PM BENCH: VERIFIABLE CLARIFICATION BENCHMARK FOR CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

As coding agents take on increasingly autonomous software engineering, a critical capability emerges: reliably and consistently resolving incomplete and contradictory specifications. We argue that this is a fundamental capability that requires objective ground truth that is not a matter of judgment. We introduce PM Bench, a benchmark that measures two complementary capabilities: *detection*, a metric for asking the correct clarification that resolves the issues injected into a complete specification, and *restraint*, a metric for not manufacturing clarification questions about a complete specification. Our central contribution is a fully automated, formalism-first construction in which a computer science formalism, rather than natural language, is the source of truth across both task generation and verification. We begin from a certified, well-specified statechart, plant controlled defects (e.g., an unreachable state, conflicting transitions, a counter overflow), and render the result into a Product Requirements Document that reads as complete prose. Because the answer key is the planted defects, scoring is verifiable: a hidden judge checks each clarifying question against the formal oracle facts. Evaluating eleven models, we find that detection is near the ceiling for frontier models while restraint is not: models are separated almost entirely by their false-alarm rate on complete specifications. We further find that the choice of scoring metric reorders the middle of the roster. A paired pilot shows that re-rendering the same certified chart as human-style prose leaves detection unchanged but erodes restraint in frontier models. The method is demonstrated on controller statecharts but generalizes to any domain expressible as a formal specification.

1 INTRODUCTION

Turning a design specification into software is rarely a matter of translation alone: real specifications are often incomplete, ambiguous, or internally contradictory, and a competent engineer is expected to notice this and ask before building. As language models take on increasingly autonomous software engineering (Jimenez et al., 2024; Yang et al., 2024), this expectation shifts to agents, and they often fall short. They can be sycophantic (Sharma et al., 2024) or take liberties without asking (Ruan et al., 2024; Ji et al., 2026; Qu et al., 2026), silently implementing something other than what was intended; and when they do ask, their questions are often superficial or unrelated to what the implementation actually depends on (Wu & Fard, 2025; Mu et al., 2023). If agents are to work from product requirements written by program managers (PMs), they must ask when a load-bearing fact is missing or contradictory, and stay quiet when the specification is complete.

Measuring this is harder than it appears. Existing clarification benchmarks degrade natural-language problems by hand or with an LLM and judge the agent's questions with another LLM or a human rater (Section 2). Neither the ground truth (what exactly is missing) nor the grade (whether the agent found it) is then objective. Most also test only whether the agent asks, not whether it refrains from asking when nothing is wrong.

We take a formalism-first approach, following model-driven engineering, where formal models are created first and remain the source of truth throughout development (Alfraihi & Lano, 2025; 2023; Sztipanovits & Karsai, 1997; Sztipanovits et al., 2012). (To avoid clashing with "model" in the machine-learning sense, we call these engineering models *formalisms*.) We build on statecharts (Harel, 1987), a formalism whose properties, such as reachability, determinism, and termina-

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

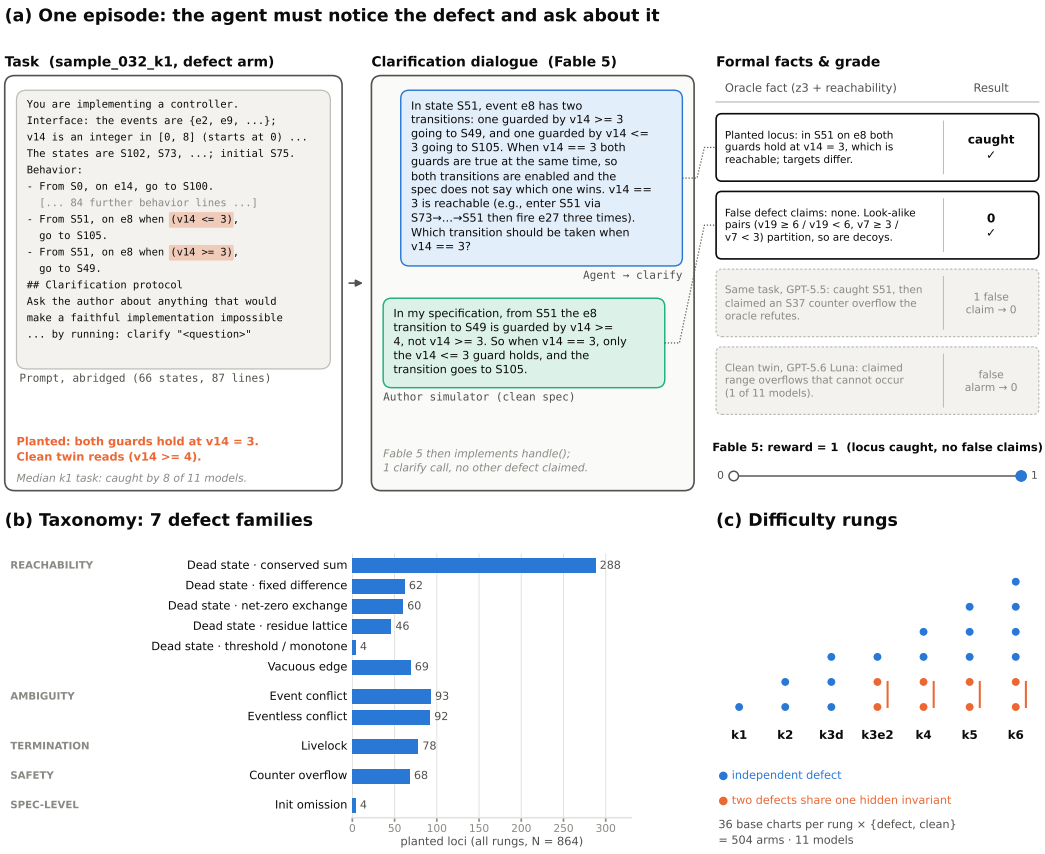


Figure 1: Scope and taxonomy of the benchmark. (a) One episode on task `sample_032_k1` (rung k1, event conflict; Appendix A), left to right: the prompt, abridged, with the planted defect highlighted (in state `S51`, event `e8` has two transitions whose guards both hold at `v14 = 3`); Fable 5's clarification question and the author simulator's answer from the clean specification; and the grade, which checks each claimed defect against the formal facts. Fable 5 catches the planted defect and makes no false claim, so its reward is 1; the greyed rows show a false claim by GPT-5.5 on the same task and a false alarm by GPT-5.6 Luna on the clean twin, each scoring 0. (b) Planted defects by type. The benchmark has seven defect families (Table 1); the five dead-state bars are subcategories of the single dead-state family. (c) The seven difficulty rungs.

tion, can be decided mechanically, much as constraint satisfaction problems allow underspecification to be defined precisely (Li et al., 2025). We generate a statechart, certify that it is well specified, and then plant defects with small, recorded edits that each break one formal property: an unreachable state, two transitions enabled at once, a counter that leaves its range, and others (Table 1). Undoing the edits yields a certified-clean twin. The author's answers to clarifying questions come from this twin, so an agent that asks about a defect is led back to the original, well-specified statechart. Both charts are rendered into the specification the agent reads, and the agent may ask clarifying questions before implementing. Because every defect is a recorded formal edit, the answer key is known before the agent runs, and an agent's claims are checked by deterministic code against the formal facts rather than by an LLM's or a human's reading of the dialogue (Figure 1).

