one another by internal title and document number, so the agent must open and read them rather than pattern-match on paths.

3.2 Tasks and grading

Every task is run under a *context condition* $c \in \mathcal{C}$. The task prompts and the codebase are identical across all conditions; what varies is the *context channel*—what the agent can reach beyond the source files themselves, and how. One condition is always the code-only control **code**, in which no document is reachable at all and the codebase can be explored only with the harness’s ordinary file tools. The remaining conditions are treatments that expose the document corpus through some channel. Two of the channels we study expose documents alone; the third also exposes a structural index over the code. The channels we study are defined in Section 6.

Retrieval tasks (63) ask the agent to locate, explain, reconcile, or plan. The hidden answer key for task t contains *correctness* points K_t (decisive facts that must be present) and *completeness* points B_t (supporting analysis). Each point is scored by a separate LLM *judge*: it is shown only the question, the numbered key points, and the answer a_t^c ,

and it returns one binary verdict $J(a_t^c, b) \in \{0, 1\}$ per point b , where 1 means the answer covers b . The judge model’s instructions are described in Appendix A.5:

$$\text{corr}_t(c) = \frac{1}{|K_t|} \sum_{b \in K_t} J(a_t^c, b), \quad (1)$$

and analogously $\text{comp}_t(c)$ over B_t .

Build tasks (30) ask the agent to change program behavior and are graded by execution. Each task has a set G_t of *test cases* that run against the agent’s edited code. A workspace that fails to compile automatically fails every test. We report two numbers per task: the fraction of tests passed, $\text{tests}_t(c) = \frac{1}{|G_t|} \sum_{g \in G_t} \mathbf{1}[g \text{ passes}]$, and the strict score

$$\text{pass}_t(c) = \mathbf{1}[\text{tests}_t(c) = 1], \quad (2)$$

which is 1 only when *every* test passes and 0 otherwise.

Task validity is enforced at generation time by two checks on the test suite: the hidden golden program must pass every test (the task is solvable), and the shipped start state must fail at least one (the task is not pre-solved). This is the synthetic analogue of SWE-bench’s fail-to-pass test requirement (Jimenez et al., 2024). Appendix A shows one task family end to end.

For a condition c the aggregate score and the **gain** over control are

$$S(c) = \frac{1}{|T|} \sum_t s_t(c), \quad \Delta_c = S(c) - S(\text{code}), \quad (3)$$

where s_t is any one of the per-task measures defined above— corr_t or comp_t on the retrieval axis, pass_t or tests_t on the build axis. $S(c)$ and Δ_c are computed separately for each measure, and never pooled across measures or across axes.

Task quality criteria. A task is defective if it exhibits any of the four failure modes: *trivial*, *unsolvable*, *mis-graded* (a test that does not check what it claims), or *misleading* (a wrong answer key, or contradictions with no resolution path). We screen for these at three points—when the task is first specified, while the scenario is built, and in the independent review passes of Section 5—and repair or drop whatever fails. We also check that no prompt leaks its answer and that every answer key contains the decisive fact.

3.3 The visibility taxonomy

Each scenario is built around *decisions*—the business rules the benchmark targets, authored as a

