GENERATION AND JUDGMENT DIVERGE IN CODE MODELS: A MATCHED BENCHMARK WITH DERIVED GROUND TRUTH

Anonymous authors
Paper under double-blind review

ABSTRACT

Code-model benchmarks usually evaluate outputs whose correctness can be checked immediately, but many software decisions concern consequences that emerge only under future load. We ask whether models that can implement a change can also predict its system-level consequence. We introduce a matched benchmark built from synthetic service specifications with known resource decompositions. Each instance yields implementation tasks, graded by execution, and judgment tasks, graded against an operational capacity law validated on running services; four matched pairs probe the same underlying knowledge. Across six Qwen2.5-Coder scales from 0.5B to 32B parameters, chance-corrected implementation scores range from $+0.275$ to $+0.616$, whereas judgment scores range from -0.044 to $+0.153$, leaving a positive gap at every scale. Quantitative judgment remains at or below chance in all 24 scale-by-type cells, while categorical judgment improves with scale. A stripped arithmetic control shows that 1.5B and 3B models can solve the final argmax once the service representation is removed, localising much of the deficit to constructing the quantities required by the capacity law. Evaluation details also matter: raw accuracy can reverse model-family ordering, and a marker-strict answer parser can introduce a 45-point artifact. We derive the exact best-constant baseline for tolerance-scored numeric outputs, separate answer extraction from grading, and reproduce the main separation in a second model family.

1 INTRODUCTION

Executable evaluation has become the dominant paradigm for code models: a task is attractive when a program can determine whether the model succeeded at evaluation time (Chen et al., 2021; Jimenez et al., 2024). This has enabled reliable measurement of code generation and repository-level repair, but it covers only consequences that are immediately observable. Many engineering decisions instead concern future operating regimes: whether a design remains viable at higher load, when an unindexed table becomes a bottleneck, or when an admission limit begins to constrain throughput. These questions are operationally important precisely because their consequences may appear only after deployment.

Evaluating such judgments is difficult because architectural benchmarks often rely on expert labels. Existing multiple-choice suites, architecture task collections, smell-repair benchmarks, diagram-generation tasks, and repository-comprehension evaluations compare model outputs with curated human judgments (Santilli et al., 2026; Adnan et al., 2026; Dinu et al., 2026; Li et al., 2026; Yang et al., 2026). Expert annotation is useful, but it makes disagreement about the label difficult to separate from disagreement about the model.

We instead construct service specifications for which the relevant resource decomposition is known by design. Questions about future capacity are labelled by an operational law that we validate on running systems, while implementation tasks are graded by executing the generated artifact. A single service instance can therefore support both a *do* task and a *judge* task with objective, independently

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.

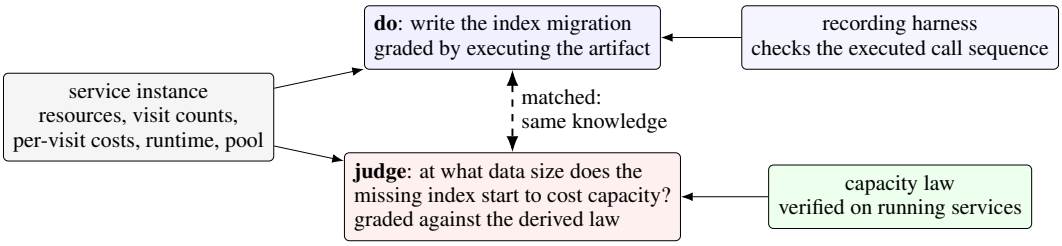


Figure 1: Benchmark design. A single service instance yields implementation (*do*) and consequence-prediction (*judge*) tasks. Do outputs are checked by execution; judge outputs are compared with labels derived from a capacity law validated on running services.

checkable ground truth. This design lets us compare production and consequence prediction on the same systems rather than across unrelated benchmarks.

Contributions.

- **Matched tasks with derived ground truth.** We introduce 11 task types over generated service specifications with known resource structure (Section 3). Do tasks are graded by execution or structured parsing; judge tasks are labelled by a capacity law validated against measurements on running services. Four task pairs instantiate the same underlying knowledge in both forms (Section 4).
- **A persistent production–judgment gap across scale.** Across 6 Qwen2.5-Coder scales spanning 64×, chance-corrected do scores are positive at every scale while judge scores remain near chance. The gap persists on instance-matched pairs and after removing the strongest do task type (Section 6).
- **Diagnostics that localise the gap.** Quantitative judge tasks remain at or below chance across all 24 scale-by-type cells, whereas categorical judgments improve. A stripped arithmetic control shows that 1.5B and 3B models can solve the core argmax when the service representation is removed, indicating that the dominant failure at those scales lies in mapping the service description to the capacity calculation rather than in the final arithmetic (Section 7).
- **Evaluation controls for heterogeneous outputs.** We derive an exact best-constant baseline for tolerance-scored numeric tasks, balance categorical labels, report answer-format compliance separately from correctness, and use a fallback answer extractor. These controls change model-family comparisons and transfer to a second model family (Sections 5 and 7.1).

