Test-Time Harness for Strong-to-Weak Capability Transfer

Anonymous authors
Paper under double-blind review

Abstract

Recent work on distillation transfers the capabilities of large models to smaller ones often by updating the latter's parameters, through teacher forcing, on-policy distillation, and related training-time methods. In this paper, we ask whether such transfer can instead occur at test time. We study **strong-to-weak scaffolding**: whether a stronger *builder model* can construct inference-time harnesses that help a weaker *target model* solve tasks more reliably without any parameter updates. Using four representative Theory-of-Mind benchmarks, each builder model uses 5% of the data as a validation set to iteratively refine its harness over multiple rounds, after which the finalized harness is evaluated on the full test set. Empirically, this form of test-time capability transfer is highly effective, nearly doubling average target-model performance from 0.49 to 0.91. Our analysis shows that the gains come primarily from offloading unstable model reasoning into deterministic code, benchmark-specific routing, and strict answer-format enforcement, rather than from encouraging the target model to reason more extensively or sample more broadly. We further find that builder-model reasoning effort improves harness quality monotonically, platform effects are modest relative to the builder model's own capability, and weaker target models receive the largest gains. These results suggest that inference-time harness design is an important complement to conventional training-time distillation, enabling strong models to transfer cognitive structure to weaker models without retraining.

1 Introduction

Recent progress in model distillation has made it increasingly plausible to deploy smaller language model experts in settings that once required much larger ones (Hinton et al., 2015; Hsieh et al., 2023; Agarwal et al., 2024). Most existing approaches transfer capability by changing the weak model itself. For instance, *data distillation* trains a small student on examples, rationales, or demonstrations produced by a stronger teacher. *On-policy distillation* further exposes the student to dense feedback, preferences, or reward signals while it acts, allowing the student to internalize behaviors that would otherwise be difficult to acquire from static data alone (Agarwal et al., 2024; Ouyang et al., 2022). These approaches are effective, but they share a common premise: closing the gap between a strong model and a weak model needs additional training.

This paper studies a complementary premise. When a small model fails on a task, the failure may reflect not only insufficient internal capability, but also excessive *cognitive load* imposed by the way the task is presented. (Sweller, 1988). A system can therefore improve performance in two ways: it can make the model more capable, or it can make the task easier for the model to solve. The first route is the dominant route of distillation. The second route is increasingly realized through **inference-time harnesses**: external structures that surround a model with routing logic, prompt templates, verification checks, memory, and tool use (Wei et al., 2022; Yao et al., 2022; Schick et al., 2023; Madaan et al., 2023). Rather updating the target model's parameters, a harness engineers the conditions under which the target model reasons.

In this paper, we investigate into this **strong-to-weak scaffolding**. Specifically, a strong *builder* model is asked to construct a scaffold: any combination of task routing, prompt templates, deterministic solvers, few-shot exemplars, verification passes, or format enforcement designed to improve a fixed weaker *target* model on a hidden test set. The target model is not trained, and the builder never observes the full test set; its only opportunity to improve performance is to design an inference-time environment that transfers across

examples. Thus, a successful scaffold must capture reusable skills and task structure rather than memorize instance-specific answers. This setting isolates a practical form of strong-to-weak transfer in which capability is transferred not through model weights, but through the harness that shapes how the weak model receives, reasons, and responds.

Strong-to-weak scaffolding is becoming more important as model deployment shifts from single prompts to agentic systems (Wang et al., 2024; Ke et al., 2025). In practice, small models are rarely used in isolation. They are embedded in pipelines that parse inputs, select tools, check answers, and decompose tasks (Yue et al., 2026). Yet we still lack a systematic account of why these harnesses help, when they are stable, which design choices matter, and how much of the improvement reflects genuine reasoning support rather than benchmark-specific shortcuts (Ullman, 2023; Riemer et al., 2024). Without such an account, harness engineering remains difficult to compare, reproduce, and improve.

We investigate these questions in the domain of Theory-of-Mind (ToM) reasoning. ToM benchmarks are a useful stress test because they require models to track nested beliefs, perspective shifts, hidden information, and Bayesian goal inference (Chen et al., 2025; Wu et al., 2023; Baker et al., 2009). These demands are challenging for smaller models, but they also contain structure that an external scaffold may exploit: tasks can often be routed by subtype, decomposed into intermediate states, checked for consistency, or solved partly through symbolic procedures (Zhang et al., 2026). To fully understand the effect of scaffolding, we analyze a large corpus of strong-builder, weak-target runs and investigate into the following aspects:

- **Effect size:** how much scaffolding improves weak target model's accuracy overall;
- **Stability:** whether independently built scaffolds produce consistent gains;
- **Validation effort:** how much times the validation data is used, and whether using more helps;
- **Scaffold techniques:** what specific mechanisms builders actually implement;
- **Platform effects:** if the builder's own agentic harness changes outcomes;
- **Target dependence:** how scaffolding strategies and gains vary with the weak target model;
- **Builder reasoning effort:** whether stronger internal reasoning of builder improves scaffolds;
- **Causal mechanisms:** which scaffold features are associated with accuracy gains;
- **Cognitive-load reduction:** how much reasoning is shifted away from the target model;
- **Failure modes:** where even the best scaffolds continue to make errors.

Our empirical results show that strong-to-weak scaffolding is both large and robust. The best scaffold raises GPT-5.4-mini from a macro-average accuracy of 0.49 to 0.91, an absolute gain of 0.42, and every builder configuration yields positive net uplift. The gains are driven less by using more validation data, sampling more, or simply eliciting longer reasoning, and more by structure externalization, deterministic offloading, and strict format enforcement, etc.

Besides, the improvements are not uniform in kind. On BigToM, the best scaffolds discover that answers can be fully exploited through structured reasoning, thus turning the benchmark into compilable skills and rules. On the other benchmarks, the gains reflect genuine reduction of reasoning burden rather than a shortcut. Residual errors concentrate in the hardest regimes, especially higher-order Hi-ToM cases with recursion depth at least two and Bayesian goal-inference subtypes. We also find that builder reasoning effort improves scaffold quality monotonically across effort tiers, platform effects are modest relative to builder-model effects, and scaffolding helps the weaker GPT-5.4-mini target more than the already stronger Gemini-3.5-flash target. In summary, this study makes three contributions:

- First, we formalize *strong-to-weak scaffolding* as a distinct inference-time capability transfer setting.
- Second, we provide a systematic empirical analysis of its effect size, stability, validation efficiency, platform and target dependence, mechanisms, and cognitive-load reduction.
- Third, we identify actionable design principles: successful scaffolds often rely on deterministic offloading, benchmark-aware routing, format control, and targeted decomposition than brute-force validation search.

Looking ahead, strong-to-weak scaffolding is valuable not only as a deployment strategy but also as a way to evaluate builder-model capability. For deployment, it offers a practical route to improving weaker or cheaper models at inference time, without changing their weights. This closely aligns with the motivation behind automatic agent-harness self-evolution. For evaluation, it reframes the question from "How well can a model solve a task?" to "How well can a stronger model construct the conditions under which a weaker model

can solve it?" Given the same hidden task, validation budget, and open-ended workspace, the quality of the resulting scaffold becomes a measure of the builder's ability to externalize reasoning into procedures, tools, feedback loops, and constraints. More broadly, this setting creates a natural platform for studying harness evolution itself: what structures builders invent, where their designs fail, how they revise them, and which forms of external organization most efficiently translate strong-model insight into weak-model performance.

2 Related Work

Capability transfer and distillation. A line of existing work studies how capabilities of a larger or stronger model can be transferred to a smaller or cheaper one through training. Classical knowledge distillation compresses an ensemble or high-capacity teacher into a deployable student by training the student to match softened teacher outputs (Hinton et al., 2015). Recent language-model distillation methods extend this paradigm by transferring rationales, traces, or task-specific reasoning supervision: for example, distilling step-by-step uses teacher-generated rationales as additional supervision for smaller task models (Hsieh et al., 2023), while on-policy distillation trains on student-generated sequences with teacher feedback to reduce the distribution mismatch between training and inference (Agarwal et al., 2024). Instruction tuning and RLHF similarly alter the model policy through supervised and preference-based training signals (Ouyang et al., 2022). Related alignment work on weak-to-strong generalization asks whether weak supervision can elicit capabilities from a stronger model, but still studies capability transfer through model updating rather than through the inference environment (Burns et al., 2023). Our work differs from these previous paradigms as a complementary test-time capability-transfer paradigm: instead of changing the weak target model's parameters, it asks whether a strong builder can externalize transferable task structure into a reusable harness that improves a weak target at inference time.

Inference-time reasoning, prompting, and decomposition. A second line of work improves model reasoning without conventional fine-tuning by changing the inference procedure. Chain-of-thought prompting elicits intermediate reasoning steps from large models (Wei et al., 2022), self-consistency improves reliability by sampling multiple reasoning paths and marginalizing over their answers (Wang et al., 2022), and least-to-most prompting decomposes hard problems into easier subproblems solved sequentially (Zhou et al., 2022). Decomposed prompting generalizes this modular view by delegating subproblems to specialized prompts, models, or symbolic functions (Khot et al., 2022). Iterative refinement methods such as Self-Refine use model-generated feedback to improve an initial answer over multiple rounds (Madaan et al., 2023), while Tree-of-Thoughts and Graph-of-Thoughts treat reasoning as search over structured intermediate states rather than as a single left-to-right trace (Yao et al., 2023; Besta et al., 2024). These methods show that test-time structure can substantially improve reasoning, but they typically optimize how the same model reasons on each instance. Our work is beyond single-model prompting by studying a cross-model scaffold-building setting in which a strong builder constructs a persistent inference-time procedure that a separate weaker target can execute across hidden examples.

Tool use, programmatic reasoning, and deterministic offloading. Another closely related perspective treats reasoning failures as failures of execution, verification, or state tracking rather than failures of language understanding alone. Toolformer trains language models to decide when and how to call external APIs (Schick et al., 2023), and ReAct interleaves natural-language reasoning with environment actions or tool calls (Yao et al., 2022). Program-aided methods such as PAL and Program-of-Thoughts ask the model to translate a problem into executable code, leaving exact computation to a Python interpreter (Gao et al., 2023; Chen et al., 2022). Faithful chain-of-thought similarly separates translation from solving by using a deterministic solver to execute symbolic reasoning chains (Lyu et al., 2023). This literature motivates the view that a model can be made more reliable by moving fragile cognitive work into external tools, code, or checkable representations. Our work follows these motivations as a systematic empirical study of when such offloading emerges automatically from strong-builder scaffold design, showing that deterministic solvers, benchmark routing, and answer-format enforcement can transfer cognitive structure to a weaker target more effectively than merely encouraging longer target-model reasoning.

Harness engineering and automated scaffold construction. Recent work increasingly treats the system surrounding an LLM, such as prompts, tools, memory, context management, execution interfaces,

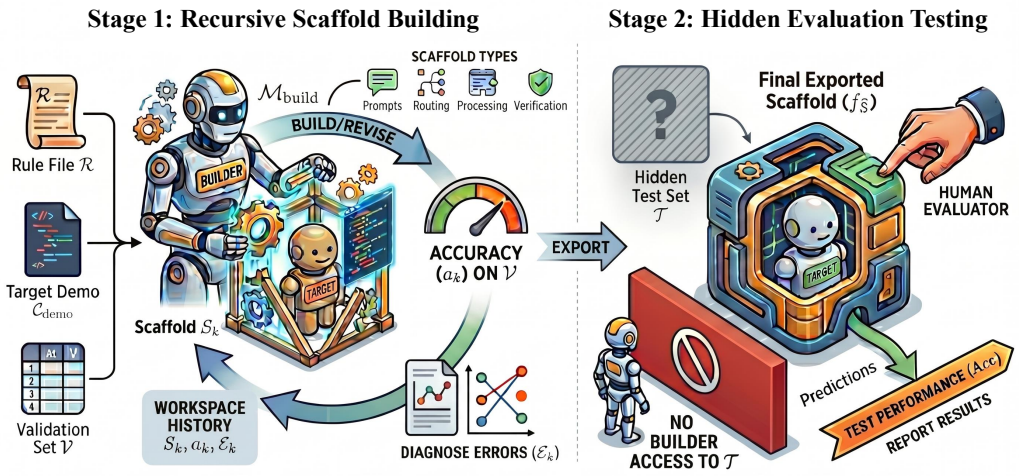


Figure 1: Overview of the **Strong-to-Weak Scaffolding** evaluation framework. During *recursive scaffold building*, the builder model iteratively refines a scaffold to improve target-model performance on validation sets. During *hidden evaluation testing*, the final scaffold is evaluated on the full hidden test set to measure target-model performance.

routing, and validation, as a first-class object of optimization. DSPy formalizes LM pipelines as declarative modules and compiles prompts or demonstrations against task metrics (Khattab et al., 2023). SWE-agent shows that the agent-computer interface itself can substantially affect coding-agent performance (Yang et al., 2024). More recent work makes scaffold design itself an optimization target: ADAS frames agentic systems as code-defined artifacts that can be discovered by a meta-agent (Hu et al., 2025), Meta-Harness searches over harness code using prior candidates, traces, and scores (Lee et al., 2026), and Harness-Bench evaluates how harness configurations affect realistic agent workflows under shared environments and budgets (Yao et al., 2026). Survey work on code as agent harness further argues that code is becoming the operational substrate for state, verification, tool use, and feedback-driven control in agentic systems (Ning et al., 2026). Our work positions itself within this harness-engineering literature but isolates a specific strong-to-weak transfer regime, measuring how builder capability, builder reasoning effort, platform choice, validation budget, target-model headroom, and deterministic offloadability jointly shape the harness quality.