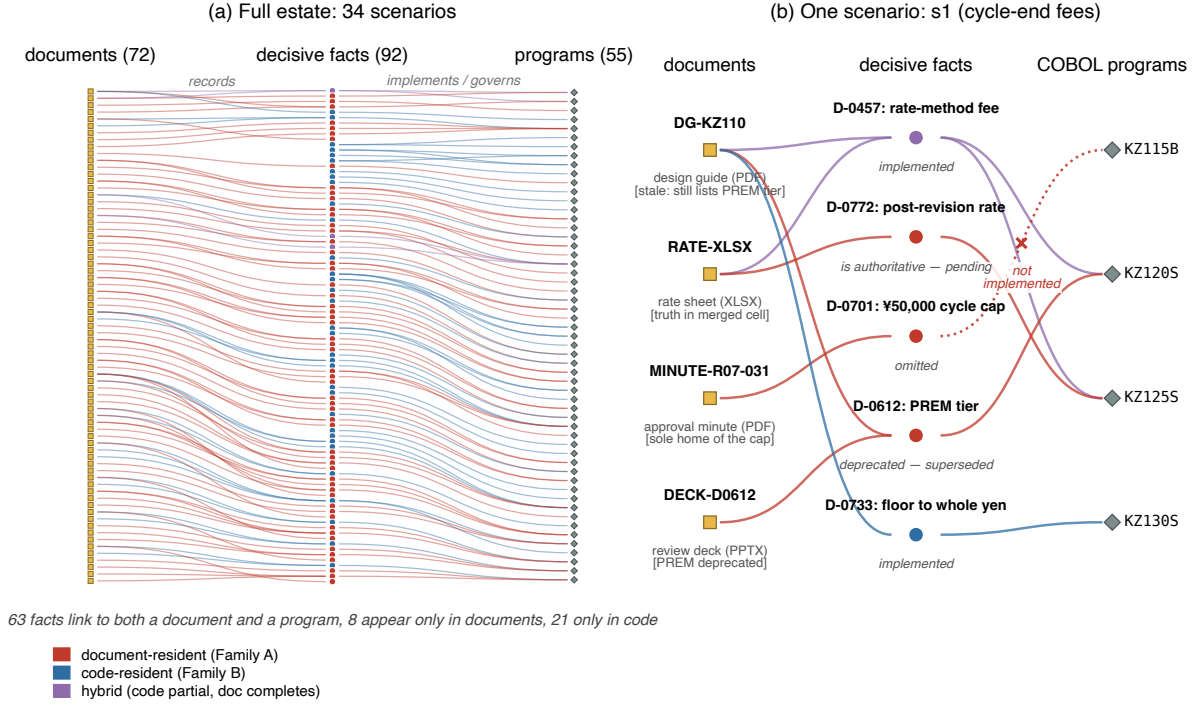


Figure 2: Where decisive facts are located. (a) Documents, facts, and the programs that implement or are governed by them, across the full codebase. Only the documents and programs that carry or implement a decisive fact are shown. (b) Anatomy of one scenario.

frozen registry (Section 4). The decisions are its decisive facts, and each carries a **visibility class** recording where the truth about it can be recovered (Table 2, Figure 2).

The classes group into three families. **Family A (document-resident, 56 facts)**: the fact is recoverable only, or decisively, from documents. **Family B (code-resident, 33 facts)**: the fact is fully recoverable from code—plain, scattered across call chains, or crossing language boundaries—while nearby documents mention only its context, acting as decoys. **Hybrid (3 facts)**: code shows part of the rule and a document completes it.

4 Constructing the Benchmark

Construction is *task-first*: nothing is generated until we have decided what each scenario should ask. Before any code or document exists, we draft the tasks we want the finished scenario to pose—a few retrieval questions and one or two build tasks—each naming the fact it turns on and where that fact will live: in code or in documents or in both. Those drafts become the benchmark’s task prompts and answer keys; here they serve as the brief for everything else. The brief is then formalized as the *decision oracle*, and the pipeline (Figure 3) is built around one invariant: *the oracle is frozen*

first, and every other artifact—code, documents, and tests—is generated to be consistent with it. Phase A builds each scenario from its frozen oracle; Phase B merges the 34 scenarios into one codebase. We describe each step below, using the fee-calculation subsystem (Figure 2b) as a running example. Appendix A shows a concrete example of each step.

The world model. A scenario’s *world model* is a small bundle of files co-authored by an LLM and a human. The main file is the *decision registry* (`decisions.yaml`), a map of the business rules the benchmark is about. Each decision records the correct rule (hidden from the agent, who must recover it), the shipped code’s `start_behavior` (what that code does *before* the rule is applied), an implementation status, a visibility class (*where the truth about it lives*; Table 2), its `home_docs` (the documents that carry it), and the programs that implement it. The status field takes four values that determine what the start code gets wrong relative to the correct (“golden”) behavior: *implemented* (start already correct—no build delta), *pending* (start reads a superseded value), *omitted* (the rule is absent from code and data and must be added), and *superseded* (start carries deprecated remnants to remove). The fee cap of Figure 2b,

Family	Visibility class	<i>n</i>	The decisive fact...
A: document-resident (56)	doc-code-drift	17	... is in a document that <i>contradicts</i> the code; a second one resolves it
	doc-authoritative [†]	15	... lives only in a document; the code has nothing, or a stale value
	locate-authority	8	... is <i>which</i> document/owner is authoritative among candidates
	doc-structural	6	... is in document structure (merged cells, footnotes, strikethrough)
	cross-modal-link	6	... emerges only by joining two document modalities (PDF×XLSX)
	tribal	2	... is operational knowledge, recorded only in a handover memo
	provenance	1	... is <i>why</i> a value was chosen (decision history)
	precedence-ordering	1	... is the priority order among overlapping rules
Hybrid (3)	code-partial→doc-completes	3	... is half-visible in code; a document supplies the missing half
B: code-resident (33)	code	13	... is plainly in one program, but must be found in a large codebase
	code-cross-language	13	... spans a language boundary (e.g. COBOL caller, C or PL/I callee)
	code-scattered	5	... is distributed across several programs' interacting logic
	code-implicit	1	... is implied by code structure, never stated
	code-drift	1	... is in code that drifted from its own comments

Table 2: The visibility taxonomy over 92 decisive facts. [†]In the released artifacts this class appears under labels source-invisible (13; the code has nothing) and doc-text (2; the code holds a stale value), which the paper merges under one name.

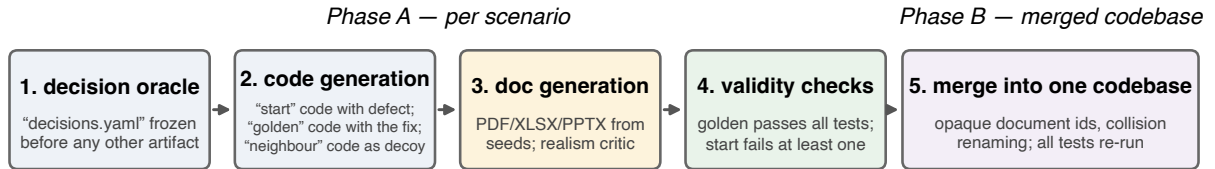


Figure 3: The generation pipeline.

for instance, is registered as omitted with visibility doc-authoritative—the rule lives only in a document while the shipped code has nothing (Table 2)—and with a single approval minute as its home document; the entry is quoted verbatim in Appendix A.1. The cap appears nowhere in the codebase, so the document-bearing conditions can recover it but the code-only control cannot.