2 Related work

Verifiable code evaluation. Function-level synthesis (Chen et al., 2021), repository-level issue resolution (Jimenez et al., 2024), and performance optimisation (He et al., 2025; Ma et al., 2025) use evaluators that can execute at scoring time. Our do tasks follow this paradigm. The judge tasks retain objective scoring but concern consequences that are not directly observable at the time the code is produced.

Architecture and software-quality evaluation. Recent work evaluates architectural knowledge, architecture-oriented tasks, smell repair, requirement-to-architecture generation, repository comprehension, and documented technical debt (Santilli et al., 2026; Adnan et al., 2026; Dinu et al., 2026; Li et al., 2026; Yang et al., 2026; Pagels et al., 2026; Zhu et al., 2026). These benchmarks largely depend on curated labels. Related software-engineering work studies technical debt, smells, and their persistence using expert-defined constructs (Cunningham, 1992; Ernst et al., 2015; Tufano et al., 2017; Palomba et al., 2015; Peters & Zaidman, 2012; Habchi et al., 2018). Our setting differs in that the judged property is a derived operational consequence with a computable target.

Generation and verification. Models can generate artifacts that they evaluate less reliably (West et al., 2024). CRUXEval pairs function generation with execution prediction on matched examples (Gu et al., 2024), while self-critique and self-correction work reports weaker verification than generation in several settings (Saunders et al., 2022; Huang et al., 2024; Stechly et al., 2024). Related studies show that model verbalizations need not faithfully reflect the computations that produced an answer (Turpin et al., 2023; Lanham et al., 2023), and LLM-as-judge methods rely directly on evaluation capability (Zheng et al., 2023). We study a different target: predicting a deferred system consequence of an artifact under load, with the target derived from an explicit operational model.

Capability decomposition and scaling. Generation–verification gaps have been studied as a constraint on self-improvement (Song et al., 2024), and training specifically for verification can improve checking performance (Chen et al., 2026). More broadly, aggregate loss often follows smooth scaling laws (Kaplan et al., 2020; Hoffmann et al., 2022), while individual task metrics can exhibit discontinuities whose interpretation depends on the metric (Wei et al., 2022a; Schaeffer et al., 2023; Ruan et al., 2024). We therefore hold the prompting protocol fixed, report format compliance separately, and use chance-corrected scores rather than raw accuracy. The normalisation is closely related to metrics used in BIG-bench and agreement statistics (Srivastava et al., 2023; Cohen, 1960); we extend the constant-predictor baseline to tolerance-scored numeric tasks. Chain-of-thought prompting is also scale-dependent (Wei et al., 2022b), which motivates keeping the decoding protocol fixed across the scale sweep.

Deferred consequences and capacity models. Prior work formalises costs associated with difficult-to-reverse actions and side effects (Zhai et al., 2026; Akewar & Ranjan, 2026; Krakovna et al., 2019; Turner et al., 2020), while a separate literature studies errors caused by underspecified requests (Li et al., 2023; 2025; Edwards & Schuster, 2026; Yang et al., 2025). Our prompts instead provide every quantity required by the capacity calculation. The judge labels use operational analysis (Denning & Buzen, 1978; Lazowska et al., 1984; Little, 1961), with additive demand for stages that share a serialising interpreter. This connects to classical scalability models and measurement-based capacity prediction (Amdahl, 1967; Gunther, 2007; Venkataraman et al., 2016; Gan et al., 2019; Zhang et al., 2021; Park et al., 2021); the analytic formulation is useful here because it yields exact labels for generated instances.

3 BENCHMARK CONSTRUCTION

Service instances. Each instance specifies a set of resources, their type (indexed lookup, unindexed aggregate, CPU-bound pass, cache read, or outbound call), visit counts, per-visit costs, a runtime descriptor indicating whether stages share one interpreter or execute in separate processes, and an admission-pool size. All costs needed for the calculation are stated in the prompt. Judge tasks therefore test how the model composes the supplied quantities rather than whether it recalls typical latency constants.

Judge tasks. We use seven judge types: the first stage to saturate; maximum sustainable throughput; whether a proposed change increases capacity; the data size at which a missing index becomes capacity-relevant; the single change that most improves capacity; throughput after changing the concurrency architecture; and whether the admission pool binds at a stated load. Labels are computed from the instance specification and the capacity law. The capacity-change task includes three variants—deferring work on the same machine, offloading it to separate machines, and reducing demand by batching—so the correct answer depends on both the proposed change and the runtime model.

Do tasks. We use four implementation types: implement the specified handler; write an index migration; move one stage to a background queue; and write the configuration that bounds in-flight requests. Handler and enqueue outputs are executed against a recording harness that checks the realised call sequence. Index and pool outputs are parsed for the required structured content; for pool configuration, both the value and the option that actually bounds concurrency must be correct.

All generations use greedy decoding and an identical prompting protocol. We run the same harness through a batched inference server (Kwon et al., 2023) and a local Apple-silicon backend.

scale	marker present (%)	judge macro, fallback	marker-strict	inflation
0.5B	4.3	-4.4	-49.6	45.2
1.5B	100.0	$+0.2$	$+0.2$	0.0
3B	61.2	$+2.2$	-36.5	38.7
7B	83.4	$+2.2$	$+1.4$	0.8
14B	98.9	$+10.8$	$+10.8$	0.0
32B	95.4	$+15.3$	$+15.3$	0.0

Table 1: Effect of answer extraction on aggregate judge score. A marker-strict parser can substantially change the score at low-compliance scales, whereas the fallback parser separates formatting compliance from task correctness.