Theory-of-Mind evaluation and mental-state scaffolds. Theory-of-Mind benchmarks provide a demanding testbed for scaffolding because they require tracking observations, beliefs, intentions, hidden information, and nested perspectives. BigToM procedurally generates social-reasoning evaluations from causal templates and shows that strong models may partially mirror human inference patterns while remaining unreliable (Gandhi et al., 2023). Hi-ToM emphasizes higher-order recursive belief reasoning and finds that LLM performance declines as recursion depth increases (Wu et al., 2023). MMToM-QA evaluates belief and goal inference in household activity settings and introduces Bayesian inverse planning accelerated by language models (Jin et al., 2024), while MuMA-ToM extends ToM evaluation to embodied multi-agent interactions with goals, beliefs, and beliefs about others' goals (Shi et al., 2025). Broader ToM evaluations such as ToMBench expand coverage across social-cognitive abilities (Chen et al., 2024), and UserHarness shows that explicit reconstruction of user beliefs, intentions, observations, and actions can provide a strong human-designed ToM harness (Qian et al., 2026). Our work does not intend as a new ToM benchmark or a manually designed ToM solver, but as a meta-evaluation of whether strong models can automatically discover reusable ToM scaffolds for weaker models and of where such scaffolds still fail when belief recursion or Bayesian goal inference resists compilation into rules and skills.

3 Method

We study an automatic harness-building setting in which a strong *builder* model constructs an inference-time scaffold for a fixed weaker *target* model. Let M_{tar} denote the target model and let M_{build} denote the builder

model. For each benchmark $\mathcal{D}^{(j)}$, we randomly sample a small validation split $\mathcal{V}^{(j)} \subset \mathcal{D}^{(j)}$ containing 5% of the benchmark examples, and reserve the remaining examples as a hidden test split $\mathcal{T}^{(j)}$. We write

$$\mathcal{V} = \bigcup_j \mathcal{V}^{(j)}, \qquad \mathcal{T} = \bigcup_j \mathcal{T}^{(j)}.$$

The builder has access only to $\mathcal{V}$ during scaffold construction, while $\mathcal{T}$ is held out and evaluated separately after the build process. At the beginning of each run, the builder is placed inside an existing agentic coding harness $\mathcal{H}_{\text{build}}$ and given an initial workspace

$$\mathcal{W}_0 = \{\mathcal{R}, \mathcal{C}_{\text{demo}}, \mathcal{V}\},$$

where $\mathcal{R}$ is a rule file describing the task instructions and submission format, $\mathcal{C}_{\text{demo}}$ is a demonstration file showing how to call M_{tar}, and $\mathcal{V}$ is the labeled validation set. The builder is not constrained to a fixed scaffold architecture. It may implement any inference-time procedure, including prompt templates, benchmark routing, deterministic pre- or post-processing, answer-format enforcement, verification passes, few-shot retrieval, or direct symbolic solvers. The only requirement is that the final scaffold exposes an entry point that can be applied to unseen test examples.

Conceptually, the builder searches over a space of possible scaffolds $\mathcal{S}$:

$$S^\star = \arg\max_{S \in \mathcal{S}} \text{Acc}\big(S, M_{\text{tar}}; \mathcal{T}\big),$$

but since $\mathcal{T}$ is hidden, the builder can only use validation performance as a proxy:

$$\hat{S} = \arg\max_{S \in \mathcal{S}_{\text{build}}} \text{Acc}\big(S, M_{\text{tar}}; \mathcal{V}\big).$$

A successful scaffold must therefore identify reusable task structure from the validation slice and transfer it to the hidden test set. After the builder finishes, a human evaluator runs the exported entry point on $\mathcal{T}$ without further builder intervention. Please refer to Algorithm 1 for more details.

Algorithm 1: Strong-to-weak scaffold building algorithm.

Require: Builder model M_{build}, target model M_{tar}
Require: Builder-side harness $\mathcal{H}_{\text{build}}$
Require: Rule file $\mathcal{R}$, target demo $\mathcal{C}_{\text{demo}}$, validation set $\mathcal{V}$
Require: Hidden full test set $\mathcal{T}$ ▷ Not visible to builder

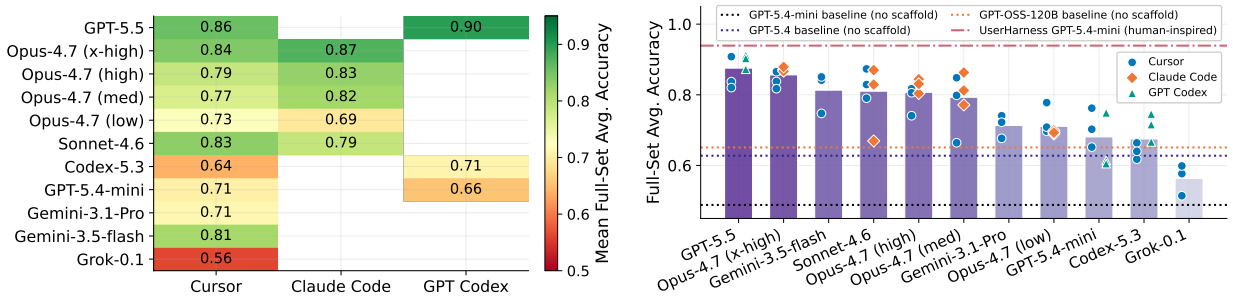
Initialize builder workspace $\mathcal{W}_0 \leftarrow \{\mathcal{R}, \mathcal{C}_{\text{demo}}, \mathcal{V}\}$
Initialize scaffold $S_0 \leftarrow \emptyset$ and index $k \leftarrow 0$
while M_{build} scaffold is not submitted **do**
  Step 1: Inspect task resources
  M_{build} reads $\mathcal{R}$, $\mathcal{C}_{\text{demo}}$, $\mathcal{V}$ ▷ Understand task
  Step 2: Propose or revise scaffold
  $S_k \leftarrow M_{\text{build}}(\mathcal{W}_k)$ ▷ Implement scaffold
  Step 3: Evaluate on validation set
  $\hat{Y}_k^{\mathcal{V}} \leftarrow S_k(M_{\text{tar}}, \mathcal{V})$
  $a_k \leftarrow \text{Acc}(\hat{Y}_k^{\mathcal{V}}, Y^{\mathcal{V}})$ ▷ Call target model
  Step 4: Diagnose and improve
  $\mathcal{E}_k \leftarrow \{(x, y, \hat{y}) \in \mathcal{V} : \hat{y} \neq y\}$
  $\mathcal{W}_{k+1} \leftarrow \mathcal{W}_k \cup \{S_k, a_k, \mathcal{E}_k\}$ ▷ Refine scaffold
  $k \leftarrow k + 1$
end while
Step 5: Export test-time entry point
$\hat{S} \leftarrow S_k$
Builder submits an executable entry point $f_{\hat{S}}(x; M_{\text{tar}})$
Step 6: Hidden evaluation
Human evaluator runs $\hat{Y}^{\mathcal{T}} \leftarrow f_{\hat{S}}(\mathcal{T}; M_{\text{tar}})$
return Test performance $\text{Acc}(\hat{Y}^{\mathcal{T}}, Y^{\mathcal{T}})$

4 Experimental Setup

Task and metric. The task aggregates four ToM datasets into a 3900-item hidden test set, including:

- **BigToM** (Gandhi et al., 2023): 1200 data points; binary belief/goal/action questions hinging on whether an agent observed a world change.
- **Hi-ToM** (Wu et al., 2023) 1200 data points; nested belief questions of recursion order 0–4, with deception and multi-room object tracking.
- **MMToM-QA** (Jin et al., 2024): 600 data points; binary Bayesian goal/belief inference from action trace.
- **MuMA-Tom** (Shi et al., 2025): 900 data points; 3-choice multi-agent belief/social-goal/belief-of-goal questions.

We employ only the text format question of every task. Each builder additionally receives a 195-item (5%) validation sample drawn by a fixed random seed. The primary metric is the unweighted *macro average* of



(a) Builder × platform mean accuracy.

(b) Builder ranking with per-run variation.

Figure 2: Overview of the GPT-5.4-mini as target main results. (a) The mean accuracy of each evaluated builder–platform configuration, with blank cells indicating configurations that were not run. (b) The builder-level mean accuracy, with individual runs displayed as platform-colored markers. The dashed line denotes the reference baselines.

the four per-benchmark full set accuracies; the number of validation evaluation uses is a secondary criterion (efficiency).

Experiment design. We control the following hyper-parameters for each experiment run:

- **Platform**: The harness that the builder model itself runs inside, including Cursor, Claude Code, and GPT Codex.
- **Builder Model**: The model that writes the scaffold for the targets, including Opus-4.7 (with different reasoning efforts from highest to lowest), Sonnet-4.6, GPT-5.5, GPT-5.4-mini, Codex-5.3, Gemini-3.1-Pro, Gemini-3.5-flash, and Grok-0.1. All the other models besides Opus-4.7 is experimented with highest reasoning effort.
- **Target Model**: The weaker model that is being scaffolded for performance improvement, including GPT-5.4-mini and Gemini-3.5-flash.
- **Repeats**: We repeat each experiment setting 3 times to investigate into the harness stability.

This yields in total 72 experiment runs. For the main setting, we use GPT-5.4-mini as the dominant target and serves as the common control for most controlled comparisons; Gemini-3.5-flash is used for the target-model only to contrast different target model's impact.

Baselines. We compare each scaffolded target model against two baseline settings:

- **Vanilla**: Each target model is called directly with the same naive prompt, without any task-specific scaffolding. This setting yields a macro-average accuracy of 0.488 for GPT-5.4-mini and 0.761 for Gemini-3.5-flash.
- **Human-Inspired Harness**: Each target model is evaluated with UserHarness (Qian et al., 2026), a human-designed harness framework for ToM problems. This setting yields a macro-average accuracy of 0.939 for GPT-5.4-mini and 0.941 for Gemini-3.5-flash.

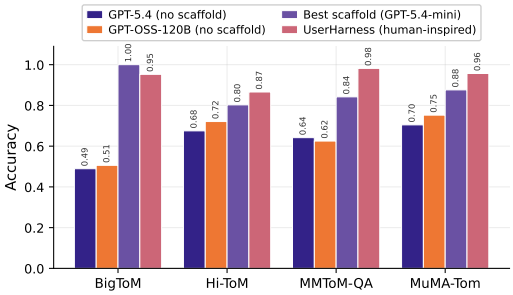
The Vanilla baseline measures the performance that scaffolding is expected to improve upon, while the human-inspired harness provides a human-designed reference point for harness effectiveness.

5 Results and Analysis

Before the detailed analyses, we first summarize the experimental design and main empirical patterns for the GPT-5.4-mini as the target model. Figure 2(a) reports the mean accuracy of each evaluated builder–platform configuration across three repeated runs. Figure 2(b) aggregates the same results by builder: each bar denotes the builder-level mean, while individual markers show the corresponding runs across platforms. We include four reference baselines for comparison: the no-scaffold baselines of GPT-5.4-mini, GPT-OSS-120B, and the stronger GPT-5.4 model, together with the human-inspired UserHarness scaffold applied to GPT-5.4-mini.