This registry is the **oracle**: its visibility/home_docs fields *are* the answer key for where every decisive fact resides, and fixing it before any code or document exists is what makes visibility a controlled property rather than an emergent behavior of generation. Freezing constrains generation rather than forbidding revision: no artifact is produced except from the registry, and when review later forces a change to it (Section 5), every artifact derived from the changed entry is regenerated and re-validated. Two companion files ensure consistency across the artifacts generated from it: contracts.yaml (excerpt in Appendix B) fixes record layouts, call signatures, each dataset’s language, and a shared value/enum *data domain*; estate.yaml lists the decisive *seed programs*, each with a

natural-language specification and the decisions it implements, along with a directive for growing realistic neighbors. Beyond these per-scenario files, a shared convention layer including the program-naming grammar, the taxonomy of companies and subsystems, a common staff roster, and the document genres is fixed once, before any scenario is authored, and every scenario is written against it. This is what lets independently authored scenarios merge into one coherent codebase rather than 34 look-alike silos.

Generating code. The contracts are first rendered *deterministically* into the shared layouts every program includes—.cpy copybooks for COBOL, .h structs for C, package-namespaced records for Java—and, for C/Java, curated headers are copied verbatim from contracts.yaml. Each decisive program is then authored twice from the *same* world model by an LLM along with a compile-and-layout check: the *golden* version receives the decision’s rule and implements it; the *start* version receives only the start_behavior—never the rule—so the start state is code written by a process that genuinely did not know the fix and the fix cannot bleed into a shipped comment

or name. The rest of the codebase is grown by LLM-authoring 15–20 real, interlinked *neighbor* programs that carry no decisive fact and serve as decoys.

Build-task tests. For each build task, we author an input *fixture*, a small loader/dump program that loads it and reads back what the program produced, and the *test cases* that check the result. Test cases are authored by an LLM from the data domain and the decision’s rule so each asserts an observable outcome rather than an implementation detail. The fixture is designed alongside them, also by an LLM, crafted so that each decision surfaces as a distinct checkable outcome. Every suite must satisfy both criteria: golden passes all tests, and start fails at least one. Each suite is then re-validated against the merged codebase in Phase B, where any regression fails the build. Appendix A.6 lists the resulting test cases for the running example. COBOL, C, and Java are compiled and run (cobc/clang/javac); PL/I and RPG have no publicly available off-platform compiler, so facts placed there are readable but not executable and serve as retrieval targets and cross-language decoys.

Generating documents. Documents follow a strict FORM $\times$ CONTENT split. *Templates* (FORM) cover eight document types—minutes, design guide, DB design, batch-job definition, and transcript (PDF); rate and mapping sheets (XLSX); and review deck (PPTX)—each with its layout, chrome, and messiness directives. *Seeds* (CONTENT) instantiate a template with fictional content along with a list naming which decision(s) the document carries and *how* it places them. A per-scenario config states, for each document, exactly which facts it may and may *not* contain—this is where each fact’s visibility-class placement is enforced. Seeds are LLM-drafted, passed through an adversarial *realism critic*, then frozen; deterministic renderers turn the frozen (template, seed) pair into the final PDF/XLSX/PPTX, embedding figures and format-tied messiness (Appendix A.3 shows one rendered passage). A per-scenario *document firewall* then asserts the *visibility invariants*—the placement rule each fact’s visibility class implies, checked in whichever direction that class forbids. A Family A fact’s value and its known paraphrases must be absent from all generated source code, so that the fact really is unavailable to the code-only control; a Family B fact must be absent from every

document, so that the document conditions cannot shortcut the code. The check is a deterministic scan, and it is context-aware rather than a bare token search: a value counts as leaked only when it co-occurs with the context that makes it decisive, which keeps coincidental matches from failing the scan (Appendix C).

Merging into one codebase. Phase B merges the 34 per-scenario codebases into a single flat folder, resolving naming collisions as it does so. Every scenario’s tests are then re-run against the *merged* codebase. Documents are copied from the per-scenario corpora into one flat folder and renamed to opaque ids with a grader-side map recording which document holds which fact, and a global firewall re-scan re-checks every visibility invariant across the merged codebase: a fact hidden from code within its own scenario must also be absent from the rest of the scenarios.

Reproducibility. The dataset generation pipeline pairs a deterministic core with a stochastic realization. The decision oracle is human and LLM co-authored and frozen. Rendering documents from fixed seeds, compilation, the test-suite validity checks, and the firewall scan are all deterministic. But the realization itself, both program source and document prose, is LLM-authored without seed or temperature control, so re-running the generator yields a different corpus. Once the realization is drawn and frozen, deterministic checks certify that any draw satisfies the identical invariants.

5 Dataset Validation

Following the benchmark-quality critiques of Reuel et al. (2024) and Vendrow et al. (2025), we treated the dataset itself as an object of study before running any baseline, and conducted five independent adversarial reviews by fresh agent sessions without authoring context. Each agent examined every scenario against the four failure modes of Section 3.2 and uncovered issues such as firewall leaks where a Family A rule value appeared in a code comment, a golden-key error contradicting the oracle, and retrieval tasks whose keys demanded facts no condition could derive. We finalized the benchmark once every finding was repaired and the reviews were all clean.