Models. Our primary scale sweep uses 6 Qwen2.5-Coder instruction-tuned models (0.5B, 1.5B, 3B, 7B, 14B, 32B) (Hui et al., 2024). Restricting the sweep to one released family reduces variation in training recipe relative to cross-family comparisons. Every model is evaluated on the same 880-item subset spanning 11 task types.

4 MATCHED PAIRS

Four pairs place the same underlying knowledge into do and judge forms: index migration versus the index threshold; pool configuration versus whether the pool binds; background enqueue versus whether offloading increases capacity; and handler implementation versus the first stage to saturate. Both members of a pair are generated from the same service instance.

To preserve pairing, we sample service instances rather than independently subsampling task types. The resulting set contains 976 items, 488 matched pairs, and 170 service instances. We rebalance the subset after sampling so that the baseline guard in Section 5 holds. All paired statistics are computed only on instances containing both forms, and bootstrap resampling is performed at the service-instance level so the two members remain coupled. Three of four paired gaps exclude zero. The largest is the index pair: the model produces the migration correctly on 87.3% of matched instances while its threshold prediction is below the constant-predictor baseline, yielding a 95.6-point gap with interval $[+89.0, +104.1]$. Across the four pairs, larger do scores coincide with larger do–judge gaps. Scores differ from Table 4 because that table uses 80 items per type, whereas the paired analysis uses only instances with both task forms.

5 MEASUREMENT METHODOLOGY

Separate answer extraction from task performance. Prompts request a final answer in a fixed format, but format compliance varies substantially across model scales and task types. A parser that accepts only the requested marker can therefore confound formatting with correctness. We first parse the marker, then fall back to the last non-empty line and the last numeric token when appropriate, and report marker compliance separately. Table 1 compares this parser with a marker-strict alternative. The strict rule leaves the aggregate judge score unchanged at 3 of 6 scales but changes the apparent do–judge gap by as much as 45.2 points at 0.5B, with the largest changes occurring at low-compliance scales.

Exact constant-predictor baseline. For a multiple-choice task, the best constant predictor attains the majority-class frequency. For a positive numeric target scored correct within multiplicative tolerance τ, the optimal constant corresponds to the densest interval of width $2\log(1+\tau)$ in log-target space (Appendix A). Thus

$$\text{chance} \;=\; \frac{1}{m}\max_{a}\Big|\{\, b:\ \log y_b - \log y_a \le 2\log(1+\tau)\,\}\Big|, \tag{1}$$

which can be computed in $O(m)$ after sorting. We report $(\text{acc}-\text{chance})/(1-\text{chance})$, equal to zero for the best constant predictor and one for perfect accuracy; values below zero indicate performance worse than that baseline.

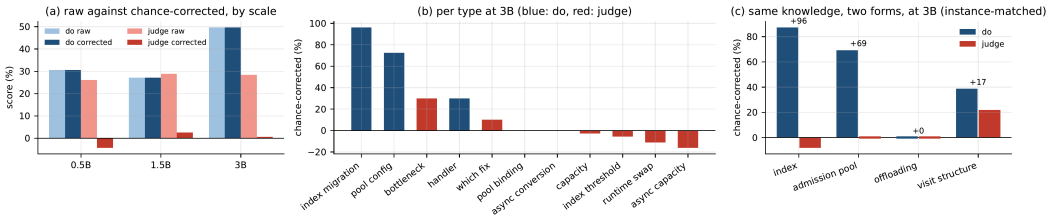


Figure 2: Performance decomposition. (a) Raw accuracy and chance-corrected score across model scales. (b) Per-task chance-corrected scores at the largest scale. (c) Do and judge scores for the four matched pairs on the instance-matched subset. The figure and tables are generated from the same scored outputs.

Item-distribution controls. We use the baseline calculation to detect task formulations with weak discrimination. For example, asking for a throughput *ratio* produces a target distribution narrow enough that a constant predictor performs well, so the released task asks for the resulting throughput instead. Multiple-choice item sets are balanced across options at construction time, and we reject released task types whose constant baseline exceeds the nominal option-frequency floor by more than five percentage points. A later fixed-size evaluation subsample can shift the empirical baseline; all reported scores therefore use the baseline recomputed on the exact evaluated subset. The instance-matched subset is rebalanced after sampling. Table 2 reports both released-set and evaluated-subset baselines.

item type	scoring	released set	evaluated subset	floor
handler / migration / enqueue / pool	executed or parsed	0.000	0.000	0.000
which stage saturates first	5-way choice	0.200	0.250	0.200
which single change helps most	4-way choice	0.250	0.263	0.250
does the change raise capacity	binary	0.500	0.537	0.500
does the pool bind	binary	0.500	**0.575**	0.500
maximum throughput	numeric, τ=0.08	0.063	0.100	–
index threshold	numeric, τ=0.12	0.059	0.100	–
throughput after architecture change	numeric, τ=0.08	0.085	0.113	–

Table 2: Best-constant baselines on the released item set and on the exact evaluated subset. Balancing is performed before the later per-type evaluation subsample, so empirical option frequencies can shift. All reported chance-corrected scores use the evaluated-subset baseline; the instance-matched subset is rebalanced after sampling.