Our contributions are:

- **A formalism-first construction for clarification benchmarks**, in which a certified formal model is the source of truth for task generation, the answer key, and grading.
- **PM Bench**, a benchmark of 252 base tasks across seven difficulty rungs and seven defect families, with 504 specifications (a defect arm and a certified-clean twin per task) and 864 planted defects.

- **A verifiable grading protocol** that separates *detection* of planted defects from *restraint* on complete specifications, in which an LLM judge only extracts claims and code decides whether each claim is true.
- **An evaluation of eleven models**, together with a paired pilot that re-renders the same certified charts as verified human-style prose.

Detection is near the ceiling for frontier models, but restraint is not: the leaders are separated almost entirely by how often they flag a defect in a complete specification, which ranges from about 3% to 43%. The ranking in the middle of the roster depends on the scoring metric, and in the pilot, human-style prose leaves detection unchanged but lowers restraint for the frontier models.

2 Related Work

A first line of work degrades otherwise-complete coding problems and asks whether a model recovers the missing information. HumanEvalComm (Wu & Fard, 2025) injects inconsistency, ambiguity, and incompleteness into HumanEval (Chen et al., 2021) problem descriptions; ClarifyGPT (Mu et al., 2023) instead infers underspecification indirectly, using generated tests over candidate outputs to decide when a task is ambiguous. More recent efforts scale and diversify the injection but keep it and its validation informal: ClarEval (Li et al., 2026a) curates 750 problems from HumanEval and LiveCodeBench and uses GPT-4o to inject three ambiguity types (missing goals, missing premises, ambiguous terminology), validated by two-stage human review; Ambig-SWE (Vijayvargiya et al., 2026) condenses fully specified SWE-Bench Verified tasks into underspecified versions with GPT-4o and recovers the differences with an LLM, starting from an existing benchmark of natural-language tasks rather than generating the fully specified task itself. Across this group the deficiency is introduced by hand or by LLM rewriting and the judgment of whether it was noticed is made by a model or a rater. We differ by construction: the ill-specification is injected by a controlled edit to a verified formalism, so it is known exactly, and grading is disposed by code against the formal oracle rather than by an LLM's or a human's reading.

A second line imposes more structure on either the deficiency or the artifact. QuestBench (Li et al., 2025) formalizes underspecification as a constraint satisfaction problem in which exactly one necessary variable assignment is missing, and asks whether a model can identify the single question that recovers it; we adopt its terminology where possible, although our formalism is the statechart, our domain is code generation, and our notion of ill-specification goes beyond 1-sufficient underspecification to include misspecification (conflicts, dead states, overflow, and the other families of Table 1). SpecBench (Hamblin et al., 2026) likewise targets specification deficiencies and, similarly to our work, categorizes them under the IEEE Std. 1028-1997 anomaly classes (omission, ambiguity, inconsistency, incorrectness), but derives its ground truth from real-world RFC and GitHub threads filtered by a panel of LLM judges; ours is derived directly from a formal specification rather than from historical expert discussion. PRDBench (Fu et al., 2026) uses coding agents to generate Product Requirements Documents from seed tasks, paralleling our rendering of a PRD from a statechart, though as a generation target rather than a verified source of truth. Asuka-Bench (Wang et al., 2026) pairs underspecified user intent with multi-round refinement and grounds evaluation in the browser-observable behavior of a generated website. We differ by anchoring both the target and the grade in a pre-specified formalism: where these benchmarks obtain structure from a CSP encoding, expert discussion, a generated document, or post-hoc observed behavior, our missing fact, its location, and the clarifying question that recovers it are all fixed before the agent is run, and the clean twin is proved well formed by the same checks.

Statecharts are a long-established formalism (Harel, 1987; Object Management Group, 2017; The MathWorks, Inc., 2024), used across embedded systems (Lee & Seshia, 2017), autonomous driving (Wang et al., 2021), engine control, UI state management (e.g., Maraev et al., 2025), network protocols (von Bochmann, 1978), biology ("biocharts") (Kugler et al., 2010), business workflows (Mok & Paper, 2002). Because our construction depends only on the formalism and not on the software controller domain we instantiate, each of these is a domain our approach extends to directly.

Closest to our formalism, OSVBench (Li et al., 2026b) also builds on statecharts and uses Z3 to decide statechart equivalence. Its aim, however, is to generate a specification for downstream verification; we invert this use of the formalism, treating the statechart as the source of truth that precisely

and formally defines the problem the coding agent must resolve, and using solver-verified edits to plant a known missing fact rather than to certify a produced spec. To our knowledge, using solver-verified edits of a formal model to generate clarification items with a known answer key is novel.

We are not aware of any prior benchmark that treats a formalized representation of the problem as the ground-truth source for both specification and verification, using it as the generative entry point from which every task artifact is derived through the generation and verification pipeline.

3 Problem Formulation

3.1 Statecharts

We introduce the simplest formal description of statecharts to state the introduced misspecifications precisely. We follow UML 2.5 statechart semantics (e.g. run-to-completion processing) (Object Management Group, 2017). Although we believe benchmarks with richer statechart semantics are a promising research direction, they are beyond the scope of this paper. We use (i) states, (ii) transitions, (iii) guard conditions and (iv) actions on transitions; as well as (v) a context with a set of variables. The actions only change a variable.

A statechart and each of its transitions are defined as

$$\mathcal{SC} = (\mathit{States},\ \mathit{InitialState},\ \mathit{Events},\ \mathit{Context},\ \mathit{Transitions}),$$
$$t = (\mathit{SourceState},\ \mathit{TriggerEvent},\ \mathit{Guard},\ \mathit{Action},\ \mathit{TargetState}) \in \mathit{Transitions},$$

where States is a finite set of states, $\mathit{InitialState} \in \mathit{States}$ is the state the machine starts in, Events is the set of event types the machine can receive, each possibly carrying typed parameters, $\mathit{Context}$ is a finite set of typed variables forming the machine's attributes, each integer variable v having a declared range $[lo_v, hi_v]$ and an initial value in that range, and $\mathit{Transitions}$ is the transition relation. A transition connects $\mathit{SourceState}, \mathit{TargetState} \in \mathit{States}$: on occurrence of $\mathit{TriggerEvent} \in \mathit{Events} \cup \{\varepsilon\}$, if Guard holds, it executes Action and moves to $\mathit{TargetState}$. ε denotes an eventless transition (with $\mathrm{Params}(\varepsilon) = \emptyset$), which becomes eligible as soon as its source state is active. $\mathit{Guard} : \mathrm{Val}(\mathit{Context}) \times \mathrm{Params}(\mathit{TriggerEvent}) \to \{\mathit{true}, \mathit{false}\}$ is a boolean-valued predicate over the current context and the incoming event's parameters; the transition is enabled only if it evaluates to true at the moment $\mathit{TriggerEvent}$ occurs (for ε, whenever the source state is active). $\mathit{Action} : \mathrm{Val}(\mathit{Context}) \times \mathrm{Params}(\mathit{TriggerEvent}) \to \mathrm{Val}(\mathit{Context})$ executes only once the transition fires, updating the context.