6 Experimental Setup

Context conditions. We study four conditions: **code**, the code-only control, in which no document is reachable; **code+docs**, which adds the raw document files (PDF/XLSX/PPTX) on disk with standard extraction tooling (poppler’s pdftotext; openpyxl/python-pptx); **code+rag**; and **code+graph**. The latter two expose the corpus through tools served over the Model Context Protocol (MCP), so the agent decides autonomously whether, what, and how often to retrieve. The **code+rag** condition provides one tool, `search_docs`: hybrid retrieval over extracted, chunked, and embedded document texts. The retrieval combines a multilingual dense embedder with BM25 over Japanese-tokenized text via reciprocal rank fusion. The **code+graph** condition provides MCP tools over a knowledge graph whose nodes are programs, code units, datasets, documents, and document sections, and whose typed edges are static code relations (calls, reads, writes, includes) plus name-resolved cross-modal links: document→entity mentions, document→document citations, and code-comment→document citations. The graph is built *deterministically* using ANTLR parsers for the five languages and rule-based entity-mention scanning over the same extracted document text. The graph builder has no access to `decisions.yaml`, visibility labels, `home_docs`, task prompts, answer keys, golden code, or test expectations.

Crucially, **code+graph** is a strict *superset* of **code+rag**: it exposes the identical retrieval tool over the same index along with the graph tools—so any gain of **code+graph** over **code+rag** is attributable to the added structure provided by various graph-based tools rather than retriever tuning.

Harness. Each (task, condition) pair is run in a fresh, independent session of the agentic coding harness Claude Code with a sandboxed workspace. The control workspace contains no documents at all. Each session has a wall-clock session timeout of 1200 seconds. In our experiments, all the sessions ended with a terminal “subtype”: “success” and none ended due to timeout or token budget exhaustion.

Evaluation subset. Running all 93 tasks for multiple trials under every condition is expensive in both time and cost, so baselines use a **stratified**

Model	Axis	code	Δ_c		
			+docs	+rag	+graph
Sonnet	retr.	0.68	+0.02	+0.17	+0.26
	build	0.27	+0.09	+0.45	+0.64

Table 3: Control score and gain per condition (Eq. 3). Retrieval = judge correctness over the answer-judged questions only; build = all-tests-pass rate.

evaluation subset: 15 scenarios / 50 tasks (36 retrieval, 14 build), chosen to be representative of the entire benchmark. The stratification criteria are as follows: every visibility class must appear, every language must appear, both build and retrieval tasks must appear in the subset.

Models and judge. We run three trials of each scenario using Claude Sonnet 5 in Claude Code to account for model stochasticity. For the LLM judge, we use Claude Haiku.

7 Results

Overall gain. Table 3 reports the control score $S(\text{code})$ and the gain Δ_c of each document condition, per grading axis. Document access delivered through the knowledge-graph tool yields the largest improvement in retrieval correctness (+0.26; paired-question bootstrap 95% confidence interval [+0.11, +0.42]), followed by the hybrid-retrieval tool (+0.17, [0.00, +0.34]), whereas placing the raw document files on disk yields no measurable gain (+0.02). Completeness follows the same ordering (+0.30, +0.16, and −0.05, respectively). The separation is larger on the build axis: the fraction of build task-runs passing all hidden tests rises from 27% under the code-only control to 36% with raw files, 73% with hybrid retrieval, and 91% with the knowledge-graph tool. At the level of individual test cases, pass rates are 48%, 56%, 89%, and 93%, respectively.

Gain by visibility family. Figure 4 slices Δ_c by visibility family. On Family A the **code+graph** condition improves correctness by +0.46 [+0.25, +0.69] from a control baseline of 0.47; on Family B control already scores 0.92 and the **code+graph** gain is +0.08 [+0.00, +0.17].

The Family-A control baseline of 0.47 reflects answer-key composition: each key mixes the task’s doc-exclusive decisive point with contextual points that are legitimately code-visible (e.g. what the shipped implementation currently does, or that no

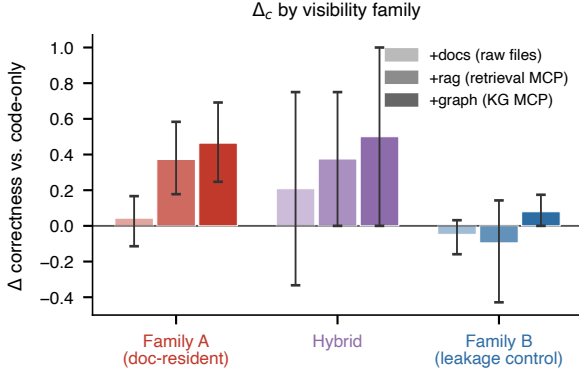


Figure 4: Retrieval-correctness gain over control by visibility family (Sonnet; paired-question bootstrap 95% confidence intervals).

(per session)	code	+docs	+rag	+graph
Cost (USD)	1.14	2.43	0.81	0.69
Wall-clock (s)	242	312	181	170
Tool calls	29	42	21	21
of which MCP	—	—	2	7
Output tokens	14.4k	19.3k	11.4k	10.4k

Table 4: Average efficiency per session.

analogous rule exists elsewhere in the codebase). Notably, the `code+rag` condition’s Family-B gain is *negative* (-0.10), which shows that injected retrieval context hurts the agent’s performance where documents are essentially distractors.