Aggregation and uncertainty. We macro-average over task types so that no single high-frequency or easy type dominates the family score, and report micro-averages as a secondary summary. All models receive the same item subset. Bootstrap intervals resample items within type and recompute the baseline on each resample; paired analyses resample complete service instances.

6 Results

Aggregate separation. Raw accuracy reverses the do–judge ordering at one scale because several judge tasks are binary or tolerance-scored and can reward a constant response; chance correction restores the same ordering at every scale (Appendix D). Across the six-model sweep, do scores are $+0.275$, $+0.250$, $+0.441$, $+0.662$, $+0.516$, $+0.616$ and judge scores are -0.044, $+0.002$, $+0.022$, $+0.022$, $+0.108$, $+0.153$, leaving a gap between 0.248 and 0.641 over a 64$\times$ parameter range. At 32B, the judge interval is above zero ($+0.153$, $[+0.047, +0.183]$), but remains well below the do score $+0.616$, for a gap of $+0.463$. Removing index migration, the strongest do task, leaves a difference of $+0.334$ with interval $[+0.285, +0.454]$.

The gap is concentrated in quantitative judgment. Table 3 separates judge tasks by answer form. Categorical tasks improve from $+0.050$, $+0.152$, $+0.179$, $+0.135$, $+0.337$, $+0.432$, whereas

scale	do	judge	judge 95% CI	gap	quantitative	categorical
0.5B	$+0.275$	-0.044	$[-0.224, +0.005]$	$\mathbf{+0.319}$	-0.114	$+0.050$
1.5B	$+0.250$	$+0.002$	$[-0.096, +0.014]$	$\mathbf{+0.248}$	-0.110	$+0.152$
3B	$+0.441$	$+0.022$	$[-0.113, +0.049]$	$\mathbf{+0.419}$	-0.096	$+0.179$
7B	$+0.662$	$+0.022$	$[-0.117, +0.059]$	$\mathbf{+0.641}$	-0.063	$+0.135$
14B	$+0.516$	$+0.108$	$[-0.017, +0.146]$	$\mathbf{+0.407}$	-0.063	$+0.337$
32B	$+0.616$	$+0.153$	$[+0.047, +0.183]$	$\mathbf{+0.463}$	-0.056	$+0.432$

Table 3: Chance-corrected macro scores for six scales of one code-model family on the same 880 items. Categorical judge tasks improve with scale, whereas quantitative judge tasks remain at or below the best-constant baseline at every scale.

item type	form	n	acc (%)	chance (%)	corrected (%)	95% CI
handler	do	80	73.8	0.0	$+73.8$	$[+63.7, +82.5]$
index migration	do	80	100.0	0.0	$+100.0$	$[+100.0, +100.0]$
async conversion	do	80	0.0	0.0	$+0.0$	$[+0.0, +0.0]$
pool config	do	80	72.5	0.0	$+72.5$	$[+62.5, +82.5]$
bottleneck	judge	80	40.0	25.0	$+20.0$	$[+0.0, +32.8]$
capacity	judge	80	6.2	10.0	-4.2	$[-16.7, +0.0]$
async capacity	judge	80	53.8	53.8	$+0.0$	$[-38.5, +21.2]$
index threshold	judge	80	1.2	10.0	-9.7	$[-21.2, -8.5]$
which fix	judge	80	87.5	26.2	$+83.1$	$[+70.4, +91.2]$
runtime swap	judge	80	3.8	11.2	-8.5	$[-21.9, -4.3]$
pool binding	judge	80	68.8	57.5	$+26.5$	$[-7.7, +50.0]$

Table 4: Per-task results at 32B. Bootstrap intervals resample items and recompute the chance baseline on each draw. All outputs at this scale are parseable, so the reported differences are not caused by missing extracted answers.

the four quantitative task types follow -0.114, -0.110, -0.096, -0.063, -0.063, -0.056 and remain at or below the constant-predictor baseline in all 24 scale-by-type cells. At the largest scale, selecting the best intervention reaches $+83.1$ and identifying the first saturated stage reaches $+20.0$, while maximum throughput, index threshold, throughput after a concurrency change, and the offloading-capacity decision score -4.2, -9.7, -8.5, and 0.0, respectively. The paired analysis shows the same asymmetry on identical service instances. The largest gap occurs for the index pair, where the implementation is a short migration statement but the judge task requires composing a growth law with competing resource demands.

7 Diagnostics of judge-task behavior

Chance correction establishes that aggregate judge performance is near the constant-predictor baseline, but different mechanisms can produce the same score. We therefore inspect answer distributions, parse rates, and generation lengths. Table 6 summarises these diagnostics at two scales.

At 3B, each binary judge task collapses to a single response across all items, with different constants for the two tasks. Their raw accuracies therefore track the empirical class frequencies rather than instance-specific predictions. The two multiple-choice tasks show strong option preference, and the threshold task uses one round-number answer on 65% of instances. On the two hardest numeric tasks, only 47 and 68 of 80 generations yield a parseable value. The median 3B judge response is six tokens, compared with 98 tokens for do tasks generated from the same service instances.