Scaffold builder	R	BigToM	Hi-ToM	MMToM	MuMA	Avg. ($\pm$sd)	Δ
Baseline (no scaffold)	–	0.503	0.569	0.412	0.469	0.488	–
GPT-5.5	6	1.000	0.803	0.842	0.857	**0.875** $\pm$ 0.036	+0.387
Opus-4.7 (x-high)	6	0.970	0.791	0.788	0.876	**0.856** $\pm$ 0.022	+0.368
Gemini-3.5-flash	3	0.986	0.712	0.778	0.777	**0.813** $\pm$ 0.047	+0.325
Sonnet-4.6	6	0.977	0.712	0.742	0.810	**0.810** $\pm$ 0.069	+0.322
Opus-4.7 (high)	6	0.922	0.739	0.777	0.791	**0.807** $\pm$ 0.033	+0.319
Opus-4.7 (med)	6	0.944	0.699	0.751	0.778	**0.793** $\pm$ 0.065	+0.305
Gemini-3.1-Pro	3	0.910	0.732	0.618	0.593	**0.713** $\pm$ 0.027	+0.225
Opus-4.7 (low)	6	0.887	0.688	0.609	0.659	**0.711** $\pm$ 0.031	+0.222
GPT-5.4-mini	6	0.981	0.649	0.619	0.474	**0.681** $\pm$ 0.062	+0.193
Codex-5.3	6	0.983	0.625	0.563	0.528	**0.675** $\pm$ 0.043	+0.187
Grok-0.1	3	0.613	0.592	0.537	0.511	**0.563** $\pm$ 0.036	+0.075

(a) Main results by builder model.



(b) Per-benchmark comparison.

Figure 3: Main results by builder model and benchmark with GPT-5.4-mini as target. (a) The performance by builder model, averaged over platforms×repeats; R denotes the number of runs pooled. (b) The per-benchmark comparison between the best scaffold and three references: raw no-scaffold GPT-5.4, raw no-scaffold GPT-OSS-120B, and the human-inspired harness.

Three observations are immediately visible and recur throughout the analysis. First, every evaluated builder–platform configuration substantially exceeds the GPT-5.4-mini no-scaffold baseline, showing that scaffolding consistently improves the weak target model. Second, the dominant source of variation is the builder model rather than the platform: builders form a clear vertical ordering, whereas platform-level differences within the same builder are comparatively small. Third, many scaffolded GPT-5.4-mini configurations surpass the no-scaffold GPT-5.4 baseline. This indicates that a well-designed scaffold can sometimes yield gains larger than upgrading to a stronger unscaffolded model, while the remaining gap to the human-inspired scaffold shows that automated scaffolding still has room for improvement.

5.1 Aspect 0: Main Results

Setting. We fix the target model to GPT-5.4-mini, whose no-scaffold direct-call baseline has a macro-average accuracy of 0.488. For each builder model, we aggregate all the runs across platforms and repeats, and report both per-benchmark accuracy and macro-average accuracy. The no-scaffold direct-call result serves as the reference baseline.

Results. We present full results in Figure 3(a) and representative statistics in Table 1. Strong-to-weak scaffolding yields a large and uniformly positive effect. Across all 57 scaffolded GPT-5.4-mini runs, the mean macro-average accuracy is 0.763, corresponding to an uplift of +0.275 over the baseline, and **100% of runs exceed the baseline**. On average, every one of the 11 builder configurations improves over the baseline. The best individual run, produced by GPT-5.5 on GPT Codex, reaches 0.912, an uplift of +0.423 (87% relative). Thus, scaffolding can lift a model that performs poorly on several tasks in the direct-call setting to near-ceiling performance.

Table 1: Headline numbers of the main results, with target model GPT-5.4-mini.

Metric	Value
GPT-5.4-mini vanilla baseline	0.488
GPT-5.4 vanilla baseline	0.619
GPT-5.4-mini human-inspired	0.939
Mean over all scaffolded runs	0.763 (+0.275)
Best run (GPT-5.5, GPT Codex)	0.912
– uplift over baseline	+0.423 (86.7%)
Builder beating vanilla baseline	100%

Figure 3(b) reports the per-benchmark results against three reference points: the raw no-scaffold GPT-5.4 and GPT-OSS-120B baselines, and the human-designed UserHarness using the same GPT-5.4-mini backbone. The best automatically built scaffold outperforms both vanilla baselines on all four benchmarks, showing that scaffolding a smaller model can exceed the gains from simply moving to a much larger unscaffolded model. Relative to UserHarness, however, performance remains task-dependent. On BigToM, where the reasoning shortcut is explicit and can be consistently exploited, the automated scaffold reaches near-ceiling performance and slightly exceeds UserHarness (1.00 vs. 0.95). Clear gaps remain on the more demanding benchmarks: Hi-ToM (0.80 vs. 0.87), MMToM-QA (0.84 vs. 0.98), and MuMA-ToM (0.88 vs. 0.96). Thus, the remaining gap is concentrated in settings where the relevant

reasoning is less amenable to compilation into deterministic structure, and where careful human harness engineering continues to provide an advantage.

Insight. The main result shows that strong-to-weak scaffolding yields a large and robust improvement. Since the builder never sees the hidden test set, the +0.275 mean uplift indicates that the learned scaffold designs transfer beyond the validation slice. The strongest scaffold lifts GPT-5.4-mini above raw GPT-5.4 and GPT-OSS-120B on every benchmark, and matches the human-inspired harness on the structured BigToM task. This suggests that, when sub-problems are compilable, a builder's reasoning can be converted into reusable inference-time structure. Where such compilation is harder, however, the remaining gap to human-inspired harness persists.

5.2 Aspect 1: Run-to-Run Stability

Setting. We assess scaffold reproducibility by measuring the variance of the final full-set macro-average across independent repeats within the same setting, defined by the same platform, builder, and target model (GPT-5.4-mini).

Results. The standard deviation in Figure 3(a) and the visualization in Figure 4 shows that the scaffold building procedure is fairly stable across platforms and builders. The mean standard deviation of all the macro-average is 0.036, roughly an order of magnitude smaller than the +0.275 mean uplift. At the same time, the widest setting has a repeat range of 0.201, indicating that the build process is still not fully deterministic.

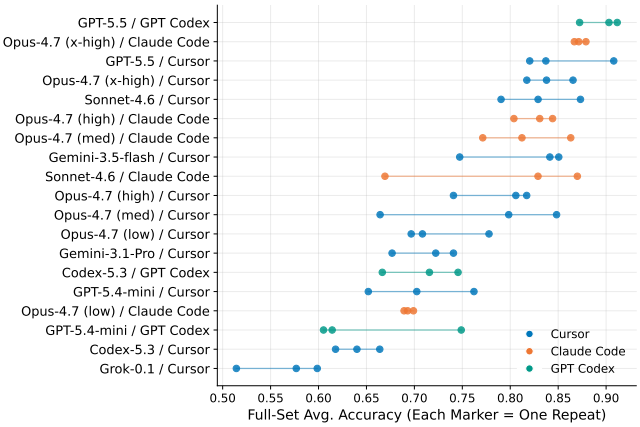


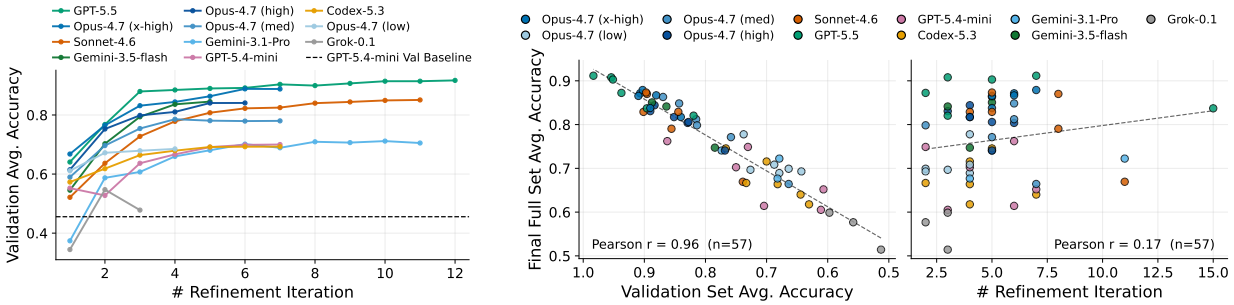
Figure 4: Each marker shows one repeat's full-set macro-average for a platform–builder combination setting, sorted by the average score; tight clusters indicate reproducible builds, while long bars indicate occasional weak scaffold builds.

Insight. Reproducibility is strong but not perfect, and the remaining instability is informative. Through case by case investigation, we discover that the largest spreads actually occur in settings where builders pursue *deterministic-solver* strategies: a single logic error in a benchmark-specific rule can shift accuracy by tens of points over a 1000+ item benchmark full set. Prompt-only scaffolds are typically more stable, but they also deliver smaller gains. This points to a practical recipe: because failures are usually visible on validation and variance is moderate, building two or three scaffolds and selecting the best validation performer offers a low-cost way to capture the upper bound of a setting's performance range.

5.3 Aspect 2: Refinement on Validation

Setting. We analyze the builder's record of its refinement trajectory on the validation set. Specifically, we capture how many validation evaluations the builder performed, how validation accuracy changed across iterations, and whether additional refinement translated into stronger final full-set performance. The target model is GPT-5.4-mini.

Results. Builders use validation evaluations sparingly, as encouraged by the secondary scoring criterion: the mean number of validation passes is 4.9 (median 5; range 2–15). As shown in Figure 5(a), within individual runs, refinement is productive: mean validation accuracy increases by 0.216 from the first logged iteration to the best logged iteration. Figure 5(b) shows two complementary patterns against the same full-set performance axis. First, the best validation score is a strong proxy for held-out performance, tracking final full-set accuracy nearly one-to-one across runs (Pearson $r = 0.96$, left panel), with only a small optimism gap on average (mean 0.021). Thus, the 5% validation sample effectively guides refinement without inducing substantial overfitting. Second, the amount of refinement itself is not predictive: the number of validation



(a) Mean validation trajectory per builder. **(b)** Validation accuracy and refinement budget vs. final full-set accuracy.

Figure 5: Illustration of builder's refinement dynamics. (a) Validation accuracy increases over iterations for each builder, using carry-forward averages across repeats. (b) Two scatter plots share the same full-set accuracy axis: the best validation score closely tracks final full-set performance ($r = 0.96$), whereas the number of refinement iterations is largely unrelated to it ($r = 0.17$).

iterations is essentially uncorrelated with final full-set accuracy (Pearson $r = 0.17$, right panel). In short, builder quality matters much more than how often the builder probes the validation set.

Insight. First, the near one-to-one relationship between best validation accuracy and final full-set accuracy, together with the small optimism gap, supports the study design: a frugal 5% validation slice provides a faithful proxy for the hidden set, guiding refinement without substantial overfitting. Second, the flat relationship between validation budget and final performance shows that the limiting factor is not the amount of feedback, but the quality of the builder's hypotheses. Strong builders reach effective scaffolds through only a few principled refinements, whereas weaker builders do not reliably improve by probing the validation set more often. For scaffolding, therefore, the relevant form of test-time compute is not simply repeated validation querying, but the reasoning used to refine the scaffold itself.

Table 2: Refinement statistics by the builder model, including validation runs, first/best validation accuracy, accuracy gain, and the validation−full optimism gap (positive indicates validation over-estimated the full set).

Builder	Val. Runs	First	Best	Gain	Val−Full Gap
Sonnet-4.6	6.8	0.521	0.856	0.334	0.045
Gemini-3.1-Pro	6.7	0.374	0.712	0.338	-0.002
GPT-5.5	5.8	0.641	0.924	0.283	0.048
Opus-4.7 (x-high)	5.5	0.668	0.888	0.221	0.032
Opus-4.7 (med)	5.2	0.590	0.794	0.204	0.001
GPT-5.4-mini	4.7	0.552	0.711	0.159	0.030
Opus-4.7 (high)	4.5	0.615	0.842	0.227	0.035
Codex-5.3	4.3	0.573	0.693	0.120	0.018
Gemini-3.5-flash	4.0	0.545	0.845	0.300	0.032
Opus-4.7 (low)	3.2	0.611	0.690	0.078	-0.021
Grok-0.1	2.7	0.344	0.556	0.212	-0.007

5.4 Aspect 3: Scaffolding Techniques

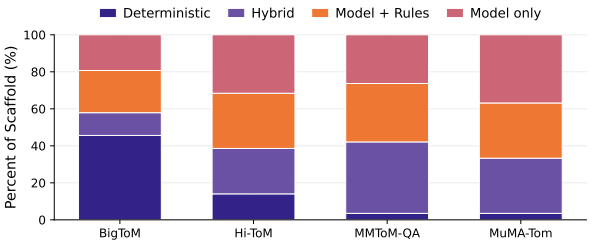
Setting. We analyze every run by reading its scaffold code and optimization log, then coding the final scaffold using a fixed twelve-technique taxonomy based on our observation. This structured extraction covers all 72 main setting's runs. We report three levels of prevalence for scaffolds that uses GPT-5.4-mini as the target: the share of scaffolds using each technique, each builder's self-declared *primary lever*, and the per-benchmark solution approach, categorized as deterministic, hybrid, model+rules, or model-only.