Under run-to-completion, after each event the enabled eventless transitions fire repeatedly until none is enabled, before the next event is accepted.

A *run* is an execution of the state machine — a concrete sequence of transitions actually fired, in order, starting from the initial state. A *feasible path* is a path of transitions for which there exists a sequence of events — with parameter values satisfying each guard in turn — that the state machine can actually execute as a run.

A *configuration* of a state machine is a complete snapshot of its current situation: the active state, together with any variable or counter values the machine tracks.

3.2 Ill-Specification

We show a few examples for the ill-specification category; the complete list can be found in Table 1 and viewed in Figure 5. The simplest one is a dead state (ds): there is no feasible path to ds. This is a well-formed statechart, but it is not a valid controller specification. It could be underspecified (e.g. missing transition), or misspecified (e.g. a mistake in the expression of a guard condition).

If two different transitions become enabled from a state due to the same event (or automatically for ε), they are called conflicting transitions, and the execution is nondeterministic (in practice the particular implementation of the event loop will pick one). This is clearly an undesired behavior for a controller.

The task for the agent is to obtain all information to resolve the issues with the ill-specified statechart.

Table 1: Defect families. Each is planted as a minimal corruption of a certified-clean chart; the dead-state row spans six families (a threshold anchor and five invariant shapes) so the surface pattern cannot be memorized.

Family	**Breaks**	**What is modified**	**IEEE 1028-1997**
dead state ($\times 6$)	reachability	a guard demands a configuration that can never hold: either a threshold anchor is broken directly, or the counters obey a hidden invariant (conserved sum, fixed difference, residue lattice, monotone growth, net-zero exchange) that forbids a feasible path; the target state can never be entered	Incorrectness, Omission
vacuous edge	reachability	a guard that is never true on the reachable region; source and target both are reachable via other paths	Incorrectness
eventless conflict	determinism	two completion (automatic) transitions enabled at the same time, targets differ (nondeterminism)	Ambiguity
event conflict	determinism	two transitions enabled at the same time on the same event whose guards overlap, targets differ	Ambiguity
livelock	termination	an eventless infinite loop with conditions and actions such that it never terminates and the counter value remains bounded	Incorrectness
overflow	safety	a self-loop pump whose guard is loosened from $c < M$ to $c \leq M$, letting a counter escape its declared range	Incorrectness
init omission	spec-level	the English specification omits an initial value that reachability depends on	Omission

4 METHOD

Benchmark tasks are generated through a multi-step procedure with verifiable endpoints to ensure information preservation.

4.1 TASK CONSTRUCTION

Each task starts with generating a ground-truth statechart which is rendered into a product requirements document (PRD) (Figure 2). The rendering step is checked by three independent parsers that confirm the original statechart construction. The statechart is then modified into an ill-specified one, producing the defect arm and an answer key naming what was modified. The agent sees the PRD rendered from the ill-specified statechart and may call `clarify`; grading is described in Section 4.2.

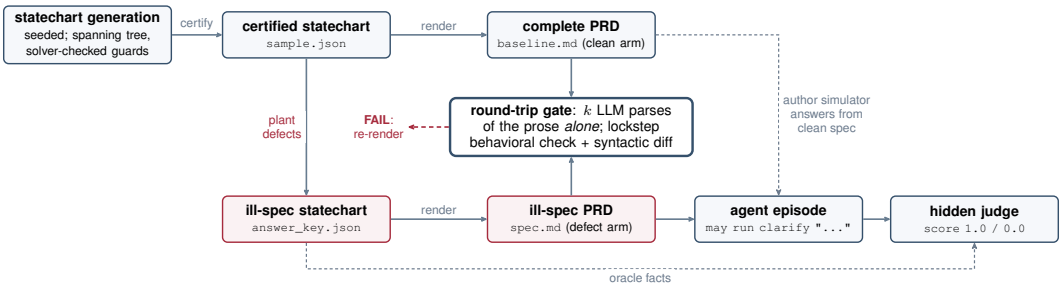


Figure 2: Construction of a task. A seeded generator builds a statechart, which is certified to be well-specified and rendered as a complete PRD (the clean arm); planting defects yields the ill-specified statechart and its answer key, which is rendered as the ill-specified PRD (the defect arm). Each PRD must pass the round-trip gate before use; the author simulator answers from the clean spec, and the judge checks claims against the oracle facts.

Main-campaign tasks use the deterministic controller template (Appendix A shows one); the box below shows the same construction rendered as PM prose, from the paired pilot described later in this section.

An example task from the PM-prose pilot — work-cell controller · k=1, family *overflow*

What the agent sees. One file: an engineer's design notes (518 words, 8 modes, 5 signals, 3 counters), followed by the fixed clarification protocol. Abridged, with the two planted lines accented:

```
# Work cell control, my notes from the whiteboard session

Writing this down while it is still fresh so I do not have to reconstruct the cell
behavior off a phone photo in three weeks.  [...]

Three machine registers get tracked.  Counter_Feeder_PartIndex runs 0 through 4 and
starts at 0.  [...]  Counter_Conveyor_CycleCount is the narrow one, 0 through 2,
starting at 0.

[...]
Once paused, Signal_Drive_MaterialAvailable adds one to Counter_Conveyor_CycleCount
and keeps the mode at Feeder_Paused.  The guard admits the register at 2 as well as
anything below it.

[...]
## Clarification protocol

Before implementing, audit the specification and decide whether it is complete and
consistent.  [...]  If you find a real problem, ask the specification's author by
running:
    clarify "<your question>"
[...]  If the specification is complete and consistent, do not ask about problems it
does not have -- implement directly.
```

Why that is a defect. At `CycleCount` = 2 the guard still admits the increment, which drives the counter to 3, outside its declared range. The oracle records this as a *domain escape* at `Feeder_Paused`. The clean twin differs in exactly these two lines (range 0–4; guard strictly below 2) and is certified pathology-free.

Result (one attempt per model, same task).

- **Fable 5 — reward 1.** Asked: *"…the guard admits the register at 2 as well as anything below ($\leq$ 2). But Counter_Conveyor_CycleCount is declared with range 0 through 2, so firing at 2 would drive it to 3, outside its range. Should the guard be strictly below 2, or is the range wrong?"* The simulated author replied from the clean twin: *"The guard is strictly below 2, not 2 or below. Also, Counter_Conveyor_CycleCount is declared with the range 0 through 4, not 0 through 2."* Judge: one planted defect caught, no false claims.
- **GPT-5.6 Sol — reward 1.** Same catch, no false claims.
- **Nemotron 3 Ultra — reward 0.** Caught the overflow, but also asserted six defects the oracle refutes (e.g. that `Station1_Loading` and `CellA_Clamping` "can never leave"; they are absorbing by design, not unreachable). Each false claim is checked against the oracle facts in code and cancels the reward.
- **Qwen3-30B — reward 0.** Asked nothing and implemented the spec as written.

Every task starts from a statechart that is proved to be well specified before any defect is added. The generator builds each chart around a spanning tree, so every state is structurally reachable; it gives each state distinct outgoing events, so the chart is deterministic by construction; and it adds back-edges and guarded transitions only when a solver confirms that the guard can actually become true. The charts in the main campaign have about 50 states, a handful of bounded integer counters and a median of 92 transitions. A chart is accepted only if it passes a set of certification checks, and the checks export a table of formal facts about it (which states are dead, which transitions never fire, which counters can overflow, which configurations enable two transitions at once). The grader later reads this table instead of re-running the solver. Generation is seeded, so rebuilding a chart from its seed produces the same bytes. Because the starting chart is certified, anything wrong with the final task was put there on purpose.