These signatures change substantially at 32B. No judge type is constant; modal-answer rates fall to 0.50–1.29 times the gold majority; each task type parses on at least 73 of 80 items; and the median judge response grows to 474 tokens. The remaining $+0.463$ do–judge gap at 32B therefore cannot be explained by constant responses, dominant option position, answer absence, or uniformly short generations.

matched pair	instances	do (%)	judge (%)	gap (%)	gap 95% CI
index	166	$+87.3$	-8.2	$\mathbf{+95.6}$	$[+89.0, +104.1]$
admission pool	114	$+69.3$	$+0.0$	$\mathbf{+69.3}$	$[+61.7, +112.7]$
offloading	128	$+0.0$	$+0.0$	$\mathbf{+0.0}$	$[+0.0, +41.5]$
visit structure	80	$+38.8$	$+21.9$	$\mathbf{+16.9}$	$[+6.9, +41.0]$

Table 5: Instance-matched do and judge scores. Both task forms are evaluated on the same service instances. The chance baseline is recomputed within the paired subset, and confidence intervals use an instance-level bootstrap that resamples both task forms together and recomputes their baselines. Three of four gaps exclude zero.

	3B			32B		
judge item type	modal	rate	$\times$ gold	modal	rate	$\times$ gold
does offloading raise capacity	`no`	100.0%	1.86	`yes`	60.0%	1.12
does the pool bind	`yes`	100.0%	1.74	`yes`	58.8%	1.02
which change helps most	`1`	75.0%	2.86	`3`	33.8%	1.29
index threshold	`100000`	65.0%	6.50	`400000`	11.2%	1.12
which stage saturates first	`1`	50.0%	2.00	`2`	26.2%	1.05
throughput after the swap	*none*	41.2%	3.67	`64`	7.5%	0.67
maximum throughput	*none*	15.0%	1.50	`16`	5.0%	0.50

Table 6: Judge-output concentration at 3B and 32B. The modal response is constant for two task types at 3B and is substantially more concentrated than the gold-label distribution for several others. At 32B, no task type is constant and modal rates are much closer to the gold-label majority.

Stripped arithmetic control. Several judge tasks ultimately require maximising a product over three to five candidates. To separate this arithmetic from service interpretation, we generate the same argmax from the same underlying quantities but present only a numbered list of count–value pairs. The control contains 370 items balanced across five answer labels, giving a constant-predictor baseline of 0.200.

	argmax with the semantics		the same argmax as	
scale	stripped out	95% CI	a capacity question	difference
0.5B	$+4.1$	$[-6.0, +6.6]$	$+11.7$	-7.6
1.5B	$+57.4$	$[+49.1, +62.4]$	$+11.7$	45.8
3B	$+61.5$	$[+53.8, +66.3]$	$+30.0$	31.5

Table 7: Chance-corrected score on the arithmetic control against the bottleneck judge item. Both ask for the argument maximising a product of two numbers over three to five candidates; the control presents the numbers as a list and the judge item presents them as a service. The control carries 370 items with a balanced answer distribution, so its chance baseline is 0.200.

At 0.5B the stripped computation remains near chance ($+4.1$, interval $[-6.0, +6.6]$), so computation itself is a limiting factor. At 1.5B and 3B, however, the stripped control reaches $+57.4$ and $+61.5$, compared with $+11.7$ and $+30.0$ on the corresponding capacity judgment. The difference is therefore not explained by the final product comparison alone. At these scales, a substantial part of the deficit arises earlier: mapping a service description into the demand representation required by the capacity calculation.

Answer-distribution diagnostics. Modal-answer frequency is inexpensive to compute and distinguishes partially informative predictions from constant responses that can obtain nontrivial raw accuracy. We report it together with chance-corrected performance because the two statistics answer different questions: chance correction removes baseline inflation, whereas the modal rate reveals how concentrated the model's output distribution is.

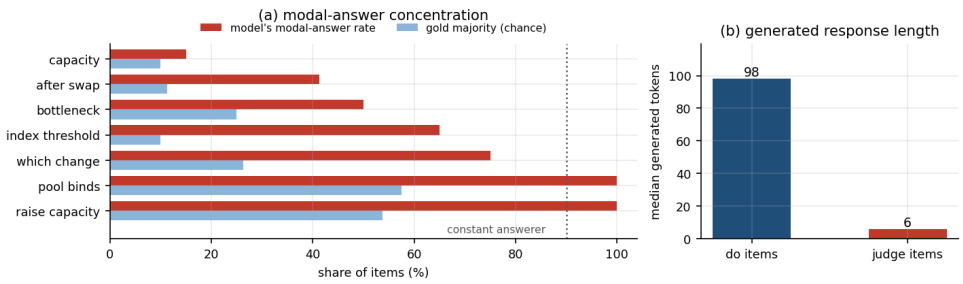


Figure 3: Judge-output diagnostics at 3B. (a) Modal-answer frequency relative to the gold-label majority for each judge task. (b) Median generated length for do and judge tasks.

		released parser		perfect answer locator	
family and scale	marker (%)	do	judge	judge	gap
Qwen2.5-Coder 1.5B	100	+0.250	+0.002	+0.002	**+0.248**
DeepSeek-Coder 1.3B	0	+0.303	−0.311	+0.162	**+0.141**

Table 8: Cross-family evaluation on the same 880 items. The released parser uses the benchmark's normal extraction rule. The oracle locator scores every output line with the same task grader and keeps the best; because it uses the gold answer, it is an upper bound on recoverable judge performance rather than an evaluation metric.