Results. From Figure 6(a), we observe that two techniques are nearly universal: robust *format enforcement*, which reliably parses the answer option, and *greedy decoding*, implemented with temperature 0. The next most common techniques are *benchmark routing* and *forced chain-of-thought*. More complex strategies are used less often: *deterministic solvers*, *self-consistency voting*, and *verification/arbiter* passes appear only in a minority of scaffolds, while *few-shot* prompting is also rare. The per-benchmark analysis in Figure 6(b) shows that technique choice is strongly task-dependent. BigToM is often solved deterministically or with hybrid rules, since its question often exposes the observed/unobserved distinction, whereas MuMA-ToM is almost always handled by the model itself with additional rules.

Insight. The taxonomy suggests that effective scaffolding is less about exotic inference tricks than about disciplined task engineering. The most common techniques, such as format enforcement, greedy decoding, routing, and forced CoT, serve as inexpensive reliability controls: they prevent the weak target model from losing accuracy to malformed outputs, format confusion, or sampling variance. The techniques that most differentiate strong scaffolds require deeper task analysis. Deterministic solvers, structured state extraction, and polarity logic are useful only when the benchmark exposes enough regularity for the builder to convert reasoning into executable structure. In this sense, the per-benchmark approach map functions as a *compilability ranking* of the ToM tasks: BigToM admits substantial rule-based compilation, whereas MuMA-ToM remains largely model-mediated. This distinction explain both the performance attribution results and the residual-error patterns in later analysis.

Technique	Prevalence	Percent of Runs
Format enforcement	57/57	100%
Greedy / temp control	56/57	98%
Benchmark routing	54/57	95%
Forced CoT	45/57	79%
Polarity / negation logic	45/57	79%
Token-budget tuning	43/57	75%
Hybrid fallback	34/57	60%
Deterministic solver	31/57	54%
Structured extraction	29/57	51%
Few-shot examples	12/57	21%
Verification / arbiter	7/57	12%
Self-consistency vote	3/57	5%

(a) Technique prevalence statistics across runs.



(b) Solution approach across benchmarks.

Figure 6: Technique prevalence and solution approach statistics across runs and benchmarks.

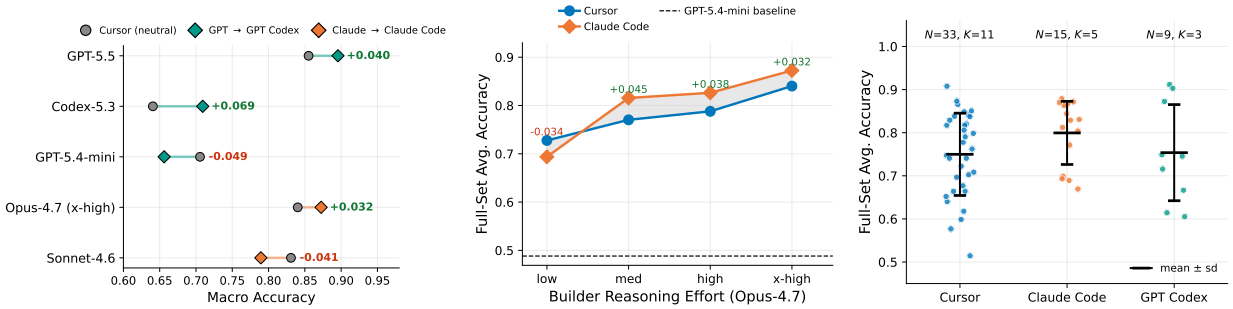
5.5 Aspect 4: Harness Platform's Impact

Setting. The platform refers to the agentic coding environment in which the builder writes and refines the scaffold. Each builder family has a *native* platform from the same vendor: GPT and Codex builders are native to GPT Codex, while Opus and Sonnet builders are native to Claude Code. Cursor serves as a neutral third-party platform shared across builders. Holding the target model fixed at GPT-5.4-mini, we ask whether builders perform better on their native platform or not. Figure 7 evaluates this question from three perspectives: a matched native-vs-Cursor comparison for builders run on both platforms, an Opus-4.7 platform×effort comparison across reasoning-efforts, and a pooled run distribution across platforms.

Results. (i) *Native-platform gains are small and inconsistent.* In the matched comparison in Figure 7(a), each builder's Cursor run is paired with its native-platform run. The short connectors point in both directions, indicating no systematic native-platform advantage. Averaged across the eight matched configurations, moving from Cursor to the native platform changes macro accuracy by only +0.013, with the native platform winning in 5 of 8 cells (paired permutation test $p = 0.484$). The effect is somewhat larger for the GPT family (+0.020 on GPT Codex, driven mainly by Codex-5.3 at +0.069) than for the Claude family (+0.008 on Claude Code), but both effects are much smaller than the across-builder variation reported in the previous section. Moreover, each family also contains a counterexample: GPT-5.4-mini and Sonnet-4.6 both perform worse on their native platform after averaging parallel runs.

(ii) *Platform advantages emerge only when the builder has enough reasoning budget to use them.* We show in Figure 7(b) the more informative pattern is a platform×effort interaction rather than a uniform platform effect. For Opus-4.7, Claude Code trails Cursor at low effort (−0.034), but leads once the builder is allowed to deliberate more extensively: medium +0.045, high +0.038, and extra-high +0.032. Thus, the native harness does not automatically improve the scaffold; its advantage materializes only when the builder can fully exploit its affordances.

(iii) *Pooled platform averages mostly reflect builder composition.* The platform marginal in Figure 7(c) should therefore be interpreted descriptively rather than causally. The three platforms host different builder rosters ($K = 11$, 5, and 3 builders), so their average scores conflate platform effects with builder selection. Claude Code has the highest marginal mean (0.799 vs. GPT Codex 0.754 and Cursor 0.750), but this mainly reflects its Opus/Sonnet-heavy roster rather than a clean platform advantage. The broad overlap among the platform clouds reinforces the main message: builder identity dominates platform identity.



(a) Matched Native-vs-Cursor comparison. **(b)** Platform×effort interaction. **(c)** Pooled platform distributions.

Figure 7: Platform effects with GPT-5.4-mini as the target. (a) Comparison of each builder's neutral Cursor run with its native-platform run, showing that matched native advantages are small and inconsistent. (b) We isolate Opus-4.7 across the full effort ladder, and shows the native-platform advantage emerges only at higher reasoning effort. (c) The pooled run distributions by platform. Overall, platform matters primarily as a second-order, conditional factor than as the main driver of scaffold quality.

Insight. The harness platform is a second-order factor, and its effect is conditional rather than universal. The matched comparisons provide little evidence for a general "builders perform best on their own platform" rule: the native advantage averages only +0.013, is not statistically reliable, and even reverses for some builders. Likewise, the apparent platform ranking in the pooled view mainly reflects roster composition rather than a causal platform effect. The more substantive finding is the platform×effort interaction: a native harness helps only when the builder has enough reasoning budget to exploit its affordances. Practically, this makes the strong-to-weak recipe more portable than platform-specific explanations would suggest: the central determinants are still builder capability and reasoning effort, not the coding environment itself. At the same time, platform-specific tuning may still matter at the frontier, where a capable high-effort builder can convert better tooling into better scaffold design.

5.6 Aspect 5: Analysis Across Target Models

Setting. We vary the target model from GPT-5.4-mini to include Gemini-3.5-flash, while holding the platform fixed to Cursor, so the only changing factor is *which model is being scaffolded*. Five builders (Opus-4.7, GPT-5.5, Gemini-3.1-Pro, Gemini-3.5-flash, and Grok-0.1) were run against both targets, with three repeats each, yielding matched within-builder comparisons. The two targets begin from very different baselines: GPT-5.4-mini is weak overall (0.488), whereas Gemini-3.5-flash is already strong, especially on Hi-ToM and MuMA-ToM (0.761 overall). We therefore analyze not only the aggregate uplift, but also where the uplift occurs, how builders adapt their strategies, and when scaffolding can become harmful, using the different lenses we developed in previous analysis.

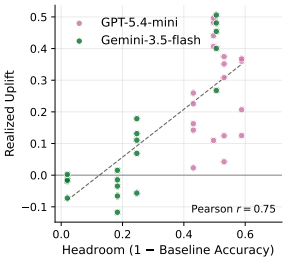
Results. As shown in Figure 8(a), overall, the weaker target benefits much more from scaffolding. Averaged over runs, scaffolding improves GPT-5.4-mini by +0.262 (0.488 → 0.750), but improves Gemini-3.5-flash by only +0.110 (0.761 → 0.871). This direction holds for all 5 builders. The aggregate contrast, however, is only the surface pattern; the per-benchmark results reveal a clearer mechanism.

(i) *Uplift follows a headroom law.* As illustrated in Figure 8(b), across all builder×benchmark×target settings, realized uplift is strongly predicted by the target's available headroom on that benchmark, $1-$baseline (Pearson $r = 0.75$). This suggests that scaffolding primarily recovers latent competence that the target model already possesses but does not reliably deploy, such as following the answer format, tracking observation cues, or maintaining recursive state. It therefore helps most where most correctable errors is left.

(ii) *The location of the gain depends on the target.* In Figure 8(c) we show that, for GPT-5.4-mini, uplift is distributed across all four benchmarks. For Gemini-3.5-flash, however, the gain is concentrated almost entirely on BigToM, which accounts for 96% of its macro uplift (0.42). This concentration is informative:

Builder	Target	Baseline	Scaffolded	Δ
Gemini-3.5-flash	GPT	0.488	0.813	+0.325
Gemini-3.5-flash	Gemini	0.761	0.872	+0.111
Gemini-3.1-Pro	GPT	0.488	0.713	+0.225
Gemini-3.1-Pro	Gemini	0.761	0.881	+0.120
GPT-5.5	GPT	0.488	0.855	+0.367
GPT-5.5	Gemini	0.761	0.901	+0.140
Grok-0.1	GPT	0.488	0.563	+0.075
Grok-0.1	Gemini	0.761	0.780	+0.019
Opus-4.7 (x-high)	GPT	0.488	0.840	+0.352
Opus-4.7 (x-high)	Gemini	0.761	0.923	+0.162

(a) Overall statistics of using the same builder on two target models, with Δ representing the macro uplift.



(b) The headroom law, with each point denotes one builder×benchmark run.

Benchmark	GPT-5.4-mini			Gemini-3.5-flash		
	Base	Scaf.	Δ	Base	Scaf.	Δ
BigToM	0.50	0.89	+0.39	0.49	0.92	+0.42
Hi-ToM	0.57	0.73	+0.16	0.82	0.77	-0.04
MMToM-QA	0.41	0.70	+0.28	0.75	0.84	+0.09
MuMA-Tom	0.47	0.71	+0.24	0.98	0.96	-0.02

(c) Per-benchmark accuracy uplift.

Target	Det.-Solver Prevalence	Model-only share by benchmark			
		BigToM	Hi-ToM	MMToM	MuMA
GPT	40%	27%	40%	27%	40%
Gemini	47%	40%	60%	53%	73%

(d) Scaffold strategy sorted by targets.

Figure 8: Weak-to-strong transfer analysis across target models. (a) Statistics of the same builder on two target models. GPT and Gemini denotes GPT-5.4-mini and Gemini-3.5-flash. (b) The headroom law: each point is one builder×benchmark run, with its realized uplift (scaffolded − baseline accuracy) plotted against the headroom the target leaves on that benchmark (1 − baseline). (c) Per-benchmark statistics of the accuracy uplift. The weak target gains everywhere; the strong target gains essentially only on BigToM and even regresses on the tasks it already handles well. (d) Builder's scaffold strategy sorted by different targets. Against the stronger Gemini, builders rely less on deterministic code and hand more benchmark tasks back to the model itself.

BigToM is both the benchmark where the stronger target still has meaningful headroom and the task whose structure is most readily compiled into rules.