A defect is planted by a small edit that keeps the chart well formed but breaks exactly one formal property (Table 1). Each edit is recorded together with how to undo it, and the recorded edits become the task's answer key. The injectors follow one rule: they may only *select* edits that already look like ordinary parts of the chart. A planted guard uses the same variables, constants, and number

of conditions as the guards around it, and a candidate with any unusual surface feature (a constant outside the counter's range, a comparison between two variables) is rejected. As a result, a defect cannot be spotted by its appearance; the agent has to work out what the chart actually does.

Tasks are grouped into seven difficulty *rungs* that vary two things: how many defects are planted and whether they share a cause. Rungs K1 and K2 plant one and two independent defects. K3D plants three unrelated defects, while K3E2 plants three defects of which two are *entangled*: they are two symptoms of one hidden invariant. For example, if every transition that touches three counters moves value between them, their sum never changes; two guards that each require a larger sum then make two different states unreachable for the same reason. Rungs K4–K6 add a fourth, fifth and sixth defect while keeping one entangled pair. Each rung has 36 base charts, each with a defect arm and a clean twin, so the ladder has 252 base tasks, 504 arms and 864 planted defects.

Every defective chart has a *clean twin* that differs from it only by the planted edits. The twin is built by undoing each recorded edit, and it is then certified again from scratch rather than assumed to be clean. In the controller example in Appendix A, for instance, the two arms differ in a single token (`v14 >= 3` against `v14 >= 4`). The twin gives the benchmark its restraint test: an agent that reports a defect on the clean arm is reacting to the look of the specification, because nothing is broken.

Both charts are then rendered into the specification the agent reads. The main campaign uses a deterministic template that writes one plain-English sentence per transition (for example, *"From S51, on e8 when (v14 >= 4), go to S49."*), so the text states the chart exactly and the defect is always derivable from it but never pointed out. The human-style PRD rendering of Figure 2 replaces this template with LLM-written prose and is checked by the round-trip gates described in Appendix C; we evaluate it separately in Section 5 (Figure 3). In both renderings, we know exactly which fact is broken, where it is, and what the author's answer is before any agent is run.

The main campaign renders every chart with the deterministic controller template. To test whether the measurement transfers to the product prose the pipeline is meant to produce, we took the compact ladder ($k \in \{1, 3, 4, 6\}$, up to 24 base charts per rung, 5–14 states). A human-style PRD rendering is accepted only if it passes a round-trip check in which three independent parsers per arm reconstruct the machine from the prose. Every reconstruction must be structurally exact and lockstep-bisimilar to the certified chart; the parsers must agree with each other; the defect must survive against the clean twin; twin drift is bounded; and the oracle facts recomputed on every reconstruction must be byte-identical to the certified ones.

4.2 GRADING

Grading divides the work between an LLM and code: the LLM reads the conversation, and code decides the reward. After each episode, an LLM judge (Opus 4.8, temperature 0) receives the clarification conversation together with the task's formal facts. It returns whether the planted defect was identified, a verbatim quote supporting that verdict, and a list of every state, transition, counter or initial value the agent described as broken. Deterministic code then checks each claimed defect against the full table of what is actually broken in the chart. A claim that matches nothing in that table is a *false claim*. Matching is exact at the token level, so a real problem in `S2` cannot excuse a claim about `S22`. The judge therefore extracts claims but does not decide whether they are true; the formal facts do.

On a defect arm the reward is 1 only if the conversation identifies *every* planted defect and makes *no* false claim; otherwise it is 0. For a task with K planted defects the judge is called once per defect, and each call is told about the other defects so that correctly raising a second defect is not counted as a false claim against the first. On a clean arm the reward is 1 unless the agent claims a defect. Ordinary implementation questions are never penalized, so restraint means not asserting a defect that does not exist; it does not mean staying silent. We chose a binary reward over partial credit because an earlier signed version with partial credit penalized correct but wordy questions and encouraged agents to list many possible problems in the hope of scoring.

The `clarify` command is answered by a separate LLM acting as the specification's author. It answers only from the clean specification, in one or two sentences, never volunteers a defect, and

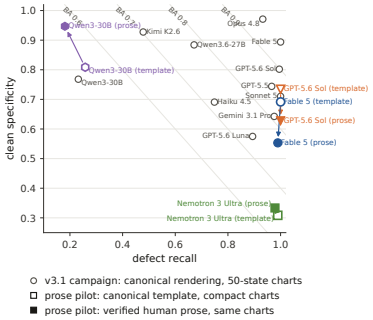


Figure 3: Defect recall against clean specificity, one point per model and surface. Grey rings are the eleven models of the main campaign on the canonical rendering of the 50-state charts (common block, one attempt per task). Colored markers are the prose pilot on the compact charts: open marker on the canonical template, filled marker on verified human prose, arrow from template to prose for the same base charts. The two corpora differ in chart size and family mix, so positions are comparable within a group and only indicative across groups. Diagonals mark constant balanced accuracy. Detection is near the ceiling for the frontier models on every surface; the spread, and the prose shift, are in specificity.

never argues. If the judge call still fails after retries, the episode is removed from every average and counted separately, so a grading failure never becomes a silent 0 or 1.

From these rewards we report three per-model quantities. *Detection* (recall) is the fraction of defect arms on which the agent made a specific defect claim. Because such a claim can be one the oracle refutes, we also report *oracle-matched recall*, which counts a defect arm only if at least one planted defect was caught, as verified against the oracle facts. On the 207-task common block, substituting it changes balanced accuracy by at most 0.06 (median 0.007); the top three ranks are unchanged, and the only reorderings are two adjacent swaps (GPT-5.5 and Sonnet 5, tied at 0.855 under detection, and Haiku 4.5 and Kimi K2.6). The largest shifts are for the weakest detectors, whose positives more often rest on refuted claims alone. *Restraint* (specificity) is the fraction of clean arms on which the agent made no defect claim. *Balanced accuracy* (BA) is their mean, and it is our primary metric because it weights both behaviors equally and a model cannot raise it by always asking or never asking. *Exact success* is the stricter task reward above, which requires every planted defect and no false claim; on multi-defect tasks it can be much lower than detection, so we report it separately.

All-defects recall is the fraction of defect arms on which the agent identified every planted defect, regardless of false claims; exact success additionally requires that no false claim is made.