Efficiency. Document access is cost-effective but only through structured channels (Table 4). Control spends its budget searching the codebase; the `code+graph` condition replaces this with a few MCP calls and is *cheaper and faster than control* while scoring higher on both axes. Raw file access is the least efficient condition; the cost is driven primarily by exhaustive reading of the opaque binary document corpus.

Qualitative analysis. The session logs behind Table 3 show three recurring patterns.

Level of variance. On the build task whose fee revision is recorded only in approval minutes, control failed differently in every trial—once “fixing” a decoy defect, once fabricating revised rates from decoy fixture data and asserting them with a confident provenance narrative, once locating the correct routine but declining to edit it without an authoritative source—whereas both tool-mediated conditions retrieved the rate sheet and the approving minutes within their first few document calls in every trial, applied the correct edit, and passed all tests in roughly a tenth of control’s wall-clock

time.

Retrieval anchoring. Quite a few of the `code+rag` errors were anchoring failures, not coverage failures: on one task the agent issued a single query, trusted the top hit—a stale design guide asserting a superseded rule—and answered wrongly, even though the superseding document was indexed and the same condition retrieved it on a neighboring task. `code+graph`, which reaches documents through their typed links to the code entity under evaluation, refined its query, found the superseding document, and answered correctly—itself noting that the guide had never been revised. Since the graph condition includes the identical retrieval tool (Section 6), these failures are not fixable by retriever tuning alone: the escape route lay in the typed links, not in better ranking.

What document access does not fix. On a multi-rule build task, every document condition recovered the approved rules yet applied only a partial fix—editing two of the three programs on the fix path—and failed the tests. Document access improves grounding; it does not enforce implementation discipline.

8 Conclusion

This paper presents TRUST-BENCH, a fully synthetic benchmark that measures the effectiveness of enterprise document access for coding agents. Our study shows that document availability materially improves both what agents answer and what they build, and that the delivery channel matters as much as availability itself. We release the artifacts and the generation recipe, which is reusable for contamination-free enterprise benchmarks beyond finance.

Limitations

Synthetic realism. No synthetic codebase can reproduce the full pathology of forty years of production COBOL. We mitigate this partially with genre-faithful documents, an adversarial realism critic, and multiple audits. **Absence of human review.** Retrieval tasks depend on LLM-as-judge, and the adversarial dataset reviews of Section 5 were likewise conducted by LLM agents. As immediate future work, we plan a human agreement study on a sample of judged answer points. **Baseline scope.** The current baseline study covers a stratified 50-task subset, not the entire benchmark, although the tasks are carefully chosen to be representative of it.

Model scope. The current study uses only Claude Sonnet 5 in Claude Code. Further studies involving different models and different harnesses are needed.

Ethics Statement

TRUST-BENCH is fully synthetic: no real customer data or identifiers were used in creating the dataset. The dataset depicts routine financial-operations logic and contains nothing enabling fraud or evasion beyond public knowledge.