(iii) *Builders adapt their strategy to the scaffolded target.* The coded taxonomy in Figure 8(d) shows that builders use less deterministic machinery when scaffolding Gemini-3.5-flash than when scaffolding GPT-5.4-mini. The share of model-only benchmark handling rises on every task, most sharply on MuMA-ToM (40% → 73%). In other words, builders appear to recognize that the stronger target can already solve Hi-ToM and MuMA-ToM benchmark tasks relatively well, and they reserve heavier rule-based interventions for the remaining compilable headroom.

(iv) *On a strong target, scaffolding can backfire.* For GPT-5.4-mini, no builder regresses below baseline on any benchmark (0/20 matched cases). For Gemini-3.5-flash, by contrast, every builder regresses on at least one benchmark (9/20 cases), especially on tasks where the baseline is already high: Hi-ToM (−0.04 on average) and near-saturated MuMA-ToM (−0.02 on average). This illustrates the risk of over-scaffolding: when the target is already close to ceiling, additional prompts, routing, or rules may disrupt correct behavior more often than they repair errors.

Insight. The target model matters less through its identity than through its *headroom*. Scaffolding acts primarily as a competence-recovery mechanism: its payoff is governed by how much latent ability the target fails to deploy, as reflected in the strong relationship between benchmark headroom and uplift. This turns the apparent rule that "uplift shrinks as the target gets stronger" into a more general principle: scaffolding helps when there are correctable failures left to recover. It also explains the observed adaptation in builder strategy. Capable builders allocate deterministic routing and rule-based machinery to sub-tasks where headroom remains, while backing off to lighter prompting where the target is already reliable. But when headroom is nearly exhausted, scaffolding can cross from assistance into interference, perturbing answers the model would otherwise get right. Practically, strong-to-weak scaffolding is therefore most valuable for weak targets; for already strong targets, it should be applied selectively at the sub-task level and gated by measured headroom.

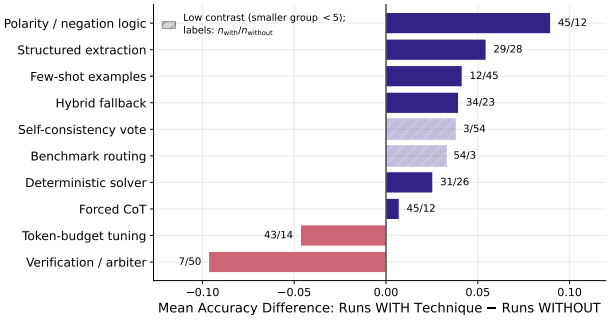
5.7 Aspect 6: Builder Reasoning Effort

Setting. We fix the builder to Opus-4.7 and the target to GPT-5.4-mini, then vary only the builder's *reasoning effort* across four tiers: low, medium, high, and extra-high. Because this sweep is available on both Cursor and Claude Code, it isolates how much the builder deliberates while writing the scaffold from both model identity and platform choice. Part of this comparison result is also shown in Figure 7(b).

Technique	$n_{\mathbf{w/}}$	$n_{\mathbf{w/o}}$	Acc. w/	Acc. w/o	Δ	Rel.
Polarity / negation logic	45	12	0.782	0.693	+0.090	
Structured extraction	29	28	0.790	0.736	+0.055	
Few-shot examples	12	45	0.796	0.755	+0.042	
Hybrid fallback	34	23	0.779	0.740	+0.040	
Self-consistency vote	3	54	0.799	0.761	+0.038	†
Benchmark routing	54	3	0.765	0.732	+0.033	†
Deterministic solver	31	26	0.775	0.750	+0.026	
Forced CoT	45	12	0.765	0.758	+0.007	
Token-budget tuning	43	14	0.752	0.799	-0.047	
Verification / arbiter	7	50	0.679	0.775	-0.097	
Greedy / temp control	56	1	0.762	0.871	-0.110	†
Format enforcement	57	0	0.763	–	–	†

† low contrast: smaller group < 5 runs; Δ in this case is unreliable.

(a) Mean accuracy of runs using vs. not using each technique.



(b) Association between each technique and accuracy.

Figure 9: Technique-level attribution analysis. (a) Mean accuracy of runs using each technique vs. not using. n_{with} denotes the number of runs using the technique. Associational/universal techniques have small Δ for lack of a contrast group. (b) The technique's association with accuracy: difference in mean full-set accuracy between runs that use each technique and runs that do not.

Results. Greater builder reasoning effort consistently improves scaffold quality. As shown in Table 3, macro accuracy increases monotonically on both platforms as effort rises. Pooled across platforms, performance moves from 0.711 at low effort to 0.793, 0.807, and 0.856 at extra-high effort, yielding a strong monotone relationship between effort tier and per-run accuracy (Spearman $\rho = 0.77$). The extra-high tier significantly outperforms both the adjacent high tier (0.856 vs. 0.807; permutation test $p = 0.013$) and the low tier ($p = 0.002$), indicating that the trend is not driven by noise. The largest gain occurs from low to medium effort, while higher tiers provide smaller but still positive improvements. This suggests that most of the benefit comes from giving the builder enough deliberation to identify a viable scaffold strategy, with additional reasoning refining rather than transforming that strategy.

Table 3: Detailed statistics of Opus-4.7 reasoning-effort sweep (with target GPT-5.4-mini) on both platforms, in ascending effort order. "Val. Runs" and "Py LOC" show how effort also changes the scaffold building process instead of just its score.

Platform	Effort	R	Avg. (±sd)	Val. Runs	Py LOC
Cursor	low	3	0.728 ± 0.036	3.7	653
Cursor	medium	3	0.770 ± 0.078	5.0	1037
Cursor	high	3	0.788 ± 0.034	4.7	972
Cursor	extra-high	3	0.840 ± 0.020	4.7	1274
Claude Code	low	3	0.694 ± 0.004	2.7	510
Claude Code	medium	3	0.816 ± 0.038	5.3	685
Claude Code	high	3	0.826 ± 0.017	4.3	870
Claude Code	extra-high	3	0.872 ± 0.005	6.3	987

Insight. Builder reasoning is a genuine monotonic lever: the more Opus-4.7 deliberates while *writing* the scaffold, the stronger the resulting scaffold becomes, with no evidence of over-engineering even at extra-high effort. This contrasts with our previous analysis, which shows additional validation evaluations did not predict final quality. The useful compute is therefore not more probing, but deeper hypothesis formation about the task structure. The returns are diminishing but remain positive: the largest gain comes from low to medium effort, suggesting that moderate deliberation is enough to discover the main structural devices, such as routing, format enforcement, and deterministic solving, while higher tiers add more specialized refinements such as polarity logic and belief-state extraction. The accompanying growth in scaffold size (∼510–650 LOC at low effort vs. ∼1000–1300 at extra-high) also supports this interpretation: greater builder reasoning compiles more task logic into the harness.

5.8 Aspect 7: Attribution of Why Does Improvement Happen?

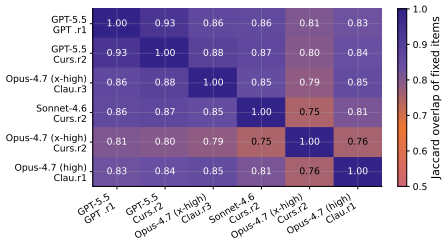
Setting. We attribute uplift to scaffold techniques by comparing, for each taxonomy item, the mean full-set accuracy of GPT-5.4-mini-target runs that use the technique with those that do not. We also conduct three cross-checks: a paired McNemar test comparing the best scaffold to the no-scaffold baseline, a complementarity analysis measuring whether different scaffolds fix the same or different errors, and a comparison between self-scaffolding and stronger-builder scaffolding. Because techniques often co-occur, the

Target	Best Scaffold	Base	Scaf.	Fixed	Broke	χ^2	p
GPT-5.4-mini	GPT-5.5/GPT Codex	0.488	0.912	1717	105	1424.4	$< 10^{-4}$
Gemini-3.5-flash	Gemini-3.5-flash/Cursor	0.761	0.939	772	63	600.3	$< 10^{-4}$

(a) Statistical significance of the best scaffold compared with the baseline.

Platform	Builder	R	Avg. (±sd)	Δ	Platform	Builder	R	Avg. (±sd)	Δ
Cursor	Self	3	0.706 ± 0.045	+0.217	Codex	Self	3	0.656 ± 0.066	+0.168
	Others	30	0.754 ± 0.098	+0.266		Others	6	0.802 ± 0.097	+0.314

(b) Comparison between self-scaffolding and stronger builders.



(c) Complementarity among repaired errors.

Figure 10: Complementarity and builder-strength analyses. (a) Statistical significance of the best scaffold compared with the baseline, measured by paired McNemar tests over 3,900 items; "Fixed" denotes baseline-wrong cases corrected by the scaffold, while "Broke" denotes baseline-right cases made incorrect. (b) Comparison between self-scaffolding, where GPT-5.4-mini builds for itself, and stronger builders under the same platform and target setting; Δ reports the uplift over GPT-5.4-mini's own baseline. (c) Pairwise Jaccard overlap among the baseline-error sets repaired by top scaffolds; although scaffolds share many easy fixes, their union covers more baseline errors than any individual scaffold, indicating complementary repair mechanisms.

technique-level comparisons are associational rather than strictly causal, but they provide a useful lens on which design choices align with higher performance.

Results. (i) *The strongest associations come from techniques that compile task structure into the scaffold.* Figure 9(a) and (b) both compare runs with and without each technique, marking low-contrast comparisons whose smaller group has fewer than five runs. Among the better-powered contrasts, the largest positive associations are *polarity/negation logic* (+0.09), *structured extraction* (+0.06), *hybrid fallback* (+0.04), and *deterministic solving*. These are not generic prompting tricks; they directly target the main failure modes of the ToM benchmarks, including MOST-vs-LEAST framing, belief-state tracking, and offloading predictable sub-problems into code. Apparent negative associations for near-universal techniques such as greedy decoding (56/1) and benchmark routing (54/3) should not be interpreted as harmful effects: their contrast groups are too small and consist of unusually weak non-user runs.

(ii) *The best scaffold produces a large, item-level reliable improvement.* As shown in Figure 10(a), the gain over the GPT-5.4-mini no-scaffold baseline is overwhelmingly significant under a paired McNemar test over the 3900 evaluation items, with $\chi^2 \gg 10^4$ and $p < 10^{-4}$. The direction of the item-level changes is also highly asymmetric: the scaffold fixes 1717 baseline errors while breaking only 105 previously correct items. Thus, the aggregate uplift is not driven by noise or by a small error redistribution, but reflects a broad shift from incorrect to correct predictions.

(iii) *A stronger builder is not necessary for uplift, but it is necessary for the highest gains.* The self-scaffold control in Figure 10(b), where GPT-5.4-mini builds a scaffold for itself, already improves performance by +0.17 to +0.22 over the no-scaffold baseline. This shows that even a weak target can use validation feedback and task structure to engineer a useful harness. However, stronger builders achieve substantially larger gains on both platforms, with the largest gap on GPT Codex (+0.31 for stronger builders vs. +0.17 for self-scaffolding). Self-scaffolding therefore recovers some accessible structure, but stronger builders are what unlock the high-performance regime.

(iv) *Different strong scaffolds capture partly different ToM skills.* The complementarity analysis in Figure 10(c) shows that top scaffolds overlap on many easy fixes but diverge on harder cases. The union of items fixed across the top scaffolds covers 97% of all baseline errors, exceeding the coverage of any individual scaffold. This indicates that scaffold designs are not merely redundant variants of the same solution: different builders discover partially distinct mechanisms for repairing the target model's reasoning.

Insight. The gains come from two complementary layers of scaffold design. The first is a reliability floor: format enforcement, routing, and greedy decoding are used by nearly all scaffolds and prevent avoidable errors from malformed outputs, format confusion, or sampling variance. Because these techniques are almost universal, they do not explain variation across runs, but they make higher performance possible. The

second layer is task-structure exploitation: polarity logic, structured belief-state extraction, and deterministic solving distinguish the strongest scaffolds by converting benchmark regularities into explicit inference-time machinery. This interpretation is reinforced by the significance and complementarity analyses. The best scaffold produces a large and reliable item-level improvement, yet different strong scaffolds repair overlapping but non-identical subsets of baseline errors. Thus, the ceiling is not determined by any single technique, but by the residual cases that remain difficult across diverse scaffold designs.

5.9 Aspect 8: Cognitive-Load Reduction

Setting. In the task instructions, builders are explicitly asked to reduce the target model's cognitive load. We operationalize this idea using scaffold provenance tags. After evaluation, every runis labeled with a method that produced it, such as "deterministic", "model_prompt", "rule", or "llm". For all these runs, we compute the determinism fraction: the share of the 3900 evaluation items answered entirely by code or structured rules. We then relate this to final accuracy and examine how it varies by benchmark. The target is GPT-5.4-mini for all runs.