5 Results

Across the eleven models, what separates the benchmark's leaders is restraint, not detection. The strongest models identify a planted defect on nearly every defective specification but differ widely in how often they flag a clean one (Figure 3): Opus 4.8 wrongly reports a defect on about 3% of clean specifications and GPT-5.6 Luna on about 43%. Catching *every* planted defect in a task (Figure 4) is much harder than catching one: Fable 5 does so on 98% of defective tasks and GPT-5.6 Sol on 84%, while the remaining models stay at or below 66%, and the middle of the ranking changes depending on which of these metrics is used (Figure 9). Requiring, in addition, that the agent make no false claim (exact success) lowers these rates further, to 89% for Fable 5, 70% for GPT-5.6 Sol, and 25% for Opus 4.8 on the same 207 tasks, and for different reasons across models: false claims account for most of Fable 5's shortfall from detection to exact success (about 83%) and about half of GPT-5.6 Sol's (46%), but almost none of Opus 4.8's (3%), which catches at least one planted defect on 93% of defective tasks (oracle-matched) yet catches every planted defect on only 28%. The kind of defect matters more than the number planted: conflicts and livelocks are caught most often, while vacuous edges and dead states hidden behind a counter invariant are caught least often (Figure 10). Finally, rewriting the same charts as human-style prose leaves detection unchanged but lowers restraint for the frontier models (Figure 3; per-rung detail in Appendix D).

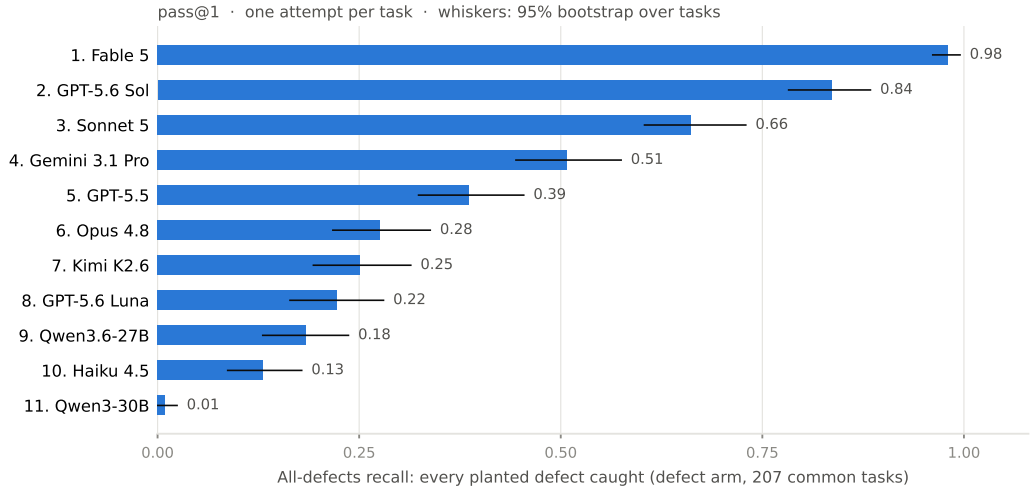


Figure 4: Leaderboard on all-defects recall: the fraction of defect tasks in which the agent caught every planted defect, pass@1 (one attempt per task), on the 207 base tasks with judge-valid results for all eleven models. Whiskers are 95% percentile bootstrap intervals over tasks (2,000 resamples). All-defects recall scores detection only; false claims and restraint on clean specifications are reported separately.

6 Limitations and Conclusions

Although we believe that the approach can be generalized to other formalisms, we have identified the following limitations. We use LLMs to convert between statecharts and prose. This does not necessarily generalize to other formalisms LLMs are not trained on. Occasionally, the generated English reads templated. The grading is dependent on a capable judge model, and the binary reward hides partial competence. More complex statechart features such as parallelism, timers, and history states are out of scope. Three further limitations concern the evaluation itself. The judge (Opus 4.8) is also one of the evaluated models; because the judge only extracts claims and code decides their truth against the formal facts, the room for self-preference is small, but we have not measured it. All models were run under a single agent scaffold with one attempt per task, so the results are conditional on that scaffold and carry per-task sampling noise. Finally, the prose pilot covers four models on compact charts without conflict families or entangled defects, so its finding is descriptive and bounds the effect of prose on the easier families only.

Constructing the benchmark from a formal model turns clarification from a judgment call into an automated measurement: the missing fact is known before the agent runs, the grade is computed against formal facts rather than opined by a judge, and not raising a false alarm becomes as measurable as catching a real one. Across eleven models, frontier systems detect nearly every planted defect, but they differ widely in how often they flag a complete specification, and human-style prose widens that gap. Because the construction depends only on the formalism, the same pipeline extends to other domains and formalisms in which ill-specification can be decided.

AI use statement

We used AI assistants in two capacities. First, AI coding agents helped implement the benchmark pipeline and analysis code and executed experiments under the authors' direction; the research questions, benchmark design, and interpretation of results are the authors'. Second, an AI assistant was used in a limited editorial capacity: refining phrasing, shortening overly long passages, and improving readability. All AI-generated code and edits were reviewed by the authors, who verified every reported number, claim, and citation.

LLMs are also components of the benchmark itself, and each use is described in the paper. The evaluated agents are LLMs; an LLM author simulator answers clarification questions from the certified-

clean specification only; an LLM judge (Opus 4.8, temperature 0) extracts claims from each dialogue, which deterministic code checks against precomputed formal oracle facts (Section 4.2); and independent LLM parsers reconstruct charts from human-style prose to verify it (Appendix C). We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

Because grades rest on formal facts, not on the judge's opinion, judge failures only remove rollouts rather than change grades: of the 5,544 rollouts on the 504 specifications (11 models), 52 (0.9%) had a failed judge call and were excluded.

Reproducibility statement

On acceptance we will release the full pipeline (generator, verifier, grader, analysis code) and all 504 evaluated specifications with their oracle facts. Because tasks are generated with certified answer keys, fresh equivalent test sets can be produced at any time to limit contamination, and we will evaluate new models on request.

References

H. Alfraihi and K. Lano. Trends and insights into the use of model-driven engineering: A survey. In *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pp. 286–295. IEEE, 2023.

H. Alfraihi and K. Lano. Report from MDE practice: An interview-based evaluation of model-driven engineering uses. *PLoS One*, 20(11):e0335461, 2025.

M. Chen, J. Tworek, H. Jun, Q. Yuan, H. Ponde de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. URL `https://arxiv.org/abs/2107.03374`.

L. Fu, B. Zhang, H. Guan, Y. Zhu, L. Qiu, W. Liu, X. Cao, X. Cai, W. Zhang, and Y. Yu. Automatically benchmarking LLM code agents through agent-driven annotation and evaluation (PRD-Bench). In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2026.

G. Hamblin, K. Song, Z. Zhu, A. Jayarajan, S. Liu, N. Vijaykumar, and G. Pekhimenko. SpecBench: Evaluating specification-level reasoning for software engineering LLM agents. *arXiv preprint arXiv:2605.30314*, 2026.

D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

Z. Ji, Z. Zhang, C. Xu, Z. Li, Y. Gao, S. Wang, and S.-C. Cheung. Coding agents are guessing: Measuring action-boundary violations in underspecified DevOps instructions. *arXiv preprint arXiv:2607.02294*, 2026.

C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2310.06770.

H. Kugler, A. Larjo, and D. Harel. Biocharts: A visual formalism for complex biological systems. *Journal of the Royal Society Interface*, 7(48):1015–1024, 2010.

E. A. Lee and S. A. Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. MIT Press, 2nd edition, 2017.

B. Z. Li, B. Kim, and Z. Wang. QuestBench: Can LLMs ask the right question to acquire information in reasoning tasks? In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2025.

J. Li, Y. Wu, and Y. Chang. ClarEval: A benchmark for evaluating clarification skills of code agents under ambiguous instructions. *arXiv preprint arXiv:2603.00187*, 2026a.