References

- Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. 2024. [SWE-bench+: Enhanced coding benchmark for LLMs](#). ArXiv:2410.06992.
- Antoine Bigeard, Langston Nashold, Rayan Krishnan, and Shirley Wu. 2025. [Finance agent benchmark: Benchmarking LLMs on real-world financial research tasks](#). ArXiv:2508.00828.
- Bloop AI. 2024. [COBOLEval: Evaluating LLMs on COBOL code generation](#). GitHub repository.
- Yupeng Cao, Haohang Li, Weijin Liu, Wenbo Cao, Anke Xu, Lingfei Qian, Xueqing Peng, Minxue Tang, Zhiyuan Yao, Jimin Huang, K.P. Subbalakshmi, Zining Zhu, Jordan W. Suchow, and Yangyang Yu. 2026. [FinTrace: Holistic trajectory-level evaluation of LLM tool calling for long-horizon financial tasks](#). ArXiv:2604.10015.
- Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Karthik Narasimhan, Lilian Ying, Jean-François Chabot, and Tejal Patwardhan. 2024. [Introducing SWE-bench verified](#). OpenAI blog.
- Anh T. V. Dau, Hieu Trung Dao, Anh Tuan Nguyen, Hieu Trung Tran, Phong X. Nguyen, and Nghi D. Q. Bui. 2024. [XMainframe: A large language model for mainframe modernization](#). ArXiv:2408.04660. Includes MainframeBench.
- Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, and 3 others. 2025. [SWE-bench pro: Can AI agents solve long-horizon software engineering tasks?](#) ArXiv:2509.16941.
- Drew Dillon and Kasyap Varanasi. 2026. [Context-augmented code generation: How product context improves AI coding agent decision compliance by 49%](#). ArXiv:2605.08112.
- Reza Esfandiarpour, Vishwas Suryanarayanan, Stephen H. Bach, Vishal Chowdhary, and Anthony Aue. 2025. [TheMCPCompany: Creating general-purpose agents with task-specific tools](#). ArXiv:2510.19286.
- Masanori Hirano. 2024. [Construction of a Japanese financial benchmark for large language models](#). ArXiv:2403.15062.
- Yufang Hou, Alessandra Pascale, Javier Carnerero-Cano, Tigran Tchrakian, Radu Marinescu, Elizabeth Daly, Inkit Padhi, and Prasanna Sattigeri. 2024. [WikiContradict: A benchmark for evaluating LLMs on real-world knowledge conflicts from Wikipedia](#). ArXiv:2406.13805.
- Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. [SWE-bench: Can language models resolve real-world GitHub issues?](#) In *The Twelfth International Conference on Learning Representations (ICLR)*.
- Shanchao Liang, Spandan Garg, and Roshanak Zilouchian Moghaddam. 2025. [The SWE-bench illusion: When state-of-the-art LLMs remember instead of reason](#). ArXiv:2506.12286.
- Jie Ouyang, Tingyue Pan, Mingyue Cheng, Ruirao Yan, Yucong Luo, Jiaying Lin, and Qi Liu. 2025. [HoH: A dynamic benchmark for evaluating the impact of outdated information on retrieval-augmented generation](#). ArXiv:2503.04800.
- Anka Reuel, Amelia Hardy, Chandler Smith, Max Lamparth, Malcolm Hardy, and Mykel J. Kochenderfer. 2024. [BetterBench: Assessing AI benchmarks, uncovering issues, and establishing best practices](#). ArXiv:2411.12990.
- Issa Sugiura, Takashi Ishida, Taro Makino, Chieko Tazuke, Takanori Nakagawa, Kosuke Nakago, and David Ha. 2025. [EDINET-bench: Evaluating LLMs on complex financial tasks using Japanese financial statements](#). ArXiv:2506.08762.
- Leo Tchourakov, Abhay Singhal, and Eno Reyes. 2026. [Legacy-bench: Can AI agents maintain the world’s most critical software?](#) Factory AI technical report.
- Joshua Vendrow, Edward Vendrow, Sara Beery, and Aleksander Madry. 2025. [Do large language model benchmarks test reliability?](#) ArXiv:2502.03461. Platinum benchmarks.
- Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2025. [CodeRAG-bench: Can retrieval augment code generation?](#) In *Findings of the Association for Computational Linguistics: NAACL 2025*.
- Qianqian Xie, Weiguang Han, Zhengyu Chen, Ruoyu Xiang, Xiao Zhang, Yueru He, Mengxi Xiao, Dong Li, Yongfu Dai, Duanyu Feng, Yijing Xu, Haoqiang Kang, Ziyang Kuang, Chenhan Yuan, Kailai

Yang, Zheheng Luo, Tianlin Zhang, Zhiwei Liu, Guojun Xiong, et al. 2024. [FinBen: A holistic financial benchmark for large language models](#). ArXiv:2402.12659.

Frank F. Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Z. Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, Mingyang Yang, Hao Yang Lu, Amaad Martin, Zhe Su, Leander Maben, Raj Mehta, Wayne Chi, Lawrence Jang, Yiqing Xie, and 2 others. 2024. [TheAgentCompany: Benchmarking LLM agents on consequential real world tasks](#). ArXiv:2412.14161.

John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. 2024a. [SWE-bench multimodal: Do AI systems generalize to visual software domains?](#) ArXiv:2410.03859.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. 2025. [SWE-smith: Scaling data for software engineering agents](#). ArXiv:2504.21798.

Xiao Yang, Kai Sun, Hao Xin, Yushi Sun, Nikita Bhalla, Xiangsen Chen, Sajal Choudhary, Rongze Daniel Gui, Ziran Will Jiang, Ziyu Jiang, Lingkun Kong, Brian Moran, Jiaqi Wang, Yifan Ethan Xu, An Yan, Chenyu Yang, Eting Yuan, Hanwen Zha, Nan Tang, and 8 others. 2024b. [CRAG – comprehensive RAG benchmark](#). In *Advances in Neural Information Processing Systems 37 (Datasets and Benchmarks Track)*.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2024. [τ-bench: A benchmark for tool-agent-user interaction in real-world domains](#). ArXiv:2406.12045.

Jie Zhu, Yimin Tian, Boyang Li, Kehao Wu, Zhongzhi Liang, Junhui Li, Xianyin Zhang, Lifan Guo, Feng Chen, Yong Liu, and Chi Zhang. 2026. [FinMCP-bench: Benchmarking LLM agents for real-world financial tool use under the model context protocol](#). ArXiv:2603.24943.

A A Worked Example: The Fee-Cap Task Family

This appendix shows the running example of Figure 2b end to end, quoting the dataset and the grading harness verbatim along with English translations. The decision at its center is the per-cycle fee cap: status omitted, so the rule is absent from the shipped code and data, and a single set of approval minutes is its only home. Everything under “hidden” below is grader-side material the agent never sees.

A.1 The oracle entry (hidden)

The frozen registry entry, abridged. Its visibility value is shown under the merged class name used in this paper; the released registry writes this class as source-invisible (Table 2, footnote):

```
D-0701:
title: per-cycle cap ¥50,000
rule: total over-limit fees per cycle
      capped at ¥50,000; set CAP-FLAG
start_behavior: NO per-cycle cap of any
               kind; fees accumulate uncapped and
               CAP-FLAG is never set
status: omitted
visibility: doc-authoritative
home_docs: [MINUTE-R07-031]
implements_in: {KZ115B: cap-application}
```

A.2 The retrieval task

Task S1-F4 gives the agent only this prompt:

サイクルエンドのover-limit fee engine が、承認済みの最新fee policy と合っているか確認してください。コードと業務文書を突き合わせ、必要なルール差分と修正対象の挙動を特定して回答してください。

(Check whether the cycle-end over-limit fee engine is consistent with the latest approved fee policy. Cross-check the code against the business documents, and identify the required rule differences and the behavior that must be fixed.)