Results. Deterministic offloading is strongly associated with scaffold quality. As shown in Figure 11(a), runs with higher determinism fractions achieve higher final accuracy (Pearson $r = 0.72$): the more work the builder converts into executable structure, the less reasoning burden remains for the weak target model. According to Figure 11(b), this relationship is also highly benchmark-dependent. BigToM is almost fully offloadable, with mean determinism around 0.94, because its questions often expose the observed/unobserved distinction needed for many answers. Hi-ToM is partially offloadable (≈ 0.51), typically through symbolic belief-state tracking. MMToM-QA is similar but slightly lower (≈ 0.44), while MuMA-ToM is least reducible to structured code (≈ 0.36), reflecting the difficulty of compiling free-form dialogue reasoning into deterministic rules. Statistics in Table 4 further shows that scaffold code size is only weakly related to accuracy ($r \approx 0.22$). Thus, what matters is not simply writing more code, but writing code that removes the right cognitive load from the target model.

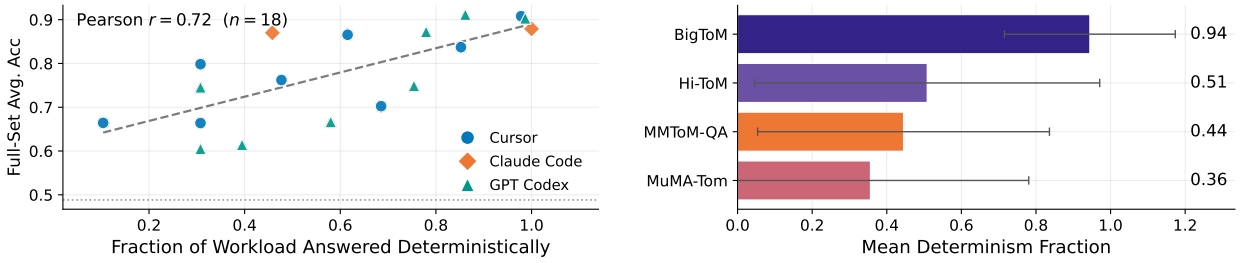
Table 4: Statistics about determinism fraction, final accuracy, and scaffold code size for runs with prediction-provenance tags. Determinism fraction is the share of items answered through deterministic codes and rules.

Builder	Platform	Det. Frac.	Acc.	Py LOC
Opus-4.7 (x-high)	Claude Code	1.00	0.879	1052
GPT-5.5	GPT Codex	0.99	0.903	1285
GPT-5.5	Cursor	0.98	0.908	1026
GPT-5.5	GPT Codex	0.86	0.912	1288
GPT-5.5	Cursor	0.85	0.837	1107
GPT-5.5	GPT Codex	0.78	0.872	1294
GPT-5.4-mini	GPT Codex	0.75	0.749	1170
GPT-5.4-mini	Cursor	0.69	0.702	1462
Opus-4.7 (x-high)	Cursor	0.62	0.865	1293
Codex-5.3	GPT Codex	0.58	0.667	1028
GPT-5.4-mini	Cursor	0.48	0.762	1596
Sonnet-4.6	Claude Code	0.46	0.870	856
GPT-5.4-mini	GPT Codex	0.39	0.614	1129
Codex-5.3	Cursor	0.31	0.664	1026
Codex-5.3	GPT Codex	0.31	0.746	903
GPT-5.4-mini	GPT Codex	0.31	0.605	779
Opus-4.7 (med)	Cursor	0.31	0.798	796
Opus-4.7 (med)	Cursor	0.10	0.664	1553

Insight. This is the mechanistic core of strong-to-weak scaffolding. The builder reduces the weak target model's cognitive load in two ways. The first is *offloading*: deterministic codes and rules answer some items directly without target model's additional reasoning efforts. The second is *structuring*: when the model is still needed, the scaffold narrows the task into a more constrained prompt with clearer inputs, output format, and reasoning focus. Offloading explains much of the variation in scaffold quality ($r = 0.72$), but its feasibility depends on how much of the task can be compiled into explicit structure. In effect, the builder pays a one-time reasoning cost to encode part of the task's decision procedure into the harness; after that, the per-item burden on the weak model is reduced, and the model is reserved for the cases that resist rule or skill compilation.

5.10 Aspect 9: Remaining Error Analysis

Setting. We analyze the residual errors of the 8 strongest GPT-5.4-mini scaffolds, defined as the best repeat from each platform×builder setting with mean accuracy above 0.80. We pool their remaining errors and slice them into multiple subcategories: BigToM by question type and observed/unobserved status, Hi-ToM by recursion order and deception, MMToM-QA by question subtype, and MuMA-ToM by label. We also



(a) Determinism fraction vs. accuracy.

(b) Benchmark-level offloadability.

Figure 11: Deterministic offloading as a measure of cognitive-load reduction. (a) scaffolds answering a larger share of items without target-model calls tend to achieve higher final accuracy. (b) offloadability varies sharply by benchmark: BigToM is almost fully compilable into code, whereas MuMA-ToM remains substantially model-dependent.

decompose each scaffold's impact into baseline errors *fixed* and baseline-correct items *broken* to distinguish genuine improvement from error trade-offs.

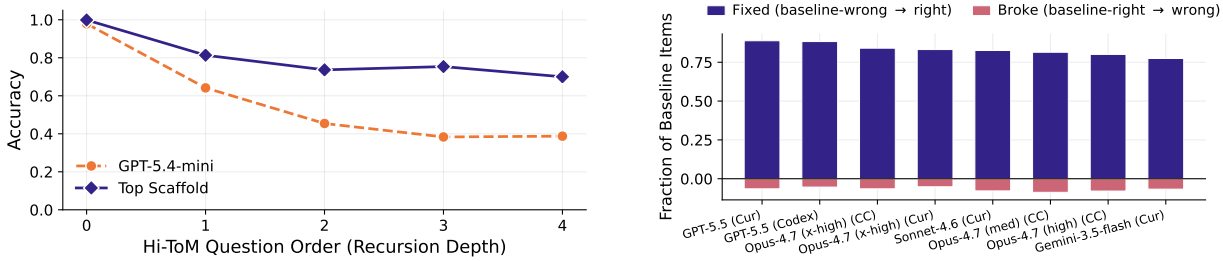
Results. (i) *Top scaffolds are broadly corrective rather than merely redistributive.* As shown in Figure 12(b), the strongest scaffolds fix a large fraction of baseline mistakes while introducing few regressions: on average, they repair 83% of baseline-wrong items and break only 7% of baseline-correct items. This asymmetry indicates that scaffolding is close to Pareto-improving over the baseline rather than simply shifting errors across examples.

(ii) *The remaining errors are concentrated in the least compilable parts of the tasks.* Table 5 shows that BigToM is essentially solved, with accuracy at least 0.95 on every slice. The residual error mass instead clusters in three harder regions. First, as shown in Figure 12(a), Hi-ToM accuracy declines with recursion depth, falling from 0.999 at order 0 to 0.700 at order 4, with deception further reducing performance. Second, MMToM-QA errors concentrate in Bayesian goal-inference subtypes, especially the type-2 "which container/goal" questions. Third, MuMA-ToM remains difficult on social-goal and belief-of-goal labels, even though simpler belief questions are nearly solved.

Table 5: Residual accuracy by fine-grained metadata slice, pooled over the 8 strongest scaffolds. Slices are grouped by benchmark, and n denotes the pooled item count.

Benchmark	Slice	Accuracy	n (pooled)
BigToM	goal/observed	0.955	1600
BigToM	belief/observed	0.980	1600
BigToM	action/observed	0.988	1600
BigToM	goal/unobserved	0.993	1600
BigToM	belief/unobserved	0.996	1600
BigToM	action/unobserved	0.998	1600
Hi-ToM	order 0	0.999	1920
Hi-ToM	order 1	0.814	1920
Hi-ToM	order 2	0.736	1920
Hi-ToM	order 3	0.754	1920
Hi-ToM	order 4	0.700	1920
Hi-ToM	deception=False	0.829	4800
Hi-ToM	deception=True	0.772	4800
MMToM	qtype 2.1	0.680	600
MMToM	qtype 2.4	0.755	600
MMToM	qtype 1.3	0.790	800
MMToM	qtype 1.2	0.800	800
MMToM	qtype 2.3	0.828	600
MMToM	qtype 2.2	0.863	600
MMToM	qtype 1.1	0.929	800
MuMA	social_goal	0.872	1616
MuMA	belief_of_goal	0.880	3968
MuMA	belief	0.985	1616

Insight. The residual error floor marks the boundary of what current scaffolds can compile away. Scaffolding succeeds when the builder can turn recurring structure into deterministic rules, skills, routing logic, or constrained prompts. It struggles when the task requires nested higher-order belief tracking under deception or Bayesian goal inference from ambiguous action traces. In these cases, the scaffold must leave more of the reasoning to GPT-5.4-mini, precisely where the weak target remains least reliable. The strong fix/break asymmetry shows that the scaffolds are genuinely improving the target rather than trading one class of mistakes for another. At the same time, the complementarity result from previous analysis, where the union of top-scaffold fixes covers 97% of baseline errors, suggests that some remaining failures are addressable by combining diverse scaffolds. The deepest recursion and goal-inference errors, however, likely require a stronger explicit belief-tracking mechanism than more variants of the same scaffold design.



(a) Hi-ToM recursion depth.

(b) Fixes versus regressions.

Figure 12: Residual-error structure among the strongest GPT-5.4-mini scaffolds. (a) Hi-ToM accuracy declines as recursion order increases, and that the scaffold advantage narrows on the deepest cases. (b) Results of decomposing each top scaffold's effect into baseline-wrong items fixed and baseline-correct items broken, showing that top scaffolds repair many more errors than introduce.

6 Synthesis and Conclusion

Across 72 runs, the evidence for **strong-to-weak scaffolding** is consistent and mechanistically interpretable. The main conclusions are as follows:

- **Scaffolding produces large and reliable gains** (Aspect 0). The mean uplift over the GPT-5.4-mini no-scaffold baseline is +0.275; 100% of runs and all 11 builder configurations exceed the baseline. The best scaffold reaches 0.912 (+0.423), surpassing the Gemini-3.5-flash no-scaffold baseline on multiple tasks and approaching the human-inspired harness reference on the same backbone (0.939), despite using no human ToM-specific engineering.
- **The procedure is reproducible, though not deterministic** (Aspect 1). The mean within-cell standard deviation is 0.036, roughly an order of magnitude smaller than the main uplift. The remaining variance is concentrated in deterministic-solver strategies, where a single implementation error can substantially affect one benchmark.
- **The method is validation-efficient** (Aspect 2). Builders use a median of 5 validation evaluations, show little evidence of overfitting to the 5% validation slice (mean validation–full gap 0.021), and obtain no clear benefit from additional probing ($r = 0.17$). Builder quality matters more than validation budget.
- **The core mechanism is cognitive-load reduction** (Aspects 3, 7, and 8). Accuracy is strongly associated with the fraction of items answered by deterministic codes, rules and scaffolds ($r = 0.72$). The best scaffolds combine a near-universal reliability floor, including format enforcement, routing, and greedy decoding, with higher-value task-structure exploitation, such as polarity logic, structured extraction, and deterministic solving.
- **Builder capability and effort dominate platform choice** (Aspects 4 and 6). Platform effects are second-order and conditional, whereas builder identity and reasoning effort are primary drivers of scaffold quality. For Opus-4.7, performance often improves monotonically with reasoning effort (Spearman $\rho = 0.77$).
- **The target model matters through headroom** (Aspect 5). Scaffolding helps most when the target leaves correctable errors on the table. As the target becomes stronger, the available headroom shrinks, and scaffolding must be applied more selectively to avoid disturbing or breaking those already-correct behaviors.
- **Remaining errors mark the less-compilable core of benchmark** (Aspect 9). Residual failures concentrate in deep belief recursion under deception and Bayesian goal inference, where reasoning cannot yet be fully reduced to rules or skills. Even so, top scaffolds can still fix about 83% of baseline errors.