S. Li, J. Jiang, T. Zhao, and J. Shen. OSVBench: Benchmarking LLMs on specification generation tasks for operating system verification. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI'26)*, 2026b.

V. Maraev, A. Berman, and S. Larsson. Combining information state update, Harel statecharts and LLMs for controllable and flexible conversational AI. In *Proceedings of the 2025 CLASP Conference on Language Models and Representations (LARP)*, pp. 28–37. Association for Computational Linguistics, 2025.

W. Y. Mok and D. Paper. Using Harel's statecharts to model business workflows. *Journal of Database Management*, 13(3):17–34, 2002. doi: 10.4018/JDM.2002070102.

F. Mu, L. Shi, S. Wang, Z. Yu, B. Zhang, C. Wang, S. Liu, and Q. Wang. ClarifyGPT: Empowering LLM-based code generation with intention clarification. *arXiv preprint arXiv:2310.10996*, 2023.

Object Management Group. OMG unified modeling language (UML), version 2.5.1. OMG Document formal/2017-12-05, 2017.

Y. Qu, Y. Zhang, Y. Zhang, G. Deng, Y. Li, L. Y. Zhang, and Y. Liu. Overeager coding agents: Measuring out-of-scope actions on benign tasks. *arXiv preprint arXiv:2605.18583*, 2026.

Y. Ruan, H. Dong, A. Wang, S. Pitis, Y. Zhou, J. Ba, Y. Dubois, C. J. Maddison, and T. Hashimoto. Identifying the risks of LM agents with an LM-emulated sandbox. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2309.15817.

M. Sharma, M. Tong, T. Korbak, D. Duvenaud, A. Askell, S. R. Bowman, et al. Towards understanding sycophancy in language models. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2310.13548.

J. Sztipanovits and G. Karsai. Model-integrated computing. *IEEE Computer*, 30(4):110–111, 1997.

J. Sztipanovits et al. Toward a science of cyber-physical system integration. *Proceedings of the IEEE*, 100(1):29–44, 2012.

The MathWorks, Inc. Stateflow: Model and simulate decision logic using state machines and flow charts. MathWorks product documentation, 2024.

S. Vijayvargiya, X. Zhou, A. Yerukola, M. Sap, and G. Neubig. Ambig-SWE: Interactive agents to overcome underspecificity in software engineering. In *International Conference on Learning Representations (ICLR)*, 2026.

G. von Bochmann. Finite state description of communication protocols. *Computer Networks*, 2 (4–5):361–372, 1978.

X. Wang, L. Sun, Y. Zhu, S. Zhou, J. Liu, F. Chen, L. Qiu, X. Cao, X. Cai, L. Zhang, and Z. Mao. Asuka-Bench: Benchmarking code agents on underspecified user intent and multi-round refinement. *arXiv preprint arXiv:2606.05920*, 2026.

Y. Wang, Z. Liu, Z. Zuo, Z. Li, L. Wang, and X. Luo. Decision making framework for autonomous vehicles driving behavior in complex scenarios via hierarchical state machine. *Autonomous Intelligent Systems*, 1(10), 2021.

J. J. W. Wu and F. H. Fard. HumanEvalComm: Benchmarking the communication competence of code generation for LLMs and LLM agent. *ACM Transactions on Software Engineering and Methodology*, 2025. arXiv:2406.00215.

J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press. SWE-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024.

A STATECHART CONTROLLER EXAMPLE

A representative task — `sample_032_k1` · rung K1 · family *event conflict*

What the agent sees. The entire prompt is one file: the rendered specification — 66 states, 11 bounded counters, 20 events, 87 behavior lines, no diagram — followed by a fixed clarification protocol. Abridged, with the conflicting pair accented:

```
You are implementing a controller.
Interface:  the events are {e2, e9, e17, ..., e8}; the state variables are
v2 is an integer in [0, 6] (starts at 0); ... v14 is an integer in [0, 8]
(starts at 0); ... implement handle(state, event, context) -> (state, context).
Any event with no described transition in the current state is ignored (no-op).
The states are S102, S73, S100, ...; the initial state is S75.
Behavior:
- From S0, on e14, go to S100.
- From S75, on e6, go to S12.
  [... 83 further behavior lines ...]
- From S51, on e8 when (v14 <= 3), go to S105.
- From S51, on e8 when (v14 >= 3), go to S49.

## Clarification protocol

Before implementing, decide whether the specification is complete and
consistent.  Ask the specification's author about anything that would make a
faithful implementation impossible:  a state that can never be reached; a guard
that can never fire; a loop that never settles; a counter that can leave its
declared range; two transitions enabled at once with no stated tie-break; an
initial value the behavior depends on but never states.  Ask by running:

    clarify "<your question>"

You will receive a real answer.  You may call clarify again with follow-up
questions until you are satisfied.  Do not resolve a conflict yourself by list
order or "first match wins".  Keep auditing after the first problem you find.
When ready, implement handle(state, event, context) in
/app/submission/statechart_task.py.  If the specification is complete,
implement directly.
```

Why that is a defect. Both guards hold at `v14 == 3`, and the generator's solver certifies that configuration is reachable in `S51`: the specification is nondeterministic there, and nowhere else. Two other states in the same file also branch two ways at once (`v19 >= 6` against `v19 < 6`, and `v7 >= 3` against `v7 < 3`), but those guards partition — they are decoys. Noticing a duplicated pair is not enough; the guards have to be checked for overlap on the reachable region.

The clean twin. The same file with one token changed: the second guard reads `v14 >= 4`. The guards become disjoint, the chart is re-proved well formed, and an agent that raises an alarm here scores 0.

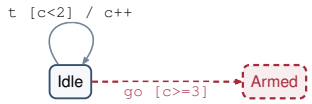
A question that scores 1 (Table 5): *"In state S51, event e8 has two transitions: one guarded by `v14 >= 3` going to S49, and one guarded by `v14 <= 3` going to S105. When v14 == 3 both guards are true at the same time, so both transitions are enabled and the spec does not say which one wins. v14 == 3 is reachable [...] Which transition should be taken when v14 == 3?"*

Difficulty rating. K1 — one planted defect. The corpus spans seven rungs, K1 through K6, up to six planted defects, entangled from K3E2 onward. Measured difficulty places this task at the median of the 36 K1 tasks: **8 of 11 models detected it** (rung K1 mean detection 0.68); on its clean twin, 1 of 11 false-alarmed.

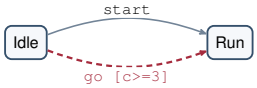
B ILL-SPECIFICATION FAMILIES AT A GLANCE

Each panel below is the smallest chart that exhibits one family. They share a deliberate skeleton — a single counter `c` declared over `[0,2]`, at most two states, and the planted element drawn dashed in red — so that the families can be read against each other rather than each on its own terms. Two pairs are worth reading together: *dead state* and *vacuous edge* plant the same unsatisfiable guard and differ only in whether the target survives by another route, and *eventless conflict* and *event conflict* plant the same guard overlap and differ only in whether the transition waits for an event.