A.3 The decisive document passage

The only place the cap exists is one passage in the minutes of a fee-regulation working group (in the benchmark data as DOC-56835.pdf, in-text document code MIN-CRD-R07-019), abridged:

議題2について、規制コミットメント対応として超過手数料にサイクル単位の上限を設ける旨を決定した。上限額は1サイクルあたり合計¥50,000 ←確定(手書)とし、料率計算結果がこれを超える場合は上限で頭打ちする。免除口座(料率0グループ)には上限を適用しない。本決定は決裁番号D-0701 / 稟議-R07-031として起案。実装はKZ110B が呼ぶ上限適用サブKZ115B に集約し、KZFEEHF のCAP-FLAG を立てる。上限額そのものは料率テーブル(KZDGRPF)にも口座データ(KZACCTF)にも保持しないため、本議事録が確定根拠となる。

(On agenda item 2, it was decided to introduce a per-cycle cap on over-limit fees as part of the regulatory-commitment response. The cap shall be a total of ¥50,000 per cycle (handwritten annotation: “confirmed”); where the computed fee exceeds it, the fee is truncated at the cap. Exempt accounts (rate-0 groups) are outside its scope. The decision is filed as approval number D-0701 / ringi R07-031. Implementation shall be concentrated in the cap-application subroutine KZ115B called by KZ110B, setting CAP-FLAG on KZFEEHF. As the cap amount itself is held

neither in the rate table (KZDGRPF) nor in the account data (KZACCTF), these minutes are the authoritative basis.)

A.4 The answer key (hidden)

Task S1-F4’s key, verbatim. Correctness points:

- D-0701 requires total over-limit fees per cycle to be capped at ¥50,000.
- When the per-cycle cap is applied/reached, CAP-FLAG must be set.
- The current shipped behavior is incorrect because over-limit fees accumulate without any per-cycle cap and CAP-FLAG is never set.

Completeness points:

- Identifies 議事録MIN-CRD-R07-019 (稟議-R07-031) as the authoritative source for the approved fee policy/card-network commitment.
- Explains that the cap is per cycle on total over-limit fees, not a per-transaction or per-account-lifetime limit.
- Mentions the implementation must clamp/suppress additional cycle over-limit fee accumulation once the cap is reached while preserving the flag for downstream processing.

A.5 The judge prompt

Each answer is graded in a fresh judge session shown only the question, the numbered key points, and the answer—never the condition that produced it. The instructions, verbatim (JSON-format mechanics omitted):

You are an impartial grader. You are given a question, a golden reference split into NUMBERED bullets, and an actual LLM answer. For each numbered bullet, decide whether the actual answer addresses that point. Scoring rules: 1 means the actual answer covers the bullet, even partially or with paraphrase / synonyms. Credit substance over form. 0 means the actual answer does not cover the bullet, or covers something opposite. Do NOT penalize the answer for going beyond the golden reference. Only score the listed bullets. Do NOT speculate about intent. If the bullet says “cites D-0457 by id” and the answer never contains “D-0457”, score 0 even if the answer talks around the same idea.

The judge returns a strict-JSON 0/1 score with evidence per point, referencing points by index; Equation 1 averages them.

A.6 The build task and its test cases

Task S1-F6 targets the same subsystem but is graded by execution. Its prompt:

サイクルエンドの与信超過手数料計算が、最新の承認済み手数料ポリシーと合っていないとの指摘があります。関連する設計資料・議事録を確認し、該当するパッチ／サブルーチンを修正して、免除条件、上限判

Test	Decision	Expected observable outcome
C1	D-0772	STD0 account: fee = 3,600 (post-revision rate 0.0180)
C2	D-0612	PREM account: fee = 3,600 (standard rate; no PREM-specific rate)
C3	D-0701	fee = 50,000 and CAP-FLAG = Y (cap reached)
C4	D-0457	rate-0 group: fee = 0, EX-EMPT = Y
C5	D-0772	GLD1 account: fee = 1,200 (preferential rate 0.0120)
C6	D-0733, D-0701	fee = 1,800 (floor of 1,800.90) and CAP-FLAG = N (under cap)

Table 5: The six test cases of build task S1-F6. Each asserts an observable outcome of one decision on a distinct fixture account, never an implementation detail.

定、手数料額が承認済みルール通りになるよう実装してください。

(It has been reported that the cycle-end over-limit fee calculation does not match the latest approved fee policy. Review the relevant design documents and minutes, fix the affected batch programs/subroutines, and implement the exemption conditions, the cap check, and the fee amounts so that they follow the approved rules.)

The fix path spans KZ115B, KZ120S, and KZ125S (called from the KZ110B driver, which needs no change) and the task turns on five registered decisions at once (the cap D-0701; a rate revision D-0772 whose post-revision column is authoritative; the deprecation of the PREM tier D-0612; rate-0 exemption D-0457; and yen flooring D-0733). The harness compiles the agent’s edited workspace, runs it against a fixed fixture, and checks the six test cases of Table 5.