Takeaway. Strong-to-weak scaffolding works because a capable builder can act as a *compiler of task competence*. It spends a one-time reasoning budget to identify structure in the task and encode that structure into an inference-time scaffold. Once compiled, a weaker and cheaper target model can execute the task at a level closer to that of much stronger models. This substitution is strongest for structured sub-problems

whose decision procedures can be made explicit, but weaker for the genuinely hard core of benchmark reasoning. In ToM tasks, this includes nested belief tracking, deception, and counterfactual goal inference. Thus, scaffolding does not replace raw reasoning capability; it reallocates it. The builder performs the structural reasoning once, and the target model handles the residual cases that remain model-dependent.

Practically, the results suggest a simple recipe: use the strongest available builder, allocate high reasoning effort during scaffold construction, spend only a modest number of validation evaluations, prioritize cognitive offloading for provable sub-tasks, and, when budget permits, build several independent scaffolds and select or ensemble them to capture complementary repairs.

7 Discussion and Future Work

Benchmark selection. We use ToM benchmarks as a representative testbed rather than as the only setting where strong-to-weak scaffolding should apply. These benchmarks are well studied, contain diverse question types, and span a wide range of difficulty. This mixture is important for our purposes: some items require genuinely hard reasoning, while others expose regularities that a scaffold can exploit to reduce the target model's cognitive load. We do not view such exploitable structure as "cheating." Under a fixed instruction and validation-only setting, discovering these structures is itself part of the builder model's capability. A strong builder should be able to inspect the validation slice, infer which parts of the task are compilable, and convert those observations into reusable inference-time skills. In this sense, the ToM benchmarks are useful precisely because they contain both scaffoldable structure and residual reasoning difficulty. This combination makes builder behavior more diverse and allows us to distinguish shallow prompt optimization from genuine task-structure discovery. Future work should test strong-to-weak scaffolding paradigm on broader benchmark families, especially those with different mixtures of symbolic structure, ambiguity, and open-ended reasoning.

Harness self-evolution. Strong-to-weak scaffolding also provides an empirical lens for studying harness self-evolution. Modern agentic coding environments such as Claude Code, Codex, and Cursor already shape model behavior through tools, prompts, workflows, and evaluation loops, but their design is still largely human-driven. Our setting asks whether models can automatically improve the harness around a target model, rather than merely produce answers within a fixed harness. This distinction is important: as agent systems become more capable, progress may come not only from improving the agent itself, but also from improving the environment that structures the agent's reasoning. By analyzing the scaffolds that builders create, we can identify which harness features consistently help, which add little value, and how different builders adapt the harness to the optimization target. In this way, automatic scaffold construction can inform human harness design while also pointing toward a future in which agents and their infrastructure co-evolve.

Toward scaffolding as a benchmark. The strong-to-weak scaffolding setup can also be developed into a standard benchmark for builder models. A benchmark could provide a workspace, a weak target model, a fixed downstream task, and a validation set. The builder would be asked to write and refine a scaffold using only the validation data, and the final score would be measured on a hidden full test set using the target model. This would evaluate a capability that ordinary task benchmarks often miss: not whether the builder can answer the questions directly, but whether it can improve another model's ability to answer them. Secondary metrics could include validation usage, inference cost, scaffold complexity, code length, robustness across repeats, and the fraction of work offloaded from the target model. Such a benchmark would make harness design measurable in a free-form yet rigorous way, capturing both final accuracy and efficiency of the builder's solution.

Two complementary routes to stronger systems. At a high level, there are two ways to improve performance on a task. One is to improve the model's internal capability; the other is to make the task easier for the model to execute. Most post-training work belongs to the first route, while harness and scaffold design belong to the second. Our work focuses on the second route, showing that a strong builder can partially substitute for target-model capability by reshaping the inference problem. These two routes should not be viewed as competitors. In the long run, models may be trained to use particular harnesses more effectively, while harnesses may be automatically optimized around the strengths and weaknesses of particular models. Recent work on using agents to design training recipes is one example of this interaction:

the harness can guide model improvement, and improved models can in turn design better harnesses. Strong-to-weak scaffolding is therefore a step toward studying this broader co-evolution of models and inference environments. It suggests that future progress will depend not only on building stronger models, but also on learning how to structure tasks so that available models can deploy their capabilities more reliably.

References

Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos Garea, Matthieu Geist, and Olivier Bachem. On-policy distillation of language models: Learning from self-generated mistakes. In *International Conference on Learning Representations*, volume 2024, pp. 21246–21263, 2024.

Chris L Baker, Rebecca Saxe, and Joshua B Tenenbaum. Action understanding as inverse planning. *Cognition*, 113(3):329–349, 2009.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pp. 17682–17690, 2024.

Collin Burns, Pavel Izmailov, Jan Hendrik Kirchner, Bowen Baker, Leo Gao, Leopold Aschenbrenner, Yining Chen, Adrien Ecoffet, Manas Joglekar, Jan Leike, et al. Weak-to-strong generalization: Eliciting strong capabilities with weak supervision. *arXiv preprint arXiv:2312.09390*, 2023.

Ruirui Chen, Weifeng Jiang, Chengwei Qin, and Cheston Tan. Theory of mind in large language models: Assessment and enhancement. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 31539–31558, 2025.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

Zhuang Chen, Jincenzi Wu, Jinfeng Zhou, Bosi Wen, Guanqun Bi, Gongyao Jiang, Yaru Cao, Mengting Hu, Yunghwei Lai, Zexuan Xiong, et al. Tombench: Benchmarking theory of mind in large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15959–15983, 2024.

Kanishk Gandhi, Jan-Philipp Fränken, Tobias Gerstenberg, and Noah Goodman. Understanding social reasoning in language models with language models. *Advances in Neural Information Processing Systems*, 36:13518–13529, 2023.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International conference on machine learning*, pp. 10764–10799. PMLR, 2023.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alex Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes. In *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 8003–8017, 2023.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, volume 2025, pp. 21344–21377, 2025.

Chuanyang Jin, Yutong Wu, Jing Cao, Jiannan Xiang, Yen-Ling Kuo, Zhiting Hu, Tomer Ullman, Antonio Torralba, Joshua Tenenbaum, and Tianmin Shu. Mmtom-qa: Multimodal theory of mind question answering. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16077–16102, 2024.

Zixuan Ke, Fangkai Jiao, Yifei Ming, Xuan-Phi Nguyen, Austin Xu, Do Xuan Long, Minzhi Li, Chengwei Qin, Peifeng Wang, Silvio Savarese, et al. A survey of frontiers in llm reasoning: Inference scaling, learning to reason, and agentic systems. *arXiv preprint arXiv:2504.09037*, 2025.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed prompting: A modular approach for solving complex tasks. *arXiv preprint arXiv:2210.02406*, 2022.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses, 2026. *URL https://arxiv. org/abs/2603.28052*, 2026.

Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. Faithful chain-of-thought reasoning. In *Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 305–329, 2023.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023.

Xuying Ning, Katherine Tieu, Dongqi Fu, Tianxin Wei, Zihao Li, Yuanchen Bei, Jiaru Zou, Mengting Ai, Zhining Liu, Ting-Wei Li, et al. Code as agent harness. *arXiv preprint arXiv:2605.18747*, 2026.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

Cheng Qian, Jiayu Liu, and Heng Ji. Userharness: Harnessing user minds for stronger agent theory-of-mind. *arXiv preprint arXiv:2605.27721*, 2026.

Matthew Riemer, Zahra Ashktorab, Djallel Bouneffouf, Payel Das, Miao Liu, Justin D Weisz, and Murray Campbell. Position: Theory of mind benchmarks are broken for large language models. *arXiv preprint arXiv:2412.19726*, 2024.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551, 2023.

Haojun Shi, Suyu Ye, Xinyu Fang, Chuanyang Jin, Leyla Isik, Yen-Ling Kuo, and Tianmin Shu. Muma-tom: Multi-modal multi-agent theory of mind. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 1510–1519, 2025.

John Sweller. Cognitive load during problem solving: Effects on learning. *Cognitive science*, 12(2):257–285, 1988.

Tomer Ullman. Large language models fail on trivial alterations to theory-of-mind tasks. *arXiv preprint arXiv:2302.08399*, 2023.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Yufan Wu, Yinghui He, Yilin Jia, Rada Mihalcea, Yulong Chen, and Naihao Deng. Hi-tom: A benchmark for evaluating higher-order theory of mind reasoning in large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 10691–10706, 2023.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023.

Yilun Yao, Xinyu Tan, Chao-Hsuan Liu, Yaoming Li, Zhengyang Wang, Wenhan Yu, Zhewen Tan, Yuxuan Tian, Guangxiang Zhao, Lin Sun, et al. Harness-bench: Measuring harness effects across models in realistic agent workflows. *arXiv preprint arXiv:2605.27922*, 2026.

Ling Yue, Kushal Raj Bhandari, Ching-Yun Ko, Dhaval Patel, Shuxin Lin, Nianjun Zhou, Jianxi Gao, Pin-Yu Chen, and Shaowu Pan. From static templates to dynamic runtime graphs: a survey of workflow optimization for llm agents. *arXiv preprint arXiv:2603.22386*, 2026.

Zhining Zhang, Chuanyang Jin, Mung Yao Jia, Shunchi Zhang, and Tianmin Shu. Autotom: Scaling model-based mental inference via automated agent modeling. *Advances in Neural Information Processing Systems*, 38:129659–129699, 2026.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*, 2022.

Appendix

A Instruction Details

In our main scaffold-building setting, we first place an instruction file in the workspace. The builder model then uses this file to construct the scaffold according to the provided instructions. The contents of the instruction file are described in detail below.

Initial Instruction Content

```
# Task: Build a Scaffold to Improve Target Model Benchmark Performance

You are entering a coding competition. Your goal is to build a scaffold/harness that improves the performance of a
downstream tested model on a hidden benchmark set. You have access to a small validation set, which is a random 2%
sample of the full benchmark. You should inspect the validation cases, design a scaffold, test it, analyze failures,
and recursively refine your scaffold before the final full-set evaluation.

## Goal and Scoring

Your final grade is based on:

1. **Average performance across the four benchmarks** -- primary criterion.
2. **Number of validation evaluations used** -- secondary criterion.

Each validation evaluation costs 1. Use validation runs carefully. Do not repeatedly test small or unprincipled
changes. Your scaffold should improve the target model's task performance and reduce its cognitive load.

## Workspace

You are assigned a workspace:

```text
/path/to/workspace
```

All code, outputs, logs, results, and notes must be saved under this workspace. You may not access other folders or
search the internet.

The workspace already contains:

```text
validation.jsonl
engines.py
```

The full benchmark file will have the same format as `validation.jsonl`.

Target Model

The tested model is:

```text
GPT-5.4-mini/Gemini-3.5-Flash
```

Please refer to `engines.py` for how to call the model.

Required Files

You must maintain these two files throughout the process:

1. `performance.csv`

Record every validation evaluation. It must include four columns for the four benchmark scores.

Suggested format:

```csv
run_id,split,benchmark_1,benchmark_2,benchmark_3,benchmark_4,average,notes
```

2. `optimization.md`

Keep a clear recursive refinement log. For each iteration, record:
```

```
* What you inspected
* What scaffold change you made
* Why you made it
* Validation result
* What you learned
* Next planned improvement

## Development Process

1. Inspect the workspace, especially `validation.jsonl` and `engines.py`.
2. Understand the input/output format, benchmark types, and evaluation requirements.
3. Build an initial scaffold and baseline evaluation script.
4. Run validation only after meaningful scaffold changes.
5. Analyze failures and refine the scaffold recursively.
6. Stop when performance plateaus or further changes risk overfitting.
7. Prepare the final full-set evaluation script.

Your scaffold may include prompt templates, task routing, answer-format enforcement, few-shot examples, verification
steps, deterministic preprocessing/postprocessing, or other methods that help the target model perform better.

Avoid hard-coding validation answers. No cheating. Optimize for generalization to the hidden full set.

## Final Deliverable

Create a bash script:

```text
full_run.sh
```

The evaluator should only need to insert or pass the full-set path to run your final harness.

Recommended usage:

```bash
bash full_run.sh /path/to/full_set.jsonl
```

The script should:

1. Load the given full-set file
2. Run your final scaffold on it
3. Save predictions/results inside the workspace
4. Append the full-set benchmark results to `performance.csv`
5. Full set testing should be parallel call of target models with max worker equals 16
6. Full set final results or analysis output should be saved in independent folder `final_eval`

Final Reminder

You have only one final chance on the hidden full set. Be strategic: inspect carefully, make each validation run count
, recursively improve the scaffold, and prioritize robust benchmark-wide performance over validation overfitting.
```

B Complete Per-Run Results

Table 6 lists all 72 runs, including their factor coordinates, graded per-benchmark accuracies, macro-average accuracies, and the number of validation evaluations used by the builder.