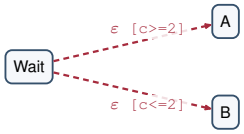
Two caveats. The *dead state* panel draws the threshold-anchor variant; the five invariant variants folded into that row of Table 1 plant the same unreachability through a hidden two-counter invariant instead of a broken threshold, and do not reduce to a two-state picture. And *init omission* is not really a chart defect at all — the chart is well formed and the omission is in the prose — so its panel annotates what the document fails to say rather than showing a broken structure.



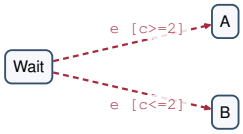
(a) dead state — `c` never exceeds 2, so the guard never holds and Armed can never be entered.



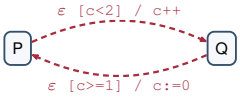
(b) vacuous edge — the same dead guard, but Run is still reachable via `start`. The edge is unfeasible, the state is reachable.



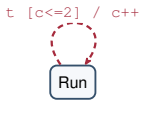
(c) eventless conflict — at `c == 2` both automatic transitions are enabled and their targets differ. No tie-break is stated.



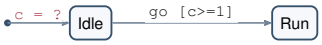
(d) event conflict — the same overlap at `c == 2`, but reached on an event rather than automatically. Only the edge labels differ from (c).



(e) livelock — an automatic loop whose counter effect nets to zero. Run-to-completion never settles.



(f) overflow — the clean guard is `[c<2]`. Loosened to `[c<=2]` the pump runs one step too long and `c` reaches 3, outside its declared range.



(g) init omission — the chart is well formed; the document simply never states what `c` starts at, and reachability depends on it.

Figure 5: One miniature per family of Table 1. Dashed red marks the planted element; everything drawn in blue is well formed. All seven share a single counter `c` declared over `[0,2]`, so the panels differ only in the mechanism being planted.

C STATECHART-TO-PRD CONVERSION IS RELIABLE

Rendering a certified chart into a PRD with an LLM is the one non-deterministic step in the pipeline. We developed an experiment to evaluate whether the conversion is reliable and confirmed the results verifiably. Let C^+ and C^- be the clean and defect charts of one task, both certified, and let D^+ and D^- be the clean and defect arms rendered from them. A parser P_i reads one arm and returns a reconstruction $P_i(D)$. Rendering a chart into a PRD can result in two forms of corruption:

1. *An arm does not denote its chart.* A $P_i(D)$ can differ from the certified chart C that D was rendered from. For example, the parser could fix the defect while writing the PRD. Specifically, $P_i(D^-)$ could be equivalent to C^+. Thus, the PRD (D^-) would be labeled as defective, while actually having patched the defects. Additionally, the conversion from $C \to D$ could introduce additional defects.
2. *The two arms differ somewhere other than the defect.* Both arms may pass the gate above and still differ in sentences unrelated to the defect.

Evidence of statechart-to-PRD reliability We confirmed conversion reliability using a smaller set of 12 tasks, two per rung for $k \in \{1, \ldots, 6\}$. The writer rendered each defect statechart C^- to a defect arm D^- using the same methodology used above.

A separate writer then produced a clean arm D^+ from D^-. This writer is provided C^- and C^+, which it deterministically compares line by line. The writer then locates the differing sentences, and rewrites them to solve the defect. We then calculate the drift ratio between the original statechart and the prose. A drift ratio of 1 is the best.

Let $n(A, B)$ be the number of sentences that differ between two documents A and B. For task t the drift ratio is

$$\rho_t = \frac{n(D_t^-,\, D_t^+)}{n\big(C_t^-,\, C_t^+\big)}, \tag{1}$$

All 72 reconstructions (36 per arm) matched their source chart both structurally and by bisimulation, no planted defect was repaired, and every clean arm was free of defects. Beyond the planted defect, the two arms differed in 52 sentences against a template floor of 48 (pooled $\rho = 1.08$); $\rho_t = 1$ on 9 of 12 tasks; on the other three the excess is at most two sentences, a deleted rationale clause on the two marked tasks.

These results establish that the prose still denotes the original statechart. Particularly, that the planted ill-specification still exists, and otherwise the construction is the same. Figure 6 shows the per-task results.

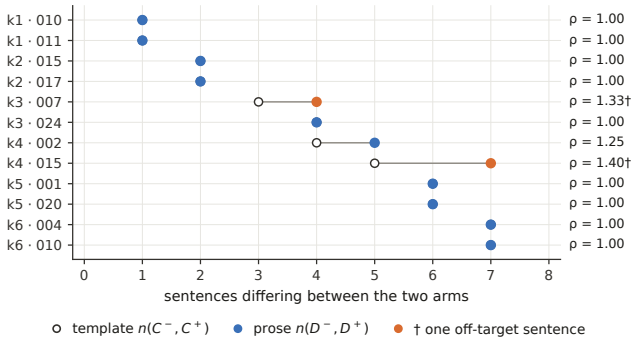


Figure 6: Twin drift on the 12-task statechart-to-PRD reliability probe (two tasks per rung, $k = 1, \ldots, 6$): the number of sentences that differ between the two prose arms, $n(D^-, D^+)$, against the template floor $n(C^-, C^+)$, with the per-task drift ratio ρ_t of Eq. 1. Pooled $\rho = 52/48 = 1.08$; $\rho_t = 1$ on 9 of 12 tasks, and 2 tasks (†) carry one off-target sentence each, a deleted rationale clause.

D CAMPAIGN FIGURES

Every task is the deterministic controller rendering of a 50-state, 6-event, 4-counter statechart, followed by the clarification protocol; the agent may call `clarify` and must then implement `handle(state, event, context)`. The outcome per rollout is whether the agent made a specific defect claim, whether or not it matches a planted defect (*predicted positive*); oracle-matched recall instead requires at least one claim to match a planted defect in the oracle facts (Section 4.2). The LLM judge proposes claims and code checks them against the formal oracle facts. Balanced accuracy (BA) is the mean of defect-arm recall and clean-arm specificity.

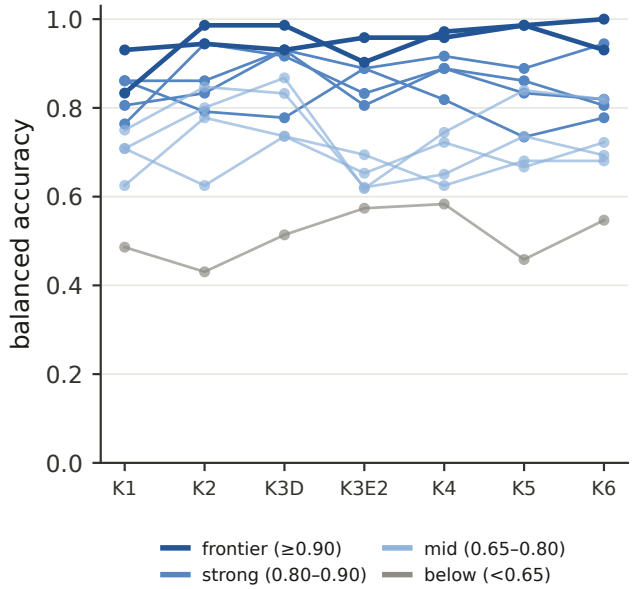


Figure 7: Per-rung balanced accuracy, all 11 models. Line colour is the model's overall tier (frontier ≥ 0.90, strong 0.80–0.90, mid 0.65–0.80, below < 0.65). Each point is 36 defective and 36 clean tasks. The rung effect is small compared with the between-model spread.