B The Contracts File (Excerpt)

contracts.yaml is a scenario’s consistency anchor (Section 4): programs, documents, and test fixtures are all authored against it, so none of the three can free-invent a field name, a record layout, or an enum value. The excerpt below is from the fee scenario of Appendix A, abridged, with Japanese labels omitted.

Record layouts. Dataset schemas are rendered *deterministically* into the shared layouts every program includes—COBOL copybooks, C structs, Java records:

```
datasets:
  KZACCTF: { org: VSAM-KSDS, prefix: AC,
            copybook: KZACCTC,
```

```

fields: [ACCT-ID, CARD-NO, GROUP-CODE,
        CYCLE-BAL, CREDIT-LIMIT, OVER-AMT,
        KYC-STATUS] }
KZDGRPF: { org: VSAM-KSDS, prefix: DG,
  copybook: KZDGRPC,
  fields: [GROUP-CODE, OL-FEE-RATE,
    OL-FEE-RATE-OLD, EXEMPT-FLAG],
  note: "OL-FEE-RATE = post-revision;
    OL-FEE-RATE-OLD = superseded (D-0772).
    Authority is doc-only." }

```

The data domain. The layouts fix the *shape* of the data; the data domain fixes its *contents* to keep it consistent across generated source code and documents.

```

data_domain:
  group_codes:          # PIC X(04)
    - { code: STD0, tier: standard,
      rate_old: "0.0150",
      rate_current: "0.0180",
      exempt: false }
    - { code: GLD1, tier: preferential,
      rate_old: "0.0100",
      rate_current: "0.0120",
      exempt: false }
    - { code: PLT2, tier: preferential,
      rate_old: "0.0080",
      rate_current: "0.0090",
      exempt: false }
    - { code: EXMP, tier: exempt,
      rate_old: "0.0000",
      rate_current: "0.0000",
      exempt: true }
    - { code: PREM, tier: deprecated,
      rate_old: "0.0080",
      rate_current: "0.0090",
      exempt: false,
      note: "D-0612 deprecated;
        maps to STD0. Starting
        KZ120S keeps a remnant." }
  standard_code: STD0
  cap_amount_yen: 50000      # D-0701, doc-only

```

Three properties of the benchmark are legible in this one block. The `rate_old/rate_current` pair is what makes the rate revision D-0772 document-resident: *both* values sit legitimately in the rate table in the final benchmark, and only the rate sheet says which is in force—and it says so through document structure, in a merged spreadsheet cell (class `doc-structural`), which is why test cases C1 and C5 of Table 5 expect the post-revision 0.0180 and 0.0120 rather than the values the start code reads. PREM is a deprecated tier the start code still carries as a remnant (D-0612, *superseded*), so a correct fix removes logic instead of adding it. And `cap_amount_yen` is registered here but written into no dataset and no program: it is the doc-authoritative fact of Appendix A.1, and the value appears in the shipped artifacts only inside the approval minutes of Appendix A.3.

Call signatures. Every CALL uses a registered linkage contract, pinned into both the golden and the start skeleton. The two versions of a program are therefore interface-compatible, and the neighbor programs grown around them call it correctly:

```

call_signatures:
  KZ115B:          # cap-application
    using: LK-CAP-PARM
    params:
      - { name: LK-FEE-YTD,
        pic: "PIC S9(11)V99 COMP-3" }
      - { name: LK-FEE-CAPPED,
        pic: "PIC S9(11)V99 COMP-3" }
      - { name: LK-CAP-FLAG,
        pic: "PIC X(01)" }

```

C The Visibility Firewall (Excerpt)

The firewall (Section 4) asserts that each decisive fact appears only where its visibility class permits. Checks are declarative and run deterministically. A check pairs the *value* with the *context* that makes it decisive, within a line window, and may name a *neutralizer*—a pattern whose presence marks a hit benign. The Family A check for the fee cap of Appendix A, verbatim:

```

{"id": "S1-cap",
 "fact": "per-cycle fee cap 50,000 yen",
 "code": [r"(?<!\d)50000(?!\d)",
  r"(?<!\d)50,000(?!\d)"],
 "ctx": [r"過枠|手数料|FEE",
  r"上限|キャップ|CAP"],
 "window": 1,
 "home": ["s1-fees"], "code_ok": []}

```

The check fails if any program in the merged codebase carries the value with both a fee word and a cap word within window lines of it; separately, it reports which scenarios' documents carry the fact, which must be only its home. `code_ok` names scenarios where the same token is legitimate rather than a leak.

Family B checks run in the opposite direction, over the document corpus. In the overseas-card scenario the decisive rule spans two languages—RPG assigns the transaction class C2 to overseas-ATM cash, and the COBOL that handles C2 accrues interest from the withdrawal date with no grace period—so the check enumerates the phrasings that would give this away (区分はC2, TXN-KBN=C2, 引出日から利息, 無利息期間なし, “no grace period”, ...) and fails if any appears in a document seed or in a rendered document. Without it, a document-bearing condition could answer a cross-language code-tracing task without reading either program, and the scenario would silently stop measuring what its visibility label claims.