7.1 Cross-family evaluation and answer extraction

The scale sweep uses one model family to reduce training-recipe variation, so we also evaluate a second code-model family on the same 880 items with identical graders and per-type baselines.

Its do score is $+0.303$, compared with $+0.250$ for the nearest Qwen scale, and both are based on execution-based grading. Judge evaluation is more sensitive to answer extraction: the second family emits the requested final-answer marker on none of its 560 judge outputs, so the fallback parser reads the final non-empty line of a long explanation. Under the released parser its judge macro is -0.311 and the do–judge gap is $+0.614$.

To quantify the effect of extraction independently of grading, we score every line using the same task grader and retain the best line, which gives an oracle upper bound for a perfect answer locator. This changes the Qwen score by 0.0 points and the second-family score by 25.2 points. Under the oracle locator, the second family's raw judge accuracy is 35.5%, compared with 28.2% for the neighbouring Qwen scale, and its do–judge gap falls to $+0.141$. Thus the production–judgment separation transfers across families, but its magnitude depends strongly on answer extraction. Reporting extraction compliance or using a family-robust locator is therefore necessary for meaningful cross-family comparisons.

8 Discussion

The experiments separate two capabilities that are often aggregated in code-model evaluation: producing a locally correct software change and predicting a deferred operational consequence of that change. On matched service instances, implementation performance is consistently stronger than consequence prediction, and scaling primarily improves categorical judgment rather than quantitative capacity estimation. The stripped arithmetic control further indicates that, once basic arithmetic is available, much of the remaining difficulty lies in constructing the right operational representation from the service description.

The results also illustrate two general measurement issues. First, heterogeneous task families require baselines matched to their answer spaces: raw accuracy can favour a constant predictor and can change a family-level conclusion. Second, answer extraction should be treated as a separate component of evaluation. A rigid marker requirement can create large apparent capability differ-

ences across model families even when the underlying grader is unchanged. The exact constant baseline, per-type balancing, answer-distribution diagnostics, and extraction-compliance reporting used here are intended to make those effects explicit.

The benchmark deliberately uses generated service specifications and an analytic capacity model so that ground truth remains objective and scalable. This improves control but narrows the domain: real systems contain interactions, workload shifts, queuing effects, and implementation details not represented by the generator. The benchmark therefore measures whether models can apply a validated operational abstraction to fully specified instances; it does not establish performance on unrestricted architecture review. Extending the same matched design to richer measured systems is a natural next step.

Reproducibility statement

The submission includes the item generator, grading harness and execution sandbox, exact evaluated item subsets, raw generations and per-item scores, chance-baseline and balancing code, and scripts that regenerate every reported table and figure from stored outputs. An independent checker recomputes judge labels from the quantities printed in each prompt rather than from generator internals; on the released checks it recovers all bottleneck and pool-binding labels and reproduces index thresholds to median relative deviation 10^{-4}. The appendix documents the numeric baseline derivation and grading rules.

Ethics statement

The benchmark uses generated service specifications and contains no personal data or third-party code. Model-produced code is executed only in a subprocess with a timeout and network access disabled.

AI use statement

GPT was used to polish manuscript wording, assist with code drafting/refactoring, and aid literature search; all outputs were reviewed and verified by the authors.

References

Bassam Adnan, Aviral Gupta, Sreemaee Akshathala, and Karthik Vaidhyanathan. Archbench: Benchmarking generative-ai for software architecture tasks. *arXiv:2603.17833*, 2026.

Mayur Akewar and Ravi Ranjan. Safecommit: Certifying when memory-grounded agents may safely act. *arXiv:2608.04289*, 2026.

Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Spring Joint Computer Conference*, 1967.

Mark Chen, Jerry Tworek, Heewoo Jun, et al. Evaluating large language models trained on code. *arXiv:2107.03374*, 2021.

Yuxin Chen, Yu Wang, Yi Zhang, Ziang Ye, Zhengzhou Cai, Yaorui Shi, Qi Gu, Hui Su, Xunliang Cai, Xiang Wang, An Zhang, and Tat-Seng Chua. Learning to self-verify makes language models better reasoners. *arXiv:2602.07594*, 2026.

Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

Ward Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1992.

Peter J. Denning and Jeffrey P. Buzen. The operational analysis of queueing network models. *ACM Computing Surveys*, 10(3):225–261, 1978.

Ion George Dinu, Marian Cristian Mihăescu, and Traian Rebedea. Smellbench: Evaluating llm agents on architectural code smell repair. *arXiv:2605.07001*, 2026.

Nicholas Edwards and Sebastian Schuster. Ask or assume? uncertainty-aware clarification-seeking in coding agents. *arXiv:2603.26233*, 2026.

Neil A. Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L. Nord, and Ian Gorton. Measure it? manage it? ignore it? software practitioners and technical debt. In *ACM ESEC/FSE*, 2015.

Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *ASPLOS*, 2019.

Alex Gu, Baptiste Roziere, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. Cruxeval: A benchmark for code reasoning, understanding and execution. In *ICML*, 2024.

Neil J. Gunther. *Guerrilla Capacity Planning*. Springer, 2007.

Sarra Habchi, Geoffrey Hecht, Romain Rouvoy, and Naouel Moha. The rise of android code smells: Who is to blame? In *IEEE/ACM Working Conference on Mining Software Repositories*, 2018.

Xinyi He, Qian Liu, Mingzhe Du, et al. Swe-perf: Can language models optimize code performance on real-world repositories? *arXiv:2507.12415*, 2025.

John Hewitt and Percy Liang. Designing and interpreting probes with control tasks. In *EMNLP*, 2019.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, et al. Training compute-optimal large language models. In *NeurIPS*, 2022.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *ICLR*, 2024.

Binyuan Hui, Jian Yang, Zeyu Cui, et al. Qwen2.5-coder technical report. *arXiv:2409.12186*, 2024.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *ICLR*, 2024.

Jared Kaplan, Sam McCandlish, Tom Henighan, et al. Scaling laws for neural language models. *arXiv:2001.08361*, 2020.

Victoria Krakovna, Laurent Orseau, Ramana Kumar, Miljan Martic, and Shane Legg. Penalizing side effects using stepwise relative reachability. In *AI Safety Workshop, IJCAI*, 2019.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *ACM SOSP*, 2023.

Tamera Lanham, Anna Chen, Ansh Radhakrishnan, Benoit Steiner, Carson Denison, Danny Hernandez, Dustin Li, Esin Durmus, Evan Hubinger, Jackson Kernion, et al. Measuring faithfulness in chain-of-thought reasoning. *arXiv:2307.13702*, 2023.

Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, 1984.

Belinda Z. Li, Alex Tamkin, Noah Goodman, and Jacob Andreas. Eliciting human preferences with language models. In *arXiv:2310.11589*, 2023.

Belinda Z. Li, Been Kim, and Zi Wang. Questbench: Can llms ask the right question to acquire information in reasoning tasks? *arXiv:2503.22674*, 2025.

Minxiao Li, Li Zhang, Shuying Yan, Fang Liu, Liehao Li, Yang Liu, et al. Benchmarking requirement-to-architecture generation with hybrid evaluation. *arXiv:2604.06683*, 2026.

John D. C. Little. A proof for the queuing formula: $l = \lambda w$. *Operations Research*, 9(3):383–387, 1961.

Jeffrey Jian Ma, Milad Hashemi, Amir Yazdanbakhsh, Kevin Swersky, Ofir Press, Enhui Li, et al. Swe-fficiency: Can language models optimize real-world repositories on real workloads? *arXiv:2511.06090*, 2025.

Christin Pagels, Simon Hacks, and Rob Henk Bemthuis. Large language models for analyzing enterprise architecture debt in unstructured documentation. *arXiv:2604.00046*, 2026.

Fabio Palomba, Dario Di Nucci, Michele Tufano, Gabriele Bavota, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. Landfill: An open dataset of code smells with public evaluation. In *IEEE/ACM Working Conference on Mining Software Repositories*, 2015.

Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. Graf: A graph neural network based proactive resource allocation framework for slo-oriented microservices. *ACM CoNEXT*, 2021.

Ralph Peters and Andy Zaidman. Evaluating the lifespan of code smells using software repository mining. In *European Conference on Software Maintenance and Reengineering*, 2012.

Yangjun Ruan, Chris J. Maddison, and Tatsunori Hashimoto. Observational scaling laws and the predictability of language model performance. In *NeurIPS*, 2024.

Tiziano Santilli, Francesco Daghero, and Mayhar Tourchi Moghaddam. Sake: Software architectural knowledge evaluation benchmark for large language models. *arXiv:2606.29520*, 2026.

William Saunders, Catherine Yeh, Jeff Wu, Steven Bills, Long Ouyang, Jonathan Ward, and Jan Leike. Self-critiquing models for assisting human evaluators. *arXiv:2206.05802*, 2022.

Rylan Schaeffer, Brando Miranda, and Sanmi Koyejo. Are emergent abilities of large language models a mirage? In *NeurIPS*, 2023.

Yuda Song, Hanlin Zhang, Carson Eisenach, Sham Kakade, Dean Foster, and Udaya Ghai. Mind the gap: Examining the self-improvement capabilities of large language models. *arXiv:2412.02674*, 2024.

Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *Transactions on Machine Learning Research*, 2023.

Kaya Stechly, Karthik Valmeekam, and Subbarao Kambhampati. On the self-verification limitations of large language models on reasoning and planning tasks. In *arXiv:2402.08115*, 2024.

Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering*, 43(11):1063–1088, 2017.

Alexander Matt Turner, Dylan Hadfield-Menell, and Prasad Tadepalli. Conservative agency via attainable utility preservation. In *AAAI/ACM Conference on AI, Ethics, and Society*, 2020.

Miles Turpin, Julian Michael, Ethan Perez, and Samuel R. Bowman. Language models don't always say what they think: Unfaithful explanations in chain-of-thought prompting. In *Advances in Neural Information Processing Systems*, 2023.

Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *USENIX NSDI*, 2016.

Jason Wei, Yi Tay, Rishi Bommasani, et al. Emergent abilities of large language models. *Transactions on Machine Learning Research*, 2022a.

Jason Wei, Xuezhi Wang, Dale Schuurmans, et al. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*, 2022b.

Peter West, Ximing Lu, Nouha Dziri, Faeze Brahman, Linjie Li, Jena D. Hwang, Liwei Jiang, Jillian Fisher, Abhilasha Ravichander, Khyathi Chandu, Benjamin Newman, Pang Wei Koh, Allyson Ettinger, and Yejin Choi. The generative ai paradox: What it can create, it may not understand. In *ICLR*, 2024.

Chenyang Yang, Yike Shi, Qianou Ma, Michael Xieyang Liu, Christian Kästner, and Tongshuang Wu. What prompts don't say: Understanding and managing underspecification in llm prompts. *arXiv:2505.13360*, 2025.

Yuexi Yang, Alyssa Wu, Ji Luo, Richeng Xuan, Zhichao Hu, Yuhong Liu, et al. Repoprobe: Benchmarking architecture-aware repository comprehension with checklists. *arXiv:2608.04783*, 2026.

Zhiyuan Zhai, Ming Li, and Xin Wang. Revisable by design: A theory of streaming llm agent execution. *arXiv:2604.23283*, 2026.

Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh, and Christina Delimitrou. Sinan: Ml-based and qos-aware resource management for cloud microservices. In *ASPLOS*, 2021.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena. In *NeurIPS*, 2023.

Yuecai Zhu, Nikolaos Tsantalis, and Peter C. Rigby. Ai-generated smells: An analysis of code and architecture in llm and agent-driven development. *arXiv:2605.02741*, 2026.

A THE BEST CONSTANT PREDICTOR FOR A TOLERANCE-SCORED NUMERIC ITEM

A numeric item with gold answer y is scored correct when the prediction $\hat{y}$ satisfies $|\log \hat{y} - \log y| \le w$ with $w = \log(1+\tau)$, which is what a multiplicative tolerance τ means. Fix a constant predictor c and let $y_1 \le \cdots \le y_m$ be the gold answers of the m items of that type. The constant scores on item b exactly when $\log c \in [\log y_b - w, \log y_b + w]$, so its total score is the number of those m intervals that contain the single point $\log c$. Maximising over c is therefore an interval stabbing problem, and its value is the largest number of gold answers falling in any window of width $2w$ on the log scale.

The maximum is attained with the window's left endpoint at a data point. If a window $[\ell, \ell + 2w]$ covers a set S of answers, sliding it right until ℓ meets $\min\{\log y_b : b \in S\}$ removes nothing from S, because every element of S is at least ℓ already, and can only admit further points on the right. Hence it suffices to consider $\ell = \log y_a$ for $a = 1, \ldots, m$, which gives

$$\text{chance} = \frac{1}{m} \max_a \big|\{\, b : \ \log y_b - \log y_a \le 2w \,\}\big|,$$

the quantity in (1). Sorting costs $O(m \log m)$ and a single two-pointer sweep then evaluates every a in $O(m)$ total, because the right endpoint of the window is nondecreasing in a.

Two properties matter for how we use it. The baseline is exact: no constant predictor scores above it, so a chance-corrected score of zero means the item type is answered no better than by a constant. And it depends only on the gold answers, not on any model, so it can be computed when the item set is built — which is what the release-time guard of §5 does.

B ITEM TEMPLATES

Each judge task contains the resource list, per-visit costs, runtime description, and pool size, followed by one question and a requested final-answer line. Each do task uses the same service instance and requests one fenced artifact. The offloading task has three variants: deferring the same work on the same machine; moving it to separate machines, whose effect depends on the runtime and resulting demand decomposition; and batching, which directly reduces demand.

scale	raw accuracy (%)		chance-corrected (%)		gap
	do	judge	do	judge	
0.5B	27.5	25.9	+27.5	−4.4	+31.9
1.5B	25.0	28.2	+25.0	+0.2	+24.8
3B	44.1	28.9	+44.1	+2.2	+41.9
7B	66.2	29.1	+66.2	+2.2	+64.1
14B	51.6	35.0	+51.6	+10.8	+40.7
32B	61.6	37.3	+61.6	+15.3	+46.3

Table 9: Raw and chance-corrected macro scores by model scale. At 1 of 6 scales, raw accuracy reverses the do–judge ordering relative to the chance-corrected metric.

C GRADING DETAILS

Handler outputs are executed against a context object that records resource calls, and the realised sequence is compared with the specification. Enqueue outputs additionally require the moved stage to appear exactly once as the correct queued operation while all remaining stages stay inline and ordered. Index outputs must contain a create-index statement for the correct table and column. Pool outputs must provide the correct value and a configuration option that actually bounds concurrency.

D RAW AGAINST CHANCE-CORRECTED, BY SCALE

E PER-TYPE RESULTS AT EVERY SCALE

The grading harness writes a per-type table, marker-compliance rates and generation-length summaries for every scored run; these are produced by one command over the released generations, which are included for all 6 scales.