Table 6: All runs, sorted by target / builder / platform / repeat. "Avg." is the macro average of the four benchmark accuracies (primary metric). "Val." is the number of validation evaluations used.

Run	Platform	Builder	Target	Rep	BigToM	Hi-ToM	MMToM	MuMA	Avg.	Val.
gemini35flash-gemini35flash	Cursor	Gemini-3.5-flash	gemini-3.5-flash	1	0.874	0.738	0.770	0.990	**0.843**	4
gemini35flash-gemini35flash-2	Cursor	Gemini-3.5-flash	gemini-3.5-flash	2	1.000	0.699	0.720	0.916	**0.834**	3
gemini35flash-gemini35flash-3	Cursor	Gemini-3.5-flash	gemini-3.5-flash	3	0.971	0.816	0.977	0.991	**0.939**	2
gemini31pro-gemini35flash	Cursor	Gemini-3.1-Pro	gemini-3.5-flash	1	0.870	0.697	0.772	0.959	**0.825**	4
gemini31pro-gemini35flash-2	Cursor	Gemini-3.1-Pro	gemini-3.5-flash	2	0.899	0.818	0.933	0.947	**0.899**	5
gemini31pro-15544413-3	Cursor	Gemini-3.1-Pro	gemini-3.5-flash	3	0.914	0.832	0.948	0.978	**0.918**	25
gpt55-gemini35flash	Cursor	GPT-5.5	gemini-3.5-flash	1	1.000	0.848	0.768	0.934	**0.888**	6
gpt55-gemini35flash-2	Cursor	GPT-5.5	gemini-3.5-flash	2	1.000	0.824	0.940	0.908	**0.918**	3
gpt55-gemini35flash-3	Cursor	GPT-5.5	gemini-3.5-flash	3	1.000	0.824	0.883	0.880	**0.897**	3
grok01-gemini35flash	Cursor	Grok-0.1	gemini-3.5-flash	1	0.876	0.813	0.813	0.971	**0.868**	2
grok01-gemini35flash-2	Cursor	Grok-0.1	gemini-3.5-flash	2	0.513	0.634	0.640	0.952	**0.685**	3
grok01-gemini35flash-3	Cursor	Grok-0.1	gemini-3.5-flash	3	0.896	0.651	0.637	0.969	**0.788**	3
opus47-gemini35flash	Cursor	Opus-4.7 (x-high)	gemini-3.5-flash	1	0.981	0.817	0.943	0.986	**0.932**	2
opus47-gemini35flash-2	Cursor	Opus-4.7 (x-high)	gemini-3.5-flash	2	0.985	0.775	0.978	0.992	**0.933**	3
opus47-gemini35flash-3	Cursor	Opus-4.7 (x-high)	gemini-3.5-flash	3	0.959	0.814	0.873	0.969	**0.904**	4
codex53-gpt54mini	Cursor	Codex-5.3	gpt-5.4-mini	1	1.000	0.587	0.367	0.518	**0.618**	4
codex53-gpt54mini-2	Cursor	Codex-5.3	gpt-5.4-mini	2	1.000	0.552	0.598	0.506	**0.664**	4
codex53-gpt54mini-3	Cursor	Codex-5.3	gpt-5.4-mini	3	0.900	0.804	0.358	0.498	**0.640**	7
gptcodex-codex53-gpt54mini	GPT Codex	Codex-5.3	gpt-5.4-mini	1	1.000	0.560	0.762	0.541	**0.716**	4
gptcodex-codex53-gpt54mini-2	GPT Codex	Codex-5.3	gpt-5.4-mini	2	1.000	0.568	0.597	0.501	**0.667**	2
gptcodex-codex53-gpt54mini-3	GPT Codex	Codex-5.3	gpt-5.4-mini	3	1.000	0.682	0.695	0.606	**0.746**	5
gemini35flash-gpt54mini	Cursor	Gemini-3.5-flash	gpt-5.4-mini	1	0.958	0.665	0.697	0.670	**0.747**	4
gemini35flash-gpt54mini-2	Cursor	Gemini-3.5-flash	gpt-5.4-mini	2	1.000	0.731	0.790	0.844	**0.841**	3
gemini35flash-gpt54mini-3	Cursor	Gemini-3.5-flash	gpt-5.4-mini	3	1.000	0.739	0.847	0.817	**0.851**	5
gemini31pro-gpt54mini	Cursor	Gemini-3.1-Pro	gpt-5.4-mini	1	0.926	0.682	0.518	0.581	**0.677**	4
gemini31pro-gpt54mini-2	Cursor	Gemini-3.1-Pro	gpt-5.4-mini	2	0.932	0.776	0.657	0.599	**0.741**	5
gemini31pro-gpt54mini-3	Cursor	Gemini-3.1-Pro	gpt-5.4-mini	3	0.872	0.738	0.680	0.600	**0.722**	11
gpt54mini-gpt54mini	Cursor	GPT-5.4-mini	gpt-5.4-mini	1	1.000	0.831	0.803	0.414	**0.762**	6
gpt54mini-gpt54mini-2	Cursor	GPT-5.4-mini	gpt-5.4-mini	2	1.000	0.643	0.560	0.607	**0.702**	4
gpt54mini-gpt54mini-3	Cursor	GPT-5.4-mini	gpt-5.4-mini	3	0.883	0.575	0.682	0.468	**0.652**	7
gptcodex-gpt54mini-gpt54mini	GPT Codex	GPT-5.4-mini	gpt-5.4-mini	1	1.000	0.556	0.447	0.454	**0.614**	6
gptcodex-gpt54mini-gpt54mini-2	GPT Codex	GPT-5.4-mini	gpt-5.4-mini	2	1.000	0.571	0.400	0.450	**0.605**	3
gptcodex-gpt54mini-gpt54mini-3	GPT Codex	GPT-5.4-mini	gpt-5.4-mini	3	1.000	0.720	0.823	0.452	**0.749**	2
gpt55-gpt54mini	Cursor	GPT-5.5	gpt-5.4-mini	1	1.000	0.831	0.597	0.921	**0.837**	15
gpt55-gpt54mini-2	Cursor	GPT-5.5	gpt-5.4-mini	2	1.000	0.831	0.873	0.928	**0.908**	3
gpt55-gpt54mini-3	Cursor	GPT-5.5	gpt-5.4-mini	3	1.000	0.824	0.845	0.612	**0.820**	3
gptcodex-gpt55-gpt54mini	GPT Codex	GPT-5.5	gpt-5.4-mini	1	1.000	0.843	0.888	0.916	**0.912**	7
gptcodex-gpt55-gpt54mini-2	GPT Codex	GPT-5.5	gpt-5.4-mini	2	1.000	0.660	0.912	0.918	**0.872**	2
gptcodex-gpt55-gpt54mini-3	GPT Codex	GPT-5.5	gpt-5.4-mini	3	1.000	0.828	0.937	0.848	**0.903**	5
grok01-gpt54mini	Cursor	Grok-0.1	gpt-5.4-mini	1	0.496	0.623	0.593	0.596	**0.577**	2
grok01-gpt54mini-2	Cursor	Grok-0.1	gpt-5.4-mini	2	0.492	0.620	0.487	0.459	**0.514**	3
grok01-gpt54mini-3	Cursor	Grok-0.1	gpt-5.4-mini	3	0.852	0.534	0.530	0.479	**0.599**	3
claudecode-opus47-gpt54mini	Claude Code	Opus-4.7 (x-high)	gpt-5.4-mini	1	0.991	0.785	0.830	0.880	**0.871**	6
claudecode-opus47-gpt54mini-2	Claude Code	Opus-4.7 (x-high)	gpt-5.4-mini	2	0.996	0.797	0.775	0.900	**0.867**	6
claudecode-opus47-gpt54mini-3	Claude Code	Opus-4.7 (x-high)	gpt-5.4-mini	3	1.000	0.782	0.787	0.947	**0.879**	7
opus47-gpt54mini	Cursor	Opus-4.7 (x-high)	gpt-5.4-mini	1	0.902	0.722	0.905	0.822	**0.838**	5
opus47-gpt54mini-2	Cursor	Opus-4.7 (x-high)	gpt-5.4-mini	2	1.000	0.916	0.728	0.818	**0.865**	5
opus47-gpt54mini-3	Cursor	Opus-4.7 (x-high)	gpt-5.4-mini	3	0.929	0.747	0.703	0.890	**0.817**	4
claudecode-opus47high-gpt54mini	Claude Code	Opus-4.7 (high)	gpt-5.4-mini	1	0.966	0.782	0.688	0.940	**0.844**	4
claudecode-opus47high-gpt54mini-2	Claude Code	Opus-4.7 (high)	gpt-5.4-mini	2	0.967	0.726	0.738	0.784	**0.804**	6
claudecode-opus47high-gpt54mini-3	Claude Code	Opus-4.7 (high)	gpt-5.4-mini	3	1.000	0.743	0.818	0.762	**0.831**	3
opus47high-gpt54mini	Cursor	Opus-4.7 (high)	gpt-5.4-mini	1	0.877	0.659	0.772	0.656	**0.741**	5
opus47high-gpt54mini-2	Cursor	Opus-4.7 (high)	gpt-5.4-mini	2	0.835	0.771	0.812	0.851	**0.817**	4
opus47high-gpt54mini-3	Cursor	Opus-4.7 (high)	gpt-5.4-mini	3	0.891	0.751	0.832	0.750	**0.806**	5
claudecode-opus47low-gpt54mini	Claude Code	Opus-4.7 (low)	gpt-5.4-mini	1	0.869	0.709	0.647	0.571	**0.699**	2
claudecode-opus47low-gpt54mini-2	Claude Code	Opus-4.7 (low)	gpt-5.4-mini	2	0.880	0.729	0.513	0.634	**0.689**	4
claudecode-opus47low-gpt54mini-3	Claude Code	Opus-4.7 (low)	gpt-5.4-mini	3	0.873	0.623	0.685	0.591	**0.693**	2
opus47low-gpt54mini	Cursor	Opus-4.7 (low)	gpt-5.4-mini	1	0.872	0.738	0.667	0.836	**0.778**	4
opus47low-gpt54mini-2	Cursor	Opus-4.7 (low)	gpt-5.4-mini	2	0.866	0.677	0.630	0.613	**0.697**	3
opus47low-gpt54mini-3	Cursor	Opus-4.7 (low)	gpt-5.4-mini	3	0.963	0.652	0.512	0.708	**0.708**	4
claudecode-opus47medium-gpt54mini	Claude Code	Opus-4.7 (med)	gpt-5.4-mini	1	0.859	0.703	0.635	0.888	**0.771**	5
claudecode-opus47medium-gpt54mini-2	Claude Code	Opus-4.7 (med)	gpt-5.4-mini	2	0.968	0.705	0.783	0.792	**0.812**	6
claudecode-opus47medium-gpt54mini-3	Claude Code	Opus-4.7 (med)	gpt-5.4-mini	3	0.945	0.748	0.823	0.937	**0.863**	5
opus47medium-gpt54mini	Cursor	Opus-4.7 (med)	gpt-5.4-mini	1	1.000	0.762	0.693	0.738	**0.798**	2
opus47medium-gpt54mini-2	Cursor	Opus-4.7 (med)	gpt-5.4-mini	2	0.893	0.566	0.733	0.464	**0.664**	7
opus47medium-gpt54mini-3	Cursor	Opus-4.7 (med)	gpt-5.4-mini	3	1.000	0.710	0.837	0.847	**0.848**	6
claudecode-sonnet46-gpt54mini	Claude Code	Sonnet-4.6	gpt-5.4-mini	1	0.994	0.586	0.598	0.499	**0.669**	11
claudecode-sonnet46-gpt54mini-2	Claude Code	Sonnet-4.6	gpt-5.4-mini	2	1.000	0.700	0.830	0.950	**0.870**	8
claudecode-sonnet46-gpt54mini-3	Claude Code	Sonnet-4.6	gpt-5.4-mini	3	0.962	0.731	0.793	0.830	**0.829**	4
sonnet46-gpt54mini	Cursor	Sonnet-4.6	gpt-5.4-mini	1	0.981	0.729	0.642	0.810	**0.790**	8
sonnet46-gpt54mini-2	Cursor	Sonnet-4.6	gpt-5.4-mini	2	0.968	0.763	0.850	0.912	**0.873**	5
sonnet46-gpt54mini-3	Cursor	Sonnet-4.6	gpt-5.4-mini	3	0.959	0.762	0.737	0.858	**0.829**	5