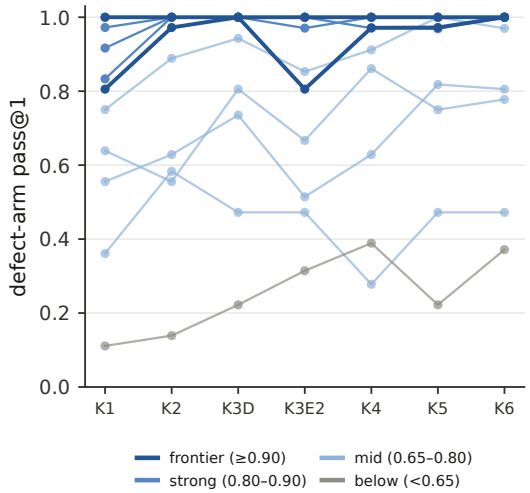


Figure 8: Defect-arm recall per rung: the fraction of defective tasks on which the agent made a specific defect claim (the recall half of balanced accuracy). Same population as the ladder above.

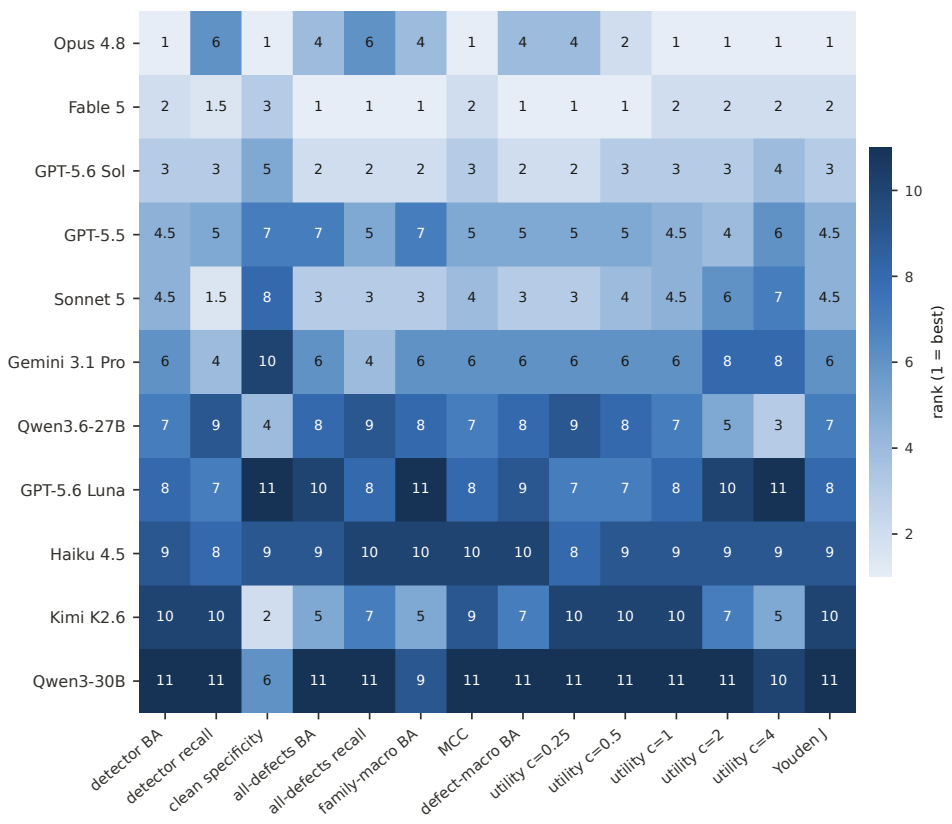


Figure 9: Model rank (1 = best) under each registered metric, on the common block of 207 base tasks completed by all 11 models. Balanced accuracy, all-defects success, and cost-weighted utility disagree on the order of the middle of the roster.

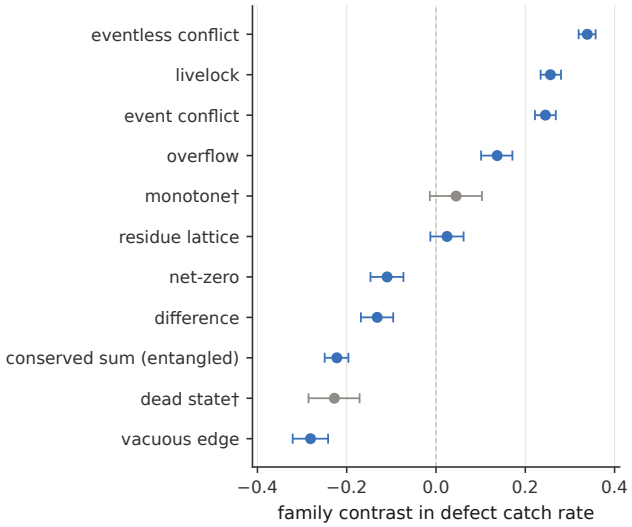


Figure 10: Difficulty by defect family. Each point is the family's contrast in defect catch rate relative to the models' own baselines, averaged with equal weight over the 11 models, with a 95% task-clustered bootstrap interval over 660 planted defects. Families marked † have fewer than 20 base tasks and are descriptive only. Family and task construction are linked, so the pattern is descriptive.

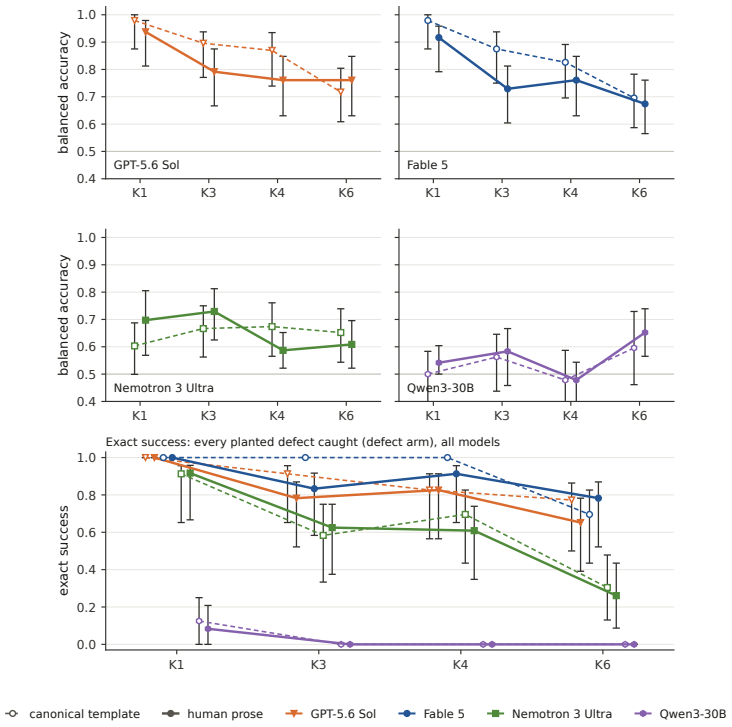


Figure 11: Paired pilot: balanced accuracy per rung on the canonical template (dashed, open markers) and on verified human prose (solid, filled) for the same base charts, one panel